Designation: E2078 – 00 (Reapproved 2016)

Standard Guide for Analytical Data Interchange Protocol for Mass Spectrometric Data[1]

This standard is issued under the fixed designation E2078; the number immediately following the designation indicates the year of original adoption or, in the case of revision, the year of last revision. A number in parentheses indicates the year of last reapproval. A superscript epsilon (ε) indicates an editorial change since the last revision or reapproval.

1. Scope

1.1 This guide covers the implementation of the Mass Spectrometric Data Protocol in analytical software applications. Implementation of this protocol requires:

1.1.1 Specification E2077, which contains the full set of data definitions. The mass spectrometric data protocol is not based upon any specific implementation; it is designed to be independent of any particular implementation so that implementations can change as technology evolves. The protocol is implemented in categories to speed its acceptance through actual use.

1.1.2 Specification E2077 contains a full description of the contents of the data communications protocol, including the analytical information categories with data elements and their attributes for most aspects of mass spectrometric tests.

1.2 The analytical information categories are a practical convenience for breaking down the standardization process into smaller, more manageable pieces. It is easier for developers to build consensus and produce working systems based on smaller information sets, without the burden and complexity of the hundreds of data elements contained in all the categories. The categories also assist vendors and end users in using the guide in their computing environments.

1.3 The network common data format (NetCDF) data interchange system is the container used to communicate data between applications in a way that is independent of both computer architectures and end-user applications. In essence, it is a special type of application designed for data interchange.

1.4 The common data language (CDL) template for mass spectrometry is a language specification of the mass spectrometry dataset being interchanged. With the use of the NetCDF utilities, this human-readable template can be used to generate an equivalent binary file and the software subroutine calls needed for input and output of data in analytical applications.

2. Referenced Documents

2.1 *ASTM Standards:*[2]

E2077 Specification for Analytical Data Interchange Protocol for Mass Spectrometric Data

2.2 *Other Standard:*

NetCDF User's Guide[3]

2.3 *ISO Standards:*[4]

8601:1988 Data elements and interchange formats, (First edition published 1988-06-15; with Technical Corrigendum 1 published 1991-05-01)

3. List of Contents and Use

3.1 *NetCDF Toolkit*—The protocol is an application programming interface (API) layered on top of the public domain NetCDF toolkit. NetCDF is a set of tools that facilitate reading or writing platform-independent, self-describing data files. All data in a NetCDF file is written using the external data representation (XDR). XDR was developed by Sun Microsystems and is used for platform-independent file systems for all workstations and personal computers. Each NetCDF data element is self-describing - it has a name, type, and dimensionality. A NetCDF file contains three parts: a *dimensions* section, which defines the names and size of all dimensions used to describe variables; a *variables* section, which defines the names, data types, dimensionality, and attributes for all variables used in the file; and finally, a *data* section, which contains the actual values assigned to the variables. Attributes are numbers or strings which augment the description of variables or the file as a whole.

3.1.1 For example, a variable "x_axis_ values" might contain an array of numbers representing the abscissa of a two-dimensional data set. It would have a dimension, possibly named "x_axis_size," which would specify the number of

[1] This guide is under the jurisdiction of ASTM Committee E13 on Molecular Spectroscopy and Separation Science and is the direct responsibility of Subcommittee E13.15 on Analytical Data.

Current edition approved April 1, 2016. Published June 2016. Originally approved in 2000. Last previous edition approved in 2010 as E2078 – 00 (2010). DOI: 10.1520/E2078-00R16.

[2] For referenced ASTM standards, visit the ASTM website, www.astm.org, or contact ASTM Customer Service at service@astm.org. For *Annual Book of ASTM Standards* volume information, refer to the standard's Document Summary page on the ASTM website.

[3] Available from Russell K. Rew, Unidata Program Center, University Corporation for Atmospheric Research, P.O. Box 3000, Boulder, CO 80307-3000, http://www.unidata.ucar.edu/.

[4] Available from International Organization for Standardization (ISO), ISO Central Secretariat, BIBC II, Chemin de Blandonnet 8, CP 401, 1214 Vernier, Geneva, Switzerland, http://www.iso.org.

abscissa points. The variable might have some descriptive attributes, such as "units" (with a value of" Seconds," perhaps), "scale_factor" (with a value of 1000.0, specifying that all stored abscissa values should be multiplied by 1000.0 to get the actual value), or "long_name" (with value" Time", which might be used to label the abscissa when drawing a plot).

3.1.2 The NetCDF toolkit has been placed in the public domain by the Unidata Program Center, a non-profit software support organization for the University Corporation for Atmospheric Research. The Unidata Program Center is funded by the National Science Foundation, National Center for Atmospheric Research, and other organizations and provides ongoing development and support of NetCDF and related tools.

3.1.3 The NetCDF version currently supported in this implementation is 2.3.2.

3.2 *Data Structures*—Each of the analytical information class tables in the specification document has a corresponding data structure; however, not every field in each table has a corresponding data element in a structure, and the data structures may have elements that do not appear in any class table. Most of these differences are due to details of the implementation which could not be hidden.

3.2.1 The data structures provide the mapping between the attribute name and data type described in the specification and the field and actual data type in the file. The actual NetCDF dimension, variable, and attribute names are hidden from the API level. These names in fact are irrelevant for application programs; it is the data structure which provides the information interchange between the application and the file.

3.2.2 Each data structure and its mapping to an analytical information class are described in detail later in this guide.

3.2.3 *Application Programming Interface Functions:*

3.2.3.1 The application programming interface provides programmatic access to the contents of the files. Mass spectral data occurs in three forms: global information, which relates to the contents of the entire file, information which describes each part of a multi-component instrument, and information which changes on a scan-by-scan basis for spectra and library entries. API functions are provided for opening a file for reading or writing; closing a file; reading and writing global, per-component instrument, and per-scan spectral and library information; initializing and clearing data structure contents; and a few miscellaneous utility functions. Each of these functions is described in detail in a later section of this guide.

3.2.4 *Enumerated Sets*—Many of the attributes listed in the Analytical Data Interchange Protocol for Mass Spectrometric Data specification have an enumerated set of associated values. The attribute may take only one value from that restricted set. In the implementation, each such attribute is defined as a formal C type, and the allowed values are defined as an enumerated set of that formal type. Each enumerated value is associated with a unique string literal, and it is these string literals, not the enumeration values, which are written to or read from the file. This practice both enforces the use of the proper enumeration values and follows the NetCDF dictum that files be self-describing. If the enumeration values were written instead of the strings, then some lookup mechanism would be required external to the NetCDF file to translate the number into something meaningful.

4. Conventions

4.1 The format convention adopted in this guide is as follows:

(1) Normal text is presented in this font (Times New Roman).

(2) API symbols (functions, formal types, etc.) are presented in **boldface Helvetica font**.

(3) Parameters to API functions are presented in *italic Helvetica font*.

(4) Example code is presented in normal Helvetica font.

4.2 *Other Conventions*—All indices begin at zero (C convention). In several data structures, a **scan_no** or **inst_no** element must be loaded before reading or writing. This identifies the scan or instrument component number for which data will be read or written. In all cases, scan or instrument component numbers begin at zero.

4.2.1 All date/time stamps are formatted using the ISO standard 8601 format referenced in the specification. An API utility function is provided for conversion between date/time information in numeric form and ISO-8601 string format (see **ms_convert_date()**, below).

5. Mass Spectrometric Data Protocol Distribution Kit

5.1 It is intended that potential users of this implementation can obtain a complete NetCDF and API distribution kit from various instrument vendors' websites. Information on how to obtain the kit will be posted on the ASTM website (www.astm.org) under Committee E01.25.

5.2 The Analytical Data Interchange Protocol for Mass Spectrometric Data distribution kit contains:

5.2.1 *Software*—NetCDF distribution kit from Unidata (with the modified makefile needed to make the kit compile out of the box).

5.2.2 *NetCDF User's Guide*—supplied by Unidata Program Center.

5.2.3 *Specification E2077.*

5.2.4 *Guide E2078.*

6. Hardware and Software

6.1 This section describes the hardware and software configurations used for testing. In general, the NetCDF system puts very few requirements on the hardware because most routines are left on disk. Only routines being used at any particular time are kept in memory. Any limitations found were typically those not imposed by NetCDF but ones imposed by the operating system or environment.

6.1.1 *Hardware (Personal Computers)*—The personal computer system hardware originally used for testing was:

6.1.1.1 Intel 80286 processor,

6.1.1.2 640K minimum,

6.1.1.3 Monochrome, EGA, VGA graphics,

6.1.1.4 20 megabyte minimum, 80 megabyte hard-disk is typical, and

6.1.1.5 A mouse (optional).

6.1.1.6 NetCDF works well on AT-class machines and higher. NetCDF does not have the items in 6.1.1.1 – 6.1.1.5 as requirements. These are just the minimum, base-level systems that were used.

6.1.2 *Software*—NetCDF runs on MS-DOS, OS/2, Macintosh, Windows 95, and Windows NT operating systems for personal computers. NetCDF was originally ported from UNIX to DOS running on an IBM-PS/2 Model 80. It was recently ported to the Macintosh OS. NetCDF is written in the C programming language, and there are FORTRAN jackets available for applications that want to use FORTRAN calls. The personal computer software originally employed for testing and developing NetCDF applications was:

6.1.2.1 Microsoft DOS V3.3 or above,

6.1.2.2 Microsoft C Compiler V6.0,

6.1.2.3 Microsoft Windows V3.0,

6.1.2.4 Microsoft Windows SDK, and

6.1.2.5 NetCDF Version 2.0.1.

6.1.3 *Workstations and Servers*—NetCDF runs easily on UNIX workstations such as Sun 3, Sun 4, VAXstations, DECstation 3100, VAXstation II running ULTRIX or VMS, and IBM RS/6000. There are no particular hardware requirements for workstation class machines, since all workstations have the minimum hardware outlined for personal computers in 6.1.1.

7. Significance and Use

7.1 *General Coding Guidelines*—The NetCDF libraries are supplied to developers as source code. End users receive the libraries in compiled binary form as part of a vendor's application.

7.1.1 Developers setting out to write a program to convert their data files to the Mass Spectrometric Data Protocol should consider using the NetCDF utilities ncgen and ncdump. After developers create the NetCDF file they should use the ncdump program to generate the ASCII representation of the data file, and examine it to ensure the data are being correctly put into the file.

7.2 *Make Files for NetCDF Libraries and Utilities*—In general the compilation is straightforward. The make files were modified after they were received from the Unidata Corporation, because they did not compile the first time on PCs. The changes needed to get the Unidata distribution to run on DOS are (*1*) rename the file MAKEFILE to UNIX.MK, and (*2*) rename MSOFT.MK to MAKEFILE, and then run NMAKE. The default switches in the Unidata distribution use the switches for the floating point coprocessor and Microsoft Windows options.

7.2.1 The protocol kit contains some complete makefile examples for Microsoft C V6.0 running on DOS. The Microsoft C V6.0 compiler manual should be consulted for the exact meaning of the compiler and linker options.

7.2.2 The VMS and SunOS compilation instructions are in directories for those operating systems.

7.3 *NetCDF Library Build Order*—The NetCDF libraries must be built in a specific order. The correct order to build the NetCDF directories is:

```
UTIL
XDR
SRC
NCDUMP
NCGEN
NCTEST
```

7.3.1 The UTIL and XDR makefiles work as distributed using NMAKE with Microsoft C V6.0.

8. CDL Template Structure

8.1 A NetCDF template is built from CDL statements and is structured into three sections: (*1*) dimension declarations, (*2*) variable declarations, and (*3*) the data section.

8.2 A few points of clarification about the CDL language are given here to facilitate its understanding. For more in-depth information on CDL, please consult the *NetCDF User's Guide*.

8.2.1 A NetCDF template starts with the word "NetCDF" followed by the dataset name.

8.2.2 CDL comments are indicated by two forward slash characters (//).

8.2.3 Section indicators (dimensions:, variables:, and data:) end with a colon character (:). These are the only tokens that end with a colon character.

8.2.4 Statements within sections end with the semicolon character (;).

8.2.5 Variable names beginning with numbers must be preceded by an underline character (_). Otherwise the ncgen parser will flag an error.

8.2.5.1 Underline characters were chosen for this protocol over hyphen characters, because some compilers may interpret hyphens as subtraction operators. The feature of CDL that allows implicit numerical datatyping of attributes in not being used in the first version of the template. Instead, all floating point attributes are being handled as strings. This forces programmers to explicitly type variables, thereby encouraging more deliberate programming styles. For example:

```
:aia_template_revision = "0.8"; //M12345
:netcdf_revision = "2.0.1"; //M12345
```

Consult the *NetCDF User's Guide* for more complete information on CDL syntax and usage.

8.2.6 Underline characters only can be used as separators between words within variable names, like:

```
aia-template-revision, or aia_template_revision.
```

8.2.7 Numerical data types for single values can be declared implicitly by putting numbers on the right side of an assignment statement, like:

```
peak_number=2; //number of peaks
```

These numerical datatypes can be floating point or integer values, and can be implicitly datatyped as such.

```
:floating_point_attribute = 1.11; //M12345
```

8.2.8 Numerical data types can be declared explicitly by preceding the variable definition by its data type. Datatype assignments can be for either single value variable definitions or for array variable definitions, for example:

```
float detector_maximum_value;
float ordinate_values(point_number);
```

There is also a way to explicitly declare datatypes on the right side of an assignment operator. Please consult the NetCDF User's Guide for details.

8.2.9 Metadata are associated with a particular variable by attaching it to that variable with a colon character, for example:

```
ordinate_values:uniform_sampling_flag=" Y";
ordinate_values:autosampler_position=" 1.01";
```

8.2.10 Global metadata can be declared simply by not attaching it to any variable, for example:

```
:aia_template_revision= "0.8"; //M12345
```

8.2.11 String attributes can be as long as needed, and are declared by enclosing the strings in quotation marks, for example:

```
:retention_unit= "time in seconds";
```

8.3 *Notes About the Mass Spectrometric Data Protocol Usage of CDL*—Some mandatory indicator codes (M-codes) for data elements such as M1234, M1, etc., are given in some comment fields of the protocol template. These are *not* part of CDL syntax. These refer to whether a given data element is mandatory for particular information categories, for example, M1234 specifies that the data element is mandatory for Categories 1, 2, 3, and 4. These M-codes are also given in the Specification E2077.

9. Other Usage Tips

9.1 *Filename Extensions*—The recommended filename extension is ".cdf," so that the full name for a NetCDF file would be "filename.cdf." This is used so that parsers used to select files can parse filenames based on the ".cdf" extension rather than some other non-standard file extension.

9.2 *Handling of a Missing Variable*—The absence of a variable implies that it is not in the file. For example, if a get operation returns an error condition, this implies that the variable does not exist in that file.

9.3 *Performance Tip For Data Value Access*—The point_number dimension was originally declared as" unlimited"; however, this was changed to a finite value because this change allows getting and putting of an entire array at once. This change is minor and will not affect programs, however, it greatly improves performance. Using point_num as unlimited restricts get/put operations to single values at a time, that is, they are treated as records.

9.4 *Getting Valid Date Time Stamps*—In order to get the correct date-time-stamp values in datasets originating from DOS and OS/2 systems, the environment variable for time zone must be set correctly. The recommended procedure is to set the offset form Greenwich Mean Time (GMT) *at product installation time*. Some examples of how to set the time-zone environment variable are as follows:

9.4.1 The command "DOS-PROMPT>tz pst 8 pdt" sets the time-zone variable to have a GMT offset of Pacific Standard Time (pst), with a value of 8 h offset from GMT, at Pacific Daylight Time (pdt).

9.4.2 The command "DOS-PROMPT>tz est 5 edt" sets the time-zone variable to have a GMT offset of Eastern Standard Time (est), with a value of 5 h offset for GMT, at Eastern Daylight Time (edt).

10. Data Structures

10.1 The protocol data structures form the heart of the information interchange. When reading a file, the API loads information from the file into the fields of the data structures. The application program is responsible for preparing the data structures for use by the API functions, for removing the information returned by the API, and for clearing the data structures for subsequent use in another API call. When writing a file, the API extracts information from the data structures and writes it to the file. The application program is again responsible for preparing and loading information into the data structures and then clearing them after the API call.

10.1.1 It is important to emphasize that **the application program is responsible for the data structure contents**. API functions are provided to initialize and clear data structure contents. These functions make several assumptions; in order to ensure proper interaction of the protocol and applications and to avoid memory allocation errors, these rules must be followed:

(1) When using API functions to read from a file, **the API allocates memory for character strings and numeric arrays**. It is the applications' responsibility to free this memory (using **free()**) after the data structure contents have been used. Failure to do so will result in a memory leak.

(2) When using API functions to write to a file, **the API assumes that the application has allocated memory for character strings and numeric arrays**. The API file writing functions do not free this memory; however, API functions which clear data structure contents assume that any non-**NULL** pointers reference allocated memory, and will free the memory and clear the pointers. It is acceptable to use pointers to statically declared storage, but the application must ensure that pointers to such storage are not passed to the data structure initialization routines.

(3) When reading an interchange file, a **NULL** pointer will be returned in character string or numeric array fields for which no data is present in the file.

(4) When writing an interchange file, **NULL** pointers may be passed in most cases for character strings or numeric arrays for which no data are present or which are inappropriate or inapplicable. Exceptions are noted in the sections below.

10.2 *Administrative Information Class*—**MS_Admin_Data**. The **MS_Admin_Data** data structure maps the administrative information class attributes and data types. It is only referenced once in code, either when reading from or writing to a file. **MS_Admin_Data** is a typedef. Table 1 describes the data structure fields, formal types, and mapping to administrative information class attributes.

10.2.1 **ms_admin_expt_t:** *Experiment Type*—The default value is shown in grey. See Table 2.

10.3 *Instrument-ID Information Class*—**MS_Instrument_Data**. Instrument data occurs on a per-component basis (that is, an instrument may be composed of one or more instrument components. The total number of components is defined using the **ms_open_write ()** or is read from the interchange file during **ms_open_read ()** (see below). When writing, the **MS_Instrument_Data** structure is filled with the data for

TABLE 1 Data Structure Fields

MS_Admin_Data				
Type	Field Name	E[A]	M[B]	Specification Attribute
char[C]	dataset_completeness		x	data set completeness[C]
char[C]	ms_template_revision		x	template revision level[C]
char[C]	comments			administrative comments
char[C]	dataset_origin			data set origin
char[C]	dataset_owner			data set owner
char[C]	experiment_title			experiment title
char[C]	experiment_date_time		x	experiment date/time stamp
(1)[D]	experiment_type	x		experiment type
Char[C]	experiment_x_ref_0			experiment cross-references[E]
char[C]	experiment_x_ref_1			experiment cross-references[E]
char[C]	experiment_x_ref_2			experiment cross-references[E]
char[C]	experiment_x_ref_3			experiment cross-references[E]
char[C]	netcdf_date_time		x	NetCDF file date/time stamp
char[C]	netcdf_revision		x	NetCDF revision level[C]
char[C]	operator_name			operator name
char[C]	source_file_reference			source file reference
char[C]	source_file_format			source file format
char[C]	source_file_date_time			source file date/time stamp
char[C]	external_file_ref_0			external file references[E]
char[C]	external_file_ref_1			external file references[E]
char[C]	external_file_ref_2			external file references[E]
char[C]	external_file_ref_3			external file references[E]
char[C]	languages		x	languages[C]
Long	number_times_processed			number of times processed
Long	number_times_calibrated			number of times calibrated
char[C]	calibration_history_0			calibration history[E]
char[C]	calibration_history_1			calibration history[E]
char[C]	calibration_history_2			calibration history[E]
char[C]	calibration_history_3			calibration history[E]
char[C]	post_expt_program_name			post-experiment program name
char[C]	pre_expt_program_name			pre-experiment program name
char[C]	error_log			error log
Long	number_instrument_components			(none)[F]

[A] The E column indicates that this is an enumerated set field. It is recorded in the interchange file as a string literal, but is represented as an enumerated type in the data structure.
[B] The M column indicates that this is a required field. When reading or writing an interchange file, an error will be generated if a mandatory field is not filled in.
[C] These fields are present in the data structure, but do not need to be filled by the application program when writing an interchange file. The API fills these fields with the appropriate values. However, on reading a file, the contents of these fields are filled with allocated strings, and must be freed by the caller.
[D] (*1*) Data type is **ms_admin_expt_t**
[E] These fields are defined in the specification as "string array" types. For convenience of the implementation and to conserve space in the interchange file, they are defined as separate strings here.
[F] The number of instrument components is returned in this field *only* when *reading* an interchange file. It is not used when writing files.

TABLE 2 Experiment Type

ms_admin_expt_t		
Value	String literal	Specification description
expt_centroid	Centroided Mass Spectrum	centroided mass spectrum
expt_continuum	Continuum Mass Spectrum	continuum mass spectrum
expt_library	Library Mass Spectrum	library mass spectrum

each instrument component in turn, and then is written to the interchange file using successive API calls. When reading, the number of instrument components is returned in the **MS_Admin_Data** structure. Data for each component is returned with successive API calls.

10.3.1 When both reading and writing, the **inst_no** field must be filled with the index number of the component. These index numbers are arbitrary, but must be sequential beginning with zero. Other fields must be filled in by the application when writing, or are filled by the API when reading. The application is responsible for initializing the **MS_Instrument_Data** structure before use, and for clearing its contents between API calls. See Table 3.

10.4 *Sample Description Information Class*—**MS_Sample_Data**. The **MS_Sample_Data** structure occurs once per interchange file. See Table 4.

10.4.1 **ms_sample_state_t:**—*sample state*. See Table 5.

10.5 *Test Method Information Class*—**MS_Test_Data**. The **MS_Test_Data** structure occurs once per interchange file. Depending on the specifics of the experiment which generated the data set, many fields will most likely be inappropriate or inapplicable. Only those fields which are appropriate need be changed from the default values set during initialization, and only those which have non-default values will be read from or written to the interchange file. See Table 6.

10.5.1 **ms_test_separation_t:**— *separation method*. See Table 7.

10.5.2 **ms_test_inlet_t:**—*mass spectrometer inlet*. See Table 8.

10.5.3 **ms_test_ioniz_t:**—*ionization method*. See Table 9.

10.5.4 **ms_test_polarity_t:**—*ionization polarity*. See Table 10.

10.5.5 **ms_test_detector_t:**—*detector type*. See Table 11.

10.5.6 **ms_test_res_t:**—*resolution type*. See Table 12.

10.5.7 **ms_test_function_t:**—*scan function*. See Table 13.

TABLE 3 MS_Instrument_Data

NOTE 1—There are no enumerated sets associated with **MS_Instrument_Data**.

NOTE 2—All string fields in this structure are restricted to 32 characters (including terminal NULL).

Type	Field Name	E	M	Specification Attribute
Long	inst_no		x	instrument component number
char[A]	name			instrument component name
char[A]	id			instrument component id
char[A]	manufacturer			instrument component manufacturer
char[A]	model_number			instrument component model number
char[A]	serial_number			instrument component serial number
char[A]	comments			instrument component id comments
char[A]	software_version			instrument component software version
char[A]	firmware_version			instrument component firmware version
char[A]	operating_system			operating system revision
char[A]	application_software			application software revision

[A] These fields are present in the data structure, but do not need to be filled by the application program when writing an interchange file. The API fills these fields with the appropriate values. However, on reading a file, the contents of these fields are filled with allocated strings, and must be freed by the caller.

TABLE 4 MS_Sample_Data

Type	Field Name	E	M	Specification Attribute
char[A]	owner			sample owner
char[A]	receipt_date_time			sample receipt date/time stamp
char[A]	internal_id			internal sample id
char[A]	external_id			external sample id
char[A]	procedure_name			sampling procedure name
char[A]	prep_procedure			sample preparation procedure
(1)[B]	state	x		sample state
char[A]	matrix			sample matrix
char[A]	storage			sample storage information
char[A]	disposal			sample disposal information
char[A]	history			sample history
char[A]	prep_comments			sample preparation comments
char[A]	comments			sample id comments
char[A]	manual_handling			manual handling precautions

[A] These fields are present in the data structure, but do not need to be filled by the application program when writing an interchange file. The API fills these fields with the appropriate values. However, on reading a file, the contents of these fields are filled with allocated strings, and must be freed by the caller.
[B] Data type is ms_sample_state_t

TABLE 5 ms_sample_state_t

Value	String Literal	Specification Description
state_solid	Solid	solid
state_liquid	Liquid	liquid
state_gas	Gas	gas
state_supercrit	Supercritical Fluid	supercritical fluid
state_plasma	Plasma	plasma
state_other	Other State	other state

10.5.8 **ms_test_direction_t:**—*scan direction*. See Table 14.

10.5.9 **ms_test_law_t:**—*scan law*. See Table 15.

10.6 *Raw Data Information Classes:*

10.6.1 *Raw Data Global Information Class*—**MS_Raw_Data_Global**. The **MS_Raw_Data_Global** structure occurs once per interchange file. The only required field is **nscans**, the number of spectral scans or library spectra recorded in the set. See Table 16.

10.6.1.1 **ms_data_mass_t:**—*mass axis units*. See Table 17.

10.6.1.2 **ms_data_time_t:**—*time axis units*. See Table 18.

10.6.1.3 **ms_data_intensity_t:**—*intensity axis units*. See Table 19.

10.6.1.4 **ms_data_format_t:**—*data format*. See Table 20.

10.7 *Raw Data Per-Scan Information Class*—**MS_Raw_Per_Scan**. A copy of this structure is completed once for each scan in the interchange file. When reading or writing, the **scan_no** field is used to indicate the index number of the scan (beginning at zero) to be read or written, respectively. Scans can be read or written in ascending or descending order; however, an error will occur if a scan number outside the range of one to (**nscans** (see **MS_Raw_Data_Global**, above) is specified.

10.7.1 There are no enumerated types in the **MS_Raw_Per_Scan** data structure. See Table 21.

10.8 *Library Data Per-Scan Information Class*—**MS_Raw_Library** —This structure occurs once per spectrum, but only for experiment type **expt_library**. For other experiment types, this structure is not used. An error will result when trying to read or write library information for experiment types other than **expt_library**. As in **MS_Raw_Per_Scan**, the **scan_no** variable must be set to the desired scan index before both reading and writing. An out-of-range index results in an error. There are no enumerated types. See Table 22.

TABLE 6 MS_Test_Data

Type	Field name	E	M	Specification Attribute
ms_test_separation_t	separation_type	x		separation experiment type
ms_test_inlet_t	ms_inlet	x		mass spectrometer inlet
Float	ms_inlet_temperature			mass spectrometer inlet temperature
ms_test_ioniz_t	ionization_mode	x		ionization mode
ms_test_polarity_t	ionization_polarity	x		ionization polarity
Float	electron_energy			electron energy
Float	laser_wavelength			laser wavelength
char[A]	reagent_gas			reagent gas
Float	reagent_gas_pressure			reagent gas pressure
char[A]	fab_type			FAB type
char[A]	fab_matrix			FAB matrix
Float	source_temperature			source temperature
Float	filament_current			filament current
Float	emission_current			emission current
Float	accelerating_potential			accelerating potential
ms_test_detector_t	detector_type	x		detector type
Float	detector_potential			detector potential
Float	detector_entrance_potential			detector entrance potential
ms_test_res_t	resolution_type	x		resolution type
char[A]	resolution_method			resolution method
ms_test_function_t	scan_function	x		scan function
ms_test_direction_t	scan_direction	x		scan direction
ms_test_law_t	scan_law	x		scan law
Float	scan_time			scan time
char[A]	mass_calibration_file			mass calibration file name
char[A]	external_reference_file			external reference file name
char[A]	internal_reference_file			internal reference file name
char[A]	comments			instrument parameter comments

[A] These fields are present in the data structure, but do not need to be filled by the application program when writing an interchange file. The API fills these fields with the appropriate values. However, on reading a file, the contents of these fields are filled with allocated strings, and must be freed by the caller.

10.8.1 *Size Restrictions on other Library String Fields (including terminal **NULL**):*

entry_id	32 bytes
source_data_file_reference	32 bytes
chemical_formula	64 bytes
wiswesser	128 bytes
smiles	255 bytes
other_structure	128 bytes
retention_reference_name	128 bytes
other_info	255 bytes

10.9 *Raw Data Per-Scan-Group Information Class*—**MS_Raw_Per_Group**. This structure is only used when the scan function is selected ion detection (function_sid), and occurs once per scan group. The group_no variable must be set to the desired group index before both reading and writing. An out-of-range index results in an error.

There are no enumerated types. See Table 23.

TABLE 7 ms_test_separation_t

Value	String Literal	Specification Description
separation_glc	Gas-Liquid Chromatography	gas-liquid chromatography
separation_gsc	Gas-Solid Chromatography	gas-solid chromatography
separation_nplc	Normal Phase Liquid Chromatography	normal phase liquid chromatography
separation_rplc	Reverse Phase Liquid Chromatography	reverse phase liquid chromatography
separation_ielc	Ion Exchange Liquid Chromatography	ion exchange liquid chromatography
separation_selc	Size Exclusion Liquid Chromatography	size exclusion liquid chromatography
separation_iplc	Ion Pair Liquid Chromatography	ion pair liquid chromatography
separation_olc	Other Liquid Chromatography	other liquid chromatography
separation_sfc	Supercritical Fluid Chromatography	supercritical fluid chromatography
separation_tlc	Thin Layer Chromatography	thin layer chromatography
separation_fff	Field Flow Fractionation	field flow fractionation
separation_cze	Capillary Zone Electrophoresis	capillary zone electrophoresis
separation_other	Other Chromatography	other chromatography
separation_none	No Chromatography	no chromatography

TABLE 8 ms_test_inlet_t

Value	String Literal	Specification Description
inlet_membrane	Membrane Separator	membrane separator
inlet_capillary	Capillary Direct	capillary direct
inlet_opensplit	Open Split	open split
inlet_jet	Jet Separator	jet separator
inlet_direct	Direct Inlet Probe	direct inlet probe
inlet_septum	Septum	septum
inlet_pb	Particle Beam	particle beam
inlet_reservoir	Reservoir	reservoir
inlet_belt	Moving Belt	moving belt
inlet_apci	Atmospheric Pressure Chemical Ionization Inlet	atmospheric pressure chemical ionization
inlet_fia	Flow Injection Analysis	flow injection analysis
inlet_es	Electrospray Inlet	electrospray inlet
inlet_infusion	Infusion	infusion
inlet_ts	Thermospray Inlet	thermospray inlet
inlet_probe	Other Probe	other probe inlet
inlet_other	Other Inlet	other inlet

TABLE 9 ms_test_ioniz_t

Value	String Literal	Specification Description
ionization_ei	Electron Impact	electron impact
ionization_ci	Chemical Ionization	chemical ionization
ionization_fab	Fast Atom Bombardment	fast atom bombardment
ionization_fd	Field Desorption	field desorption
ionization_fi	Field Ionization	field ionization
ionization_es	Electrospray Ionization	electrospray ionization
ionization_ts	Thermospray Ionization	thermospray ionization
ionization_apci	Atmospheric Pressure Chemical Ionization	atmospheric pressure chemical ionization
ionization_pd	Plasma Desorption	plasma desorption
ionization_ld	Laser Desorption	laser desorption
ionization_spark	Spark Ionization	sparkionization
ionization_thermal	Thermal Ionization	thermal ionization
ionization_other	Other Ionization	other ionization

TABLE 10 ms_test_polarity_t

Value	String Literal	Specification Description
polarity_plus	Positive Polarity	positive
polarity_minus	Negative Polarity	negative

TABLE 11 ms_test_detector_t

Value	String Literal	Specification Description
detector_em	Electron Multiplier	electron multiplier
detector_pm	Photomultiplier	photomultiplier
detector_focal	Focal Plane Array	focal plane array
detector_cup	Faraday Cup	Faraday cup
detector_dynode_em	Conversion Dynode Electron Multiplier	conversion dynode electron multiplier
detector_dynode_pm	Conversion Dynode Photomultiplier	conversion dynode photomultiplier
detector_multicoll	Multicollector	multi-collector
detector_other	Other Detector	other detector

TABLE 12 ms_test_res_t

Value	String Literal	Specification Description
resolution_constant	Constant Resolution	constant
resolution_proportional	Proportional Resolution	proportional

TABLE 13 ms_test_function_t

Value	String Literal	Specification Description
function_scan	Mass Scan	mass scan
function_sid	Selected Ion Detection	selected ion detection
function_other	Other Function	other function

TABLE 14 ms_test_direction_t

Value	String Literal	Specification Description
direction_up	Up	up
direction_down	Down	down
direction_other	Other Direction	other direction

TABLE 15 ms_test_law_t

Value	String Literal	Specification Description
law_linear	Linear	linear
law_exponential	Exponential	exponential
law_quadratic	Quadratic	quadratic
law_other	Other Law	other law

11. Application Programming Interface

11.1 There are a number of commonly used functions available in the application programming interface. These are grouped as follows:

11.1.1 Opening and closing interchange files,

11.1.2 Reading and writing global data,

11.1.3 Reading and writing per-component instrument data,

11.1.4 Reading and writing per-scan raw and library data,

11.1.5 Data structure initialization and clearing, and

11.1.6 Utility routines.

11.2 There are some additional API functions of lesser importance; these are not required for normal use of the protocol, but are described for completeness.

11.2.1 Note that, the header file “ms10.h” referenced below existed in earlier, preliminary, non-ASTM implementations also named “ms11.h” and subsequently “ms12.h”. It was finally named, perhaps confusingly, “ms10.h” to indicate a version “1.0” for the originally intended AIA standard.

11.3 *File Open and Close*—Interchange files are opened either for reading or writing. On most operating systems, opening a file for writing will destroy any existing file of the same name (on VMS, a new version is created). A file opened

TABLE 16 MS_Raw_Data_Global

Type	Field name	E	M	Specification Attribute
Long	nscans		x	number of scans
Long	starting_scan_no			starting scan number
Int	has_masses			(none)[A]
Int	has_times			(none)[A]
Double	mass_factor			mass axis scale factor[B]
Double	time_factor			time axis scale factor[B]
Double	intensity_factor			intensity axis scale factor[B]
Double	intensity_offset			intensity axis offset[C]
ms_data_mass_t	mass_units	x		mass axis units
ms_data_time_t	time_units	x		time axis units
ms_data_intensity_t	intensity_units	x		intensity axis units
ms_data_intensity_t	total_intensity_units	x		total intensity units
ms_data_format_t	mass_format	x		mass axis data format
ms_data_format_t	time_format	x		time axis data format
ms_data_format_t	intensity_format	x		intensity axis data format
char[A]	mass_label			mass axis label
char[A]	time_label			time axis label
char[A]	intensity_label			intensity axis label
Double	mass_axis_global_min			mass axis global range[D]
Double	mass_axis_global_max			mass axis global range[D]
Double	time_axis_global_min			time axis global range[D]
Double	time_axis_global_max			time axis global range[D]
Double	intensity_axis_global_min			intensity axis global range[D]
Double	intensity_axis_global_min			intensity axis global range[D]
Double	calibrated_mass_min			calibrated mass range[D]
Double	calibrated_mass_max			calibrated mass range[D]
Double	run_time			actual run time
Double	delay_time			actual delay time
Short	uniform_flag			uniform sampling flag
char[A]	Comments			raw data global comments

[A] These fields are used only when reading an interchange file, and indicate the presence of mass or time data, or both, in the interchange file. This allows applications to set up in advance to receive mass or time data or both.
[B] Scale factors default to 1.0. Scale factors are used as follows: When reading data arrays, the values returned in the arrays should each be multiplied by the respective scale factor to obtain the true values. When writing data arrays, the scale factor represents the divisor applied to the true values to obtain the values recorded in the interchange file. In either case, the numbers present in the mass, time, and intensity values arrays (see **MS_Raw_Per_Scan** data structure, below) represent the scaled, not the true values. The application is responsible for performing the appropriate scaling when reading or writing.
[C] The intensity axis offset defaults to 0.0. There are no offsets for the time or mass axes. When reading, the offset should be added to the recorded intensity values (**after scaling**) to obtain the true intensity values. When writing, the offset should be subtracted from the true values (**before scaling**).
[D] These fields are defined in the specification as ranges; for convenience of implementation, they are split into separate variables for minimum and maximum values.

TABLE 17 ms_data_mass_t

Value	String Literal	Specification Description
mass_m_z	M/Z	m/z
mass_arbitrary	Arbitrary Mass Units	arbitrary units
mass_other	Other Mass Units	other units

TABLE 18 ms_data_time_t

Value	String Literal	Specification Description
time_seconds	Seconds	seconds
time_arbitrary	Arbitrary Time Units	arbitrary units
time_other	Other Time Units	other units

TABLE 19 ms_data_intensity_t

Value	String Literal	Specification Description
intensity_counts	Total Counts	total counts
intensity_cps	Counts Per Second	counts per second
intensity_volts	Volts	volts
intensity_current	Current	current
intensity_arbitrary	Arbitrary Intensity Units	arbitrary units
intensity_other	Other Intensity	other units

TABLE 20 ms_data_format_t

Value	String Literal	Specification Description
data_short	Short	short (16-bit signed integer)[A]
data_long	Long	long (32-bit signed integer)[B]
data_float	Float	float (32-bit floating point)
data_double	Double	double (64-bit floating point)

[A] Default for mass and time data.
[B] Default for intensity data.

for reading must already exist. There are two file open API calls, one for read access and one for write access. These functions do much more than open the file; they read in or write out NetCDF dimension names and sizes and variable names and dimensionalities, respectively, and place the file in the appropriate mode for further operations.

11.3.1 **ms_open_read**—*open an interchange file for reading:*

Syntax:

#include "ms10.h"
int ms_open_read (char * *filename*)

Description:

The **ms_open_read** routine opens the interchange file named by *filename* and associates a file identifier with it. Any dimensions and variables defined in the file are read into internal API data structures. The file must exist, be readable, and be an interchange format file.

Return values:

If successful, the **ms_open_read** routine returns a non-negative **int** file identifier for use in subsequent API calls. On error, the routine returns the error code **MS_ERROR** (defined in ms10.h).

11.3.2 **ms_open_write**—*open an interchange file for writing:*

Syntax:

#include "ms10.h"
int ms_open_write (char * *filename* , **ms_admin_expt_t** *expt_type*, **long** *nscans*, **long** *ninst*, **ms_data_format_t** *mass_fmt*, **ms_data_format_t** *time_fmt*, **ms_data_format_t** *inty_fmt*, **int** *has_masses*, **int** *has_times*)

Description:

The **ms_open_write** routine creates and opens the interchange file specified by *filename* and associates a file identifier with it. The NetCDF dimension and variable definitions are written to the file, then the file is placed in data recording mode. The application must have the file system permissions necessary to create and write to the file.

The other arguments are:

TABLE 21 MS_Raw_Per_Scan

NOTE 1—Mass, time, and intensity arrays are declared as **void ***. In use, however, they are declared as arrays of type appropriate to the mass, time, and intensity data format (see **MS_Raw_Data_Global**, above), and simply cast to **void *** in the data structure. On writing, the API extracts the data and casts it back to the appropriate types before writing to the file. On reading, the API creates new arrays of the correct types, reads data into them, then casts to **void *** before returning to the application.

Type	Field name	E	M	Specification Attribute
Long	scan_no		x	scan number
Long	actual_scan_no		x	actual scan number
Long	points		x	number of points
void[A]	masses		x	mass axis values[A]
void[A]	times		x	time axis values[A]
void[A]	intensities		x	intensity axis values[B]
Long	flags		x	number of flags
long[A]	flag_peaks			flagged peaks[C]
short[A]	flag_values			flag values[C]
Double	total_intensity			total intensity
Double	a_d_rate			a/d sampling rate
Short	a_d_coadditions			a/d coaddition factor
Double	scan_acq_time			scan acquisition time
Double	scan_duration			scan duration
Double	mass_range[2]			mass scan range
Double	time_range[2]			time scan range
Double	inter_scan_time			inter-scan time
Double	resolution			resolution

[A] On reading, one of these pointers may be returned as **NULL** if the respective data is not found in the file. On writing, if one of these types is not present **for all scans in the file**, a **NULL** pointer may be passed in for the missing type. **Either mass values, time values, or both must be present**. All mass/intensity or time/intensity data occur as matched pairs; mass/time/intensity data occur as matched triplets. Therefore, if mass/time/intensity values are present for one scan, they must be present for all scans. Mass data is an array of **mass_format type**; time data is an array of **time_format** type. **All datum values, whether mass or time, are recorded in ascending order.**

On writing, if a scan has no data (**points = 0**), then **NULL** pointers may be passed for **masses, times, intensities, flag_peaks**, and **flag_values**. On reading, **NULL** pointers will be returned.

[B] Intensity axis values are an array of **intensity_format** type. There must be a one-to-one match between intensity datum points and the corresponding mass, time or mass/time point.

[C] On writing, these pointers may be passed as **NULL** if **flags = 0**. On reading, **NULL** pointers will be returned if there are no flagged peaks.

On either reading or writing, the datum values in the **flag_peaks** array correspond to the index of the peak in the mass or time values array (starting at zero). For example, a scan with ten masses, the first, fifth, and sixth of which are flagged, would have a **flag_peaks** array containing the values (0, 4, 5).

Flag_values datum points are each the logical OR of individual flag values, and apply to the corresponding datum point in the **flag_peaks** array.

expt_type	the enumerated set value which specifies the experiment type.
nscans	number of scans to be recorded
ninst	number of instrument components (if zero, instrument variables will not be defined in the file)
mass_fmt	the enumerated set value which specifies the mass values type
time_fmt	the enumerated set value which specifies the time values type
inty_fmt	the enumerated set value which specifies the intensity values type
has_masses	if non-zero, specifies that mass data will be recorded
has_times	if non-zero, specifies that time data will be recorded; one of these two arguments must be non-zero.

Return values:

If successful, the **ms_open_write** routine returns a non-negative **int** file identifier for use in subsequent API calls. On error, the routine returns the error code **MS_ERROR**.

TABLE 22 MS_Raw_Library

Type	Field name	E	M	Specification Attribute
Long	scan_no		x	(none)
char[A]	entry_name		x	entry name[A]
char[A]	entry_id			entry id
Long	entry_number			original entry number
char[A]	source_data_file_reference			source data file reference
char[A]	cas_name			CAS name[A]
char[A]	other_name_0			other names[A,B]
char[A]	other_name_1			other names[A,B]
char[A]	other_name_2			other names[A,B]
char[A]	other_name_3			other names[A,B]
Long	cas_number			CAS number
char[A]	formula			chemical formula
char[A]	wiswesser			Wiswesser notation
char[A]	smiles			SMILES notation
char[A]	molfile_reference			MOL file reference name
char[A]	other_structure			other structure notation
Double	retention_index			retention index
char[A]	retention_type			retention index type
Double	absolute_retention			absolute retention time
Double	relative_retention			relative retention
char[A]	retention_reference			retention reference name
Long	retention_cas			retention reference CAS number
Float	mp			melting point
Float	bp			boiling point
Double	chemical_mass			chemical mass
Long	nominal_mass			nominal mass
Double	accurate_mass			accurate mass
char[A]	other_info			other information

[A] There are a maximum of six names (entry, CAS, and four other) permitted per entry. **A limit of 255 characters per name is imposed.**

[B] These fields are defined as a string array in the specification document; they are implemented are separate string variables here for convenience.

TABLE 23 MS_Raw_Per_Group

Type	Field name	E	M	Specification Attribute
Long	group_no		x	(none)
Long	mass_count		x	number of masses in group
Long	starting_scan		x	starting scan number
double[A]	masses		x	group masses[A]
double[A]	sampling_times			sampling times[A]
double[A]	delay_times			delay times[A]

[A] These are parallel arrays; that is, for every mass, there is a corresponding sampling time and delay time entry. The delay time for the last mass in the group may be set to zero (since there is no next mass).

Important —The API assumes that these three arrays have constant dimensionality equal to the **maximum number** of masses in any group. (See **ms_write_group_global ()** and **ms_read_group_global ()**, below). On input or output, the API fills unused values with the floating point default. On file write, these arrays are assumed to be defined (and owned) by the caller. On file read, the arrays will be dynamically allocated by the API. It is the caller's responsibility to free this storage after use.

11.3.3 **ms_close**—*close an open file:*

Syntax:

#include "ms10.h"
void ms_close (int *file_id*)

Description:

The **ms_close** routine closes the previously opened interchange file associated with the file identifier *file_id* and disassociates the file identifier. No additional API calls may be made using the file identifier after this call completes.

Return values:

No values are returned. Any errors are ignored.

11.4 *Reading and Writing Global Data*—Global data occurs once per interchange file. Data structures which contain global data are: **MS_Admin_Data, MS_Sample_Data, MS_Test_Data, and MS_Raw_Data_Global**. In the implementation, most of the fields in these data structures are written as NetCDF dimensions or global attributes.

11.4.1 **ms_read_global**—*read global information:*

Syntax:

#include "ms10.h"

int ms_read_global (int*file_id* , **MS_Admin_Data *** *admin_data* , **MS_Sample_Data*** *sample_data* , **MS_Test_Data *** *test_data*, **MS_Raw_Data_Global *** *raw_data*)

Description:

Reads global information from the interchange file associated with *file_id*. The file must have been opened using **ms_open_read**. Pointers to the data structures must be non-**NULL** and reference valid structures. On return, the data structure fields will be filled with values read from the interchange file. See the discussion of data structure initialization and clearing, below.

Return values:

If the call is successful, the code **MS_NO_ERROR** is returned. On an error, the code **MS_ERROR** is returned. An error will occur if the file identifier is invalid, any data structure pointer is **NULL**, memory allocation fails to allocate storage for input data, or on an internal NetCDF error.

11.4.2 **ms_write_global**—*write global information:*

Syntax:

#include "ms10.h"

int ms_write_global (int *file_id*, **MS_Admin_Data *** *admin_data*, **MS_Sample_Data *** *sample_data*, **MS_Test_Data *** *test_data*, **MS_Raw_Data_Global *** *raw_data*)

Description:

Writes global information to the interchange file associated with *file_id*. The file must have been opened using **ms_open_write**. Pointers to the data structures must be non- **NULL** and reference valid structures. Values are extracted from the data structure fields and are written to the interchange file. It is important to initialize the data structures (using **ms_init_global ()**) before filling them. Any data structure element which has a NULL value will not be written. See the discussion of data structure initialization and clearing, below.

Return values:

If the call is successful, the code MS_NO_ERROR is returned. On an error, the code MS_ERROR is returned. An error will occur if the file identifier is invalid, any data structure pointer is NULL, or on an internal NetCDF error.

11.4.3 **ms_read_group_global**—*read group global information*

Syntax:

#include "ms10.h"

int ms_read_group_global (int *file_id*, **long *** *number_of_groups*, **long *** *maximum_number_of_masses_in_group*)

Description:

The **ms_read_group_global** function retrieves global scan group information from the interchange file associated with *file_id*. The file must have been opened using **ms_open_read**. Scan group information is stored only for experiments for which the scan function is **function_sid**. If the scan function for the file is not **function_sid** or no scan group data has been recorded in the file, zeros will be stored in the locations pointed to by *number_of_groups* and *maximum_number_of_masses_in_group*.

Return values:

If the call is successful, the code MS_NO_ERROR is returned. On an error, the code MS_NO_ERROR is returned. An error will occur if the file identifier is invalid, any data structure pointer is **NULL**, or on an internal NetCDF error.

11.4.4 **ms_write_group_global**— *write group global information.*

Syntax:

#include "ms10.h"

int ms_write_group_global (int *file_id*, **long** *number_of_groups*, **long** *maximum_number_of_masses_in_group*)

Description:

The **ms_write_group_global** function defines global scan group information to the interchange file associated with *file_id*. The file must have been opened using **ms_open_write**. Scan group information is stored **only** for experiments for which the scan function is **function_sid**. Scan group data is stored as a block of dimension *number_of_groups* by *maximum_number_of_masses_in_group*. Extra values in any group which contains fewer than the *maximum_number_of_masses_in_group* will be set to the default floating point value. There is no implementation-defined upper limit for *number_of_groups* or *maximum_number_of_masses_in_group*, but when writing an interchange file, these should be set to the actual values encountered in the file.

Return values:

If the call is successful, the code **MS_NO_ERROR** is returned. On an error, the code **MS_ERROR** is returned and zeros are stored in the pointer locations. An error will occur if the file identifier is invalid, if the interchange file is not open for writing, or on an internal NetCDF error.

11.5 *Reading and Writing Instrument Information*—Instrument information is stored in the interchange file as arrays of fields, one array element per instrument component. This information is read and written on a per-component basis. To implement indexing into these arrays, each component is assigned an index number, from zero to (number_instrument

_components-1) as specified in the MS_Admin_Data structure (and in the ninst argument to ms_open_write). The assignment of index numbers to components is arbitrary, and since instrument component information is simply character strings, there is no programmatic way of determining what each component is (for example, a gas chromatograph or a mass spectrometer). With future refinement of the data interchange specification for GLP/GALP, some means of identifying component types may be developed.

ms_read_instrument or ms_write_instrument can only be called **after** calls to ms_open_read and ms_read_global or ms_open_write and ms_write_global, respectively.

11.5.1 **ms_read_instrument**—*read instrument information.*

Syntax:

#include "ms10.h"

int ms_read_instrument (int *file_id*, **MS_Instrument _Data *** *component_data*)

Description:

The ms_read_instrument routine reads information for a single instrument component from the interchange file referenced by *file_id*. The file must have been opened using ms_open_read. The pointer to the *component_data* structure must be valid, and the inst_no field must be set to the index number of a valid instrument component. Upon a successful return, the other fields in the data structure will be filled with component data read from the file. This routine should be called once for each component; the data structure should be cleared after each call to ensure that memory allocated during the previous call is freed.

Return values:

On a successful return, the data structure contents are filled out and the code MS_NO_ERROR is returned. On error, the code MS_ERROR is returned. Errors will occur if the *file_id* is not valid, the data structure pointer is NULL, the inst_no value is out of range, or on an internal NetCDF error. On an error return, data structure contents are not valid.

11.5.2 **ms_write_instrument**—*write instrument information.*

Syntax:

#include "ms10.h"

int ms_write_instrument (int *file_id*, **MS_Instrument _Data *** *component data*)

Description:

The **ms_write_instrument** routine writes information for a single instrument component to the interchange file referenced by *file_id*. The file must have been opened using **ms_open_read**. The pointer to the *component_data* structure must be valid, and the **inst_no** field must be set to the index number of a valid instrument component. Upon a successful return, the other fields in the data structure must be filled with component data read to be written to the file. This routine should be called once for each component; the data structure should be initialized before each call.

Return values:

On a successful return, the code **MS_NO_ERROR** is returned. On error, the code **MS_ERROR** is returned. Errors will occur if the file_id is not valid, the data structure pointer is **NULL**, the **inst_no** value is out of range, or on an internal NetCDF error.

11.6 *Reading and Writing Scan Information*—Scan information, both raw and library, is stored in the interchange file as arrays of fields. In each array, one field is designated for each scan. Information is read and written on a per-scan basis. Each scan has a scan number, an index in the range zero to **(nscans-1)** from the data structure **MS_Raw_Data_Global** (and the argument to **ms_open_write**).

MS_Raw_Per_Scan data must be supplied for each scan, for all experiment types. For experiment type **expt_library** only, **MS_Raw_Library** data must also be supplied for each scan. Both data structures should be properly initialized (and cleared, if appropriate). As described above, mass, time, and intensity arrays are cast to **void*** pointers in the data structures, but are actually of formal types as specified by the mass, time, and intensity data format fields. Application programs are responsible for ensuring that formal types and actual types are consistent for these arrays, otherwise their contents will be written or read incorrectly (and will end up as garbage in the file or in the data structure).

For both reading and writing, the **scan_no** field in both data structures must be set to the desired scan index. There is no concept of "the next scan", so the scan index must be explicitly specified on both read and write.

ms_read_per_scan or **ms_write_per_scan** can only be called **after** calls to **ms_open_read** and **ms_read_global** or **ms_open_write** and **ms_write_global**, respectively.

11.6.1 **ms_read_per_scan**—*read per-scan information.*

Syntax:

#include "ms10.h"

int ms_read_per_scan (int *file_id*, **MS_Raw_Per_Scan *** *scan_data*, **MS_Raw_Library *** *library_data*)

Description:

The **ms_read_per_scan** routine reads information for a single scan from the interchange file referenced by file_id. The file must have been opened using **ms_open_read**. The pointer to the scan_data structure must be valid, and the **scan_no field** must be set to the index number of a valid scan. If the experiment type is **expt_library**, the *library_data* structure pointer must also be valid, and its **scan_no** field set to the same scan number. **For other experiment types, the *library_data* pointer MUST be NULL**. Upon a successful return, the other fields in the data structure(s) will be filled with raw (and library, if appropriate) data read from the file. This routine should be called once for each scan; the data structure(s) should be cleared after each call to ensure that memory

allocated during the previous call is freed.

Return values:

On a successful return, the data structure contents are filled out and the code **MS_NO_ERROR** is returned. On error, the code **MS_ERROR** is returned. Errors will occur if the *file_id* is not valid, the **MS_Raw_Per_Scan** data structure pointer is **NULL**, the **scan_no** value is out of range, or on an internal NetCDF error. Errors will also occur if a non-**NULL** pointer is supplied for non-library files. On an error return, data structure contents are not valid.

11.6.2 **ms_write_per_scan**—*write per-scan information.*

Syntax:

#include "ms10.h"

int ms_write_per_scan (int *file_id*, **MS_Raw_Per_Scan *** *scan_data*, **MS_Raw_Library *** *library_data*)

Description:

The **ms_write_per_scan** routine writes information for a single scan to the interchange file referenced by *file_id*. The file must have been opened using **ms_open_write**. The pointer to the *scan_data* structure must be valid, and the **scan_no** field must be set to the index number of a valid scan. If the experiment type is **expt_library**, the *library_data* structure pointer must also be valid, and its **scan_no** field set to the same scan number. **For other experiment types, the** *library_data* **pointer MUST be NULL** . This routine should be called once for each scan; the data structure(s) should be initialized (and cleared, if appropriate) before each call. See the discussion of the **MS_Raw_Per_Scan** and **MS_Raw_Library** data structures above for more details.

Return values:

On a successful completion, the code **MS_NO_ERROR** is returned. On error, the code **MS_ERROR** is returned. Errors will occur if the *file_id* is not valid, the **MS_Raw_Per_Scan** data structure pointer is **NULL**, the **scan_no** value is out of range, or on an internal NetCDF error. Errors will also occur if a non-**NULL** pointer is supplied for non-library files.

11.7 *Reading and Writing Scan Group Information*—Scan group information is stored for experiments in which that scan function is **function_sid**. Scan groups have no meaning for other scan function types. Once scan group global information has been written to or retrieved from an interchange file (see **ms_write_group_global** and **ms_read_group_global**, above), data for each scan group is then written or read. The **MS_Raw_Per_Group** data structure is filled for writing or reading each scan group.

Scan group mass, sampling time, and delay time information is stored as a rectangular block, dimensioned *number of scan groups by maximum number of masses in any group* (see above). In order to ensure that unused elements in the arrays can be properly filled with default values, all arrays in **MS_Raw_Per_Group** are of length maximum number of masses in group. On writing, the API creates and fills the arrays, and it is the caller's responsibility to free them after use. Failure to do so will cause a memory leak on successive calls to **ms_read_per_group**.

ms_read_per_group can only be called **after** calls to **ms_open_read** and **ms_read_group_global. ms_write_per_group** can only be called after calls to **ms_open_write** and **ms_write_group_global**. On both read and write calls, the **group_no** field must be filled with the desired group number prior to calling. Errors will result otherwise.

11.7.1 **ms_read_per_group**—*read per-scan group information.*

Syntax:

#include "ms10.h"

int ms_read_per_group (int *file_id*, **MS_Raw_Per_Group *** *group_data*)

Description:

The **ms_read_per_group** function reads information for a single scan group from the interchange file referenced by *file_id*. The file must have been opened using **ms_open_read**. The pointer to the group_data structure must be valid, and the **group_no** field must be set to the index number of a valid group. Upon a successful return, the fields in the data structure will be filled in. The **masses, sampling_times**, and **delay_times** arrays will be allocated by the API and filled with values read from the file. Any unused array elements will be set to the **MS_NULL_FLT** default. The caller is responsible for freeing the arrays after use. An initialization function, **ms_init_group**, is provided for this purpose (see below).

Return values:

On a successful return, the value **MS_NO_ERROR** will be returned, and the data structure completed. On error, the value **MS_ERROR** will be returned and the data structure contents will be undefined. Errors will occur if the file identifier is invalid, the file is not open for reading, the group index is out of range, any pointer is NULL, or on an internal error.

11.7.2 **ms_write_per_group**—*write per-scan information.*

Syntax:

#include "ms10.h"

int ms_write_per_group (int *file_id*, **MS_Raw_Per_group *** *group_data*)

Description:

The **ms_write_per_group** function writes information for a single scan to the interchange file referenced by *file_id*. The file must have been opened using **ms_open_write**. The pointer to the *group_data* structure must be valid, and the **group_no** field must be set to the index number of a valid group. The **mass_count** field must be set to the actual number of masses in the group (which must be less than or equal to the maximum declared in the **ms_write_group_global** call). The **masses, sampling_times**, and **delay_times** arrays must be set by the caller, and must be of a length equal to the maximum number of masses. Any unused array

elements will be set to the **MS_NULL_FLT** default.

Return values:

On success, the value **MS_NO_ERROR** will be returned, while on error, the value **MS_ERROR** will be returned. Errors will occur if the file identifier is invalid, the file is not open for writing, the group index is out of range, and pointer is **NULL**, or on an internal error.

11.8 *Data Structure Initialization and Clearing*—Three routines are provided for initializing and clearing data structure contents. On writing, the API routines copy the contents of data structures into the file; any data structure fields which have NULL values are generally ignored. On reading, API routines copy values only into data structure fields for which data is present in the interchange file. Other fields are left with NULL values.

11.8.1 The initialization and clearing routines have the following conventions for initializing data structure fields:

(1) All **float** and **double** fields are initialized to the constant **MS_NULL_FLT**, defined in the include file ms10.h.

(2) All **short** and **long** fields are initialized to the constant **MS_NULL_INT**.

(3) All **char *** and **void *** pointers are initialized to **NULL**.

11.8.1.1 On *clearing* operations, the memory referenced by all non-**NULL** pointers is freed before initialization. On *initialization*, the pointers are simply set to **NULL**, regardless of content. *Initializing* data structures which contain pointers to allocated memory could cause a memory leak if the pointer references are not remembered elsewhere by the application. Data fields containing pointers to statically allocated memory should not be *cleared*, or memory access errors will occur.

11.8.1.2 In all routines below, the first parameter is a Boolean flag which indicates whether a *clear* or an *initialization* should be performed. If non-zero, the fields are *cleared*, otherwise they are *initialized*. The remaining parameters are pointers to data structures; if a **NULL** is passed in place of any pointer, the data structure reference represented by that argument is ignored without error.

11.8.2 **ms_init_global**—*initialize global data structures*.

Syntax:

#include “ms10.h”

void ms_init_global (int *clear_flag*, **MS_Admin_Data *** *admin_data*, **MS_Sample_Data *** *sample_data*, **MS_Test_Data *** *test_data*, **MS_Raw_Data_Global *** *raw_data*)

Description:

The **ms_init_global** routine initializes (and optionally clears, if *clear_flag* is non-zero) the contents of global data structures. Any pointer passed as **NULL** is ignored. Data structures should be *initialized* before use, and cleared after use. All **ms_read_...** routines allocate memory for strings and numeric arrays; failure to *clear* data structures containing such fields after a read will result in a memory leak unless the pointers have been copied to other locations. New memory is allocated for each read operation; old pointers are not reused, and their contents will become lost if not properly cleared.

Return values:

The function has no return value; any internal errors are ignored.

11.8.3 **ms_init_instrument**—*initialize instrument data structure*.

Syntax:

#include “ms10.h”

void ms_init_instrument (int *clear_flag*, **MS_Instrument_Data *** *component_data*)

Description:

The **ms_init_instrument** routine initializes (and optionally clears, if *clear_flag* is non-zero) the contents of the instrument component data structure. Any pointer passed as **NULL** is ignored. Data structures should be initialized before use, and *cleared* after use. All **ms_read_...** routines allocate memory for strings and numeric arrays; failure to *clear* data structures containing such fields after a read will result in a memory leak unless the pointers have been copied to other locations. New memory is allocated for each read operation; old pointers are *not reused*, and their contents will become lost if not properly cleared.

Return values:

The function has no return value; any internal errors are ignored.

11.8.4 **ms_init_per_scan**—*initialize per-scan data structures*

Syntax:

#include “ms10.h”

void ms_init_per_scan (int *clear_flag*, **MS_Raw_Per_Scan *** *raw_data*, **MS_Raw_Library *** *library_data*)

Description:

The **ms_init_per_scan** routine initializes (and optionally clears, if *clear_flag* is non-zero) the contents of the per scan raw and library data structures. Any pointer passed as NULL is ignored. Data structures should be *initialized* before use, and *cleared* after use. All **ms_read_...** routines allocate memory for strings and numeric arrays; failure to *clear* data structures containing such fields after a read will result in a memory leak unless the pointers have been copied to other locations. New memory is allocated for each read operation; old pointers are *not reused*, and their contents will become lost if not properly cleared.

Return values:

The function has no return value; any internal errors are ignored.

11.8.5 **ms_init_per_group**—*initialize per-scan group data structure*.

Syntax:

#include “ms10.h”

void ms_init_per_group (int *clear_flag*, **ms_Raw_Per_Group * group_data**

Description:

The **ms_init_per_group** function initializes (and optionally clears, if *clear_flag* is non-zero) the contents of the per-scan group data structure. Any pointer passed as **NULL** is ignored. Data structures should be *initialized* before use and *cleared* after use. All **ms_read_...** routines allocate memory for strings and arrays; failure to *clear* data structures containing such fields after a read will result in a memory leak unless the pointers have been cached for later release by the application. New memory is allocated for each read operation; old pointers are *not reused*, and their contents will become lost if not properly cleared.

Return values:

This function has no return value; any internal errors are ignored.

11.9 *Utility Functions:*

11.9.1 **ms_read_TIC**—*read total ion chromatogram.*

Syntax:

#include "ms10.h"

int ms_read_TIC (int *file_id*, **long *** *number_of_points_in_TIC*, **double * *** *TIC_intensities*, **double * *** *TIC_times*)

Description:

The **ms_read_TIC** routine is a convenience function to permit reading the total ion chromatogram in a single call, rather than retrieving the TIC point-by-point through successive calls to the **ms_read_per_scan** routine. Before this function can be called, the routines **ms_open_read** and **ms_read_global** must be called (in that order). Arguments to this function are a pointer to a variable which will be loaded with the number of points in the TIC array and pointers which will be loaded with the addresses of arrays containing the TIC intensity and the time values. These arrays are allocated during the call to **ms_read_TIC** and must be freed by the caller after use.

Return values:

On success, the value **MS_NO_ERROR** is returned, and the two pointers will be set as described. If the call fails, the error code **MS_ERROR** will be returned, *number_of_points_in_TIC* will be set to zero, and *TIC_intensities* and *TIC_times* will be set to **NULL**. Failure will occur if the file id is invalid, any pointer is **NULL**, or an internal error occurs.

11.9.2 **ms_convert_date**—*ISO 8601 date conversions.*

Syntax:

#include "ms10.h"

```
typedef struct {
        char *          string;
        int             year;
        int             month;
        int             day;
        int             hour;
        int             minute;
        int             second;
        int             differential;
}       MS_Date_Time;
```

int ms_convert_date (int *to_string_flag*, **MS_Date_Time *** *date_info*)

Description:

The **ms_convert_date** routine converts date and time information to or from ISO 8601 time stamp string format and integer values. If the *to_string_flag* is non-zero, the **string** field is loaded with a time stamp in ISO 8601 format using the remaining integer fields. This string is allocated from memory, and must be freed after use. If the flag is zero, then the **string field** (which must be in ISO 8601 format) is parsed into the appropriate integer data fields. **Year** is an absolute number (for example, 1992), referenced to 0 AD. **Hour** is on a 24-h system, with values from 0 to 23. **Minute** ranges from 0 to 59. Second ranges from 0 to 61, allowing for two "leap seconds." **Differential** is the signed offset in hours and minutes from Greenwich Mean Time. A complete description of the ISO 8601 time stamp format can be found in Specification E2077.

Time data should **always** be presented as Coordinated Universal Time (UTC or Greenwich Mean Time, as disseminated by time signals); this is local time plus the differential, which therefore, shows the correction to local time. If only local times were (incorrectly) used, then interchange files recorded in different time zones would be improperly time stamped on the receiving end. The POSIX standard specifies two time functions: **gmtime()** and **localtime()**, which return GMT and local time, respectively. The **gmtime()** function should be used if possible.

Return values:

On success, the routine returns **MS_NO_ERROR**. On error, the code **MS_ERROR** is returned. Errors will occur if the data structure pointer is **NULL**, and on conversion from a string, if the **string** pointer is **NULL** or the string has length shorter than that of a proper ISO 8601 string.

11.9.3 **ms_string_to_enum**—*string literal to enum conversion.*

Syntax:

#include "ms10.h"

int ms_string_to_enum (char * *string*)

Description:

The **ms_string_to_enum** routine performs a lookup of the string value and returns the corresponding enumerated type value. The lookup is case-insensitive; white space is significant, and must match, however.

Return values:

On a successful lookup, the routine returns the enumerated type value (cast to **int**). On error, the routine returns the code **MS_ERROR**. Errors will occur if the *string* pointer is **NULL** or the string can not be found in the lookup table.

11.9.4 **ms_enum_to_string**—*enum to string literal conversion*.

Syntax:

#include "ms10.h"
char * ms_enum_to_string (int *enum_value*)

Description:

The **ms_enum_to_string** routine performs a lookup of the *enum_value* value (cast to **int**) and returns the corresponding string literal value. The returned string **MUST** be treated as read-only, since the pointer references a static table in memory.

Return values:

On a successful lookup, the routine returns the pointer to the character string corresponding to the enumerated type value. On error, the routine returns a **NULL** pointer. An error will occur if the *enum_value* is out of range.

11.9.5 **ms_copy_array**—*array data type conversion*.

Syntax:

#include "ms10.h"
void ms_copy_array (void * *input_array*, **ms_data_format_t** *input_type*, **long** *input_count*, **void *** *output_array*, **ms_data_format_t** *output_type*, **int** *round_flag*)

Description:

The **ms_copy_array** copies a numeric array of one data type into an array of another data type, with optional rounding of floating point values to integers. This routine is useful when data read from an interchange file is in a different format than that desired for the target data file or vice versa. Both input and output arrays must exist, and the output array must be of a size sufficient to hold *input_count* values. If the input is a floating point type (that is, **data_float** or **data_double**), the output is an integer type (that is, **data_short** or **data_long**), then if the *round_flag* is non-zero, rounding will occur prior to truncation to the integer. No checking is made for overflow, underflow, or loss of significance.

Return values:

There is no return value.

11.10 *Miscellaneous Internal Routines*—The following routines are used internally by other API functions and are not generally used by application developers. Under special circumstances, such as reading or writing vendor-specific information in interchange files, these routines may be useful.

11.10.1 **ms_associate_id**—*allocate protocol internal data*.

Syntax:

#include "ms10.h"
int ms_associate_id (int *file_id*)

Description:

The protocol API uses a number of internal variables which reference NetCDF variables, dimensions, and other parameters. When an interchange file is opened for reading or writing, a unique set of these internal variables is associated with the NetCDF file id. This function creates the internal variable set and the association with file id. It is the first step after the **ncopen** or **nccreate** function call.

Return values:

On a successful return, the code **MS_NO_ERROR** is returned. On error, **MS_ERROR** is returned. Errors will occur if the internal variable set cannot be allocated or an internal NetCDF error occurs.

11.10.2 **ms_dissociate_id**—*free AIA MS internal data definitions*.

Syntax:

#include "ms10.h"
int ms_dissociate_id (int *file_id*)

Description:

The **ms_dissociate_id** routine removes the association between the NetCDF file id and internal data structures, and frees the space allocated for those data structures. This routine is used in **ms_close** as the last step in closing an interchange file.

Return values:

On a successful return, the code **MS_NO_ERROR** is returned. On error, **MS_ERROR** is returned. Errors will occur if the *file_id* does not correspond to a valid interchange file

11.10.3 **ms_write_dimensions**—*write dimension definitions*.

Syntax:

#include "ms10.h"
int ms_write_dimensions (int *file_id*, **long** *nscans*, **long** *ninst*)

Description:

The **ms_write_dimensions** routine is used by **ms_open_write** to define the standard dimensions to the interchange file. It should never be used by applications unless the standard **ms_open_write** routine is replaced with a custom version. *Nscans* is the number of scans to be written; *ninst* is the number of instrument components (may be zero).

The protocol dimension names and sizes are defined in internal data structures which are inaccessible to application programs.

Return values:

On a successful return, the code **MS_NO_ERROR** is returned. On error, **MS_ERROR** is returned. Errors will occur if the *file_id* does not correspond to a valid interchange file, if the interchange file is not opened for writing, if the file is not in definition mode, or on an internal NetCDF error.

11.10.4 **ms_read_dimensions**—*read protocol dimension definitions*.

Syntax:

#include “ms10.h”
int ms_read_dimensions (int *file_id*)

Description:

The **ms_read_dimensions** routine is used by **ms_open_read** to extract the standard dimensions from the interchange file. It should never be used by applications unless the standard **ms_open_read** routine is replaced with a custom version. The protocol dimensions which have been defined in the file are read into internal data structures inaccessible to application programs.

Return values:

On a successful return, the code **MS_NO_ERROR** is returned. On error, **MS_ERROR** is returned. Errors will occur if the *file_id* does not correspond to a valid interchange file, if the interchange file is not opened for reading, or on an internal NetCDF error.

11.10.5 **ms_write_variables**—*write protocol variable definitions.*

Syntax:

#include “ms10.h”
int ms_write_variables (int *file_id*, **ms_admin_expt_t** *expt_type*, **ms_data_format_t** *mass_fmt*, **ms_data_format_t** *time_fmt*, **ms_data_format_t** *intensity_fmt*, **int** *has_masses*, **int** *has_times*)

Description:

The **ms_write**_variables routine is used by ms_open_write to define the standard variables to the interchange file. It should **never** be used by applications unless the standard **ms_open_write** routine is replaced with a custom version.

The arguments are the same as those used by **ms_open_write**. The standard variable names and dimensionalities are defined in internal data structures which are inaccessible to application programs.

Return values:

On a successful return, the code **MS_NO_ERROR** is returned. On error, **MS_ERROR** is returned. Errors will occur if the *file_id* does not correspond to a valid interchange file, if the interchange file is not opened for writing, if the file is not in definition mode, or on an internal NetCDF error.

11.10.6 **ms_read_variables**—*read protocol variable definitions*.

Syntax:

#include “ms10.h”
int ms_read_variables (int *file_id*)

Description:

The **ms_read_variables** routine is used by **ms_open_read** to extract the standard variable definitions from the interchange file. It should **never** be used by applications unless the standard **ms_open_read** routine is replaced with a custom version. The standard variables which have been defined in the file are read into internal data structures inaccessible to application programs.

Return values:

On a successful return, the code **MS_NO_ERROR** is returned. On error, **MS_ERROR** is returned. Errors will occur if the *file_id* does not correspond to a valid interchange file, if the interchange file is not opened for reading, or on an internal NetCDF error.

11.10.7 **ms_read_enum_attribute**— *read on enumerated type value*.

Syntax:

#include “netcdf.h”
#include “ms10.h”
int ms_read_enum_attribute (int *file_id*, **int** *variable_id*, **char *** *attribute_name*)

Description:

The **ms_read_enum_attribute** routine is used by **ms_read_global** to extract enumerated type values from the interchange file. Enumerated types are stored in the interchange file as string literals in compliance with the convention that NetCDF files be self-describing. On reading, the strings are converted to their corresponding enumerated value and returned as such. Applications which store custom enumerated types can use this routine (and a modified string to type lookup table) to extract their enumerated strings from the file.

The *variable_id* is the identifier assigned by NetCDF to the variable the attribute describes. Attributes defined on a global (file) basis use the constant **NC_GLOBAL**.

There is no corresponding function for *writing* enumerated attribute values; it is easy to write one using the functions **ms_enum_to_string and ncattput** (from the NetCDF toolkit).

Return values:

On a successful return, the enumerated type value (cast to **int**) is returned. On error, **MS_ERROR** is returned. Errors will occur if the *file_id* does not correspond to a valid interchange file, if the interchange file is not opened for reading, if the attribute does not exist or have an **NC_CHAR type**, a conversion for the string cannot be found, or on an internal NetCDF error.

11.10.8 **ms_write_string_variable**—*write a string variable value*.

Syntax:

#include “ms10.h”
int ms_read_string_variable (int *file_id*, **int** *variable_id*, **long** *index_1*, **long** *index_2*, **int** *max_length*, **char *** *string*)

Description:

The **ms_write_string_variable** routine facilitates writing an arbitrary string variable value to the interchange file associated with *file_id*. The *variable_id* is the identifier used by NetCDF to reference the variable. This function can write any type of string variable value: a simple string, one string

from an array of strings, and one string from a two dimensional array of strings (for example, the **other_name** variable in **MS_Raw_Library** occurs as an array of four strings in an array of scans).

The *index_1* and *index_2* arguments are used for indexing into one- and two-dimensional string arrays, respectively. For simple strings, both arguments MUST be set to –1; for a one-dimensional array, *index_1* should be set to the appropriate index value, and *index_2* set to– 1; and for two-dimensional arrays, both indices should be set to proper values.

The *max_length* argument specifies the maximum allowed length for the string value. Strings longer than this length will be truncated. The *string* argument references the string to be written.

Return values:

On a successful return, the code **MS_NO_ERROR** is returned. On error, the code **MS_ERROR** is returned. Errors will occur if the *file_id* does not correspond to a valid interchange file, if the interchange file is not opened for writing and is in data mode, if the variable does not exist or have an **NC_CHAR** type, if the indices are not valid, or on an internal NetCDF error.

11.10.9 **ms_read_string_variable**—*read a string variable value*.

Syntax:

#include “ms10.h”

char* ms_read_string_variable (int *file_id*, **int** *variable_id*, **long** *index_1*, **long** *index_2*, **int** *max_length*)

Description:

The **ms_read_string_variable** routine facilitates reading an arbitrary string variable value from the interchange file associated with *file_id*. The *variable_id* is the identifier used by NetCDF to reference the variable. This function can read any type of string variable value: a simple string, one string from an array of strings, and one string from a two dimensional array of strings (for example, the **other_name** variable in **MS_Raw_Library** occurs as an array of four strings in an array of scans).

The *index_1* and *index_2* arguments are used for indexing into one- and two-dimensional string arrays, respectively. For simple strings, both arguments MUST be set to –1; for a one-dimensional array, *index_1* should be set to the appropriate index value, and *index_2* set to– 1; and for two-dimensional arrays, both indices should be set to proper values.

The *max_length* argument specifies the maximum allowed length for the string value. Strings longer than this length will be truncated.

The pointer returned by this function references allocated memory; the memory must be freed by the application after use or a memory leak may occur.

Return values:

On a successful return, a pointer to an allocated and filled character string will be returned. On error, a **NULL** pointer will be returned. Errors will occur if the *file_id* does not correspond to a valid interchange file, if the interchange file is not opened for reading, if the variable does not exist or have an **NC_CHAR type**, if the indices are not valid, memory cannot be allocated, or on an internal NetCDF error.

12. Peak Flags

12.1 Peak flag definitions are detailed in Specification E2077. Flag values are implemented as a set of #define bit fields. Table 24 shows the correspondence between flag specification and #define value. See Table 24.

12.2 Peak flags are referenced as arrays in the **MS_Raw_Per_Scan** data structure. The **flag_peaks** field is an array of long integers whose values represent the *indices* (where *zero* is the index of the *first* peak) of the peaks in the **masses** or **times** arrays which have flags assigned. The **flag_values** field is an array of **short** integers whose values are the flags for the corresponding peak. Flag values may be composite, formed by the logical OR of #define values from Table 24. For example, if the third peak in a scan is both saturated and a reference peak, then **flag_peaks** and **flag_values** would look like:

```
flags:          1
flag_peaks:     { 2 } /* zero is the first peak */
flag_values:    { MS_FLAG_SATURATED | MS_FLAG_REFERENCE }
```

12.3 Specific flag values may be extracted by logical AND of the flag value array element with the #define value: for example, (*value &* **MS_FLAG_SATURATED**) is non-zero if the **MS_FLAG_SATURATED** flag is set in *value*.

13. The ncopts and ncerr Variables

13.1 The NetCDF toolkit is generally not required when developing ASTM MS API applications. However, one int variable, **ncopts**, is exported from the NetCDF toolkit and must be assigned a value in order for the protocol to operate properly. This variable controls how the NetCDF toolkit handles errors. There are four values (defined in the file “netcdf.h” in the NetCDF toolkit):

(1) **ncopts = 0**: The NetCDF toolkit ignores all errors, and simply returns error codes.

(2) **ncopts = NC_FATAL**: The program exits on NetCDF toolkit errors.

(3) **ncopts = NC_VERBOSE**: The toolkit prints an error message and returns an error code.

TABLE 24 Peak Flags

#define	Specification Value
MS_FLAG_NOT_HRP	NOT HIGH RESOLUTION
MS_FLAG_MISSED_REF	MISSED REFERENCE
MS_FLAG_UNRESOLVED	UNRESOLVED
MS_FLAG_DBL_CHARGED	DOUBLY_CHARGED
MS_FLAG_REFERENCE	REFERENCE
MS_FLAG_EXCEPTION	EXCEPTION
MS_FLAG_SATURATED	SATURATED
MS_FLAG_SIGNIFICANT	SIGNIFICANT
MS_FLAG_MERGED	MERGED
MS_FLAG_FRAGMENTED	FRAGMENTED
MS_FLAG_AREA_HEIGHT	AREA/HEIGHT
MS_FLAG_MATH_MODIFIED	MATH MODIFIED
MS_FLAG_NEGATIVE	NEGATIVE INTENSITY
MS_FLAG_EXTENDED	EXTENDED ACCURACY
MS_FLAG_CALCULATED	CALCULATED
MS_FLAG_LOCK_MASS	LOCK MASS

(4) **ncopts = NC_VERBOSE | NC_FATAL**: The toolkit prints an error message followed by a program exit.

13.2 Under normal usage, **ncopts** should be set to zero, and the application should handle errors appropriately. Using the **NC_VERBOSE** option is useful when developing applications, but generally should not be used for code meant for public use.

13.3 An additional externally defined **int** variable, **ncerr**, contains a NetCDF-specific error code which can be used after an error has occurred to determine the nature of the error. However, because protocol errors can arise either at the NetCDF or API level, this variable may not be useful as a diagnostic.

14. Examples

14.1 *Reading a Protocol Interchange File*—The protocol Application Programming Interface and data structures provide all the tools necessary to read mass spectral data from an interchange file. The following steps must be followed when developing an application to read interchange files. It is not important which is read first, instrument or scan data.

(1) Declare protocol data structure variables (**MS_Admin_Data**, etc.).

(2) Open the interchange file for reading (**ms_open_read**).

(3) Initialize global data structures (**ms_init_global**, with *clear_flag* set to zero).

(4) Read global information (**ms_read_global**), then do something with it (for example, write it back out in another format).

(5) If the file contains instrument data and the application wants to read it, then for each instrument component, do:

(a) initialize the instrument component data structure (**ms_init_instrument** with the *clear_flag* set to zero),

(b) set the *inst_no* field of the **MS_Instrument_Data** structure to the index of the component, and

(c) read the component data (**ms_read_instrument**) and do something with it.

(d) clear the instrument component data structure (*ms_init_instrument* with the *clear_flag* set to one).

(6) Optionally, read the total ion chromatogram (**ms_read_TIC**), and free the TIC values array after use.

(7) Optionally, if the file contains selected ion detection data:

(a) read global group information (**ms_read_group_global**), then for each group do:

(b) initialize the per-group data structure (**ms_init_per_group** with the *clear_flag* set to zero,

(c) set the *group_no* field to the index of the group,

(d) read the group data (**ms_read_per_group**) and do something with it, and

(e) clear the per-group data structure (**ms_init_per_group** with the *clear_flag* set to one).

(8) For each scan, do:

(a) initialize the raw per-scan (and library, if the file contains library data) data structure(s) (**ms_init_per_scan**, with the *clear_flag* set to zero),

(b) set the *scan_no* field of the **MS_Raw_Per_Scan** (and **MS_Raw_Library**) data structure(s) to the scan index,

(c) read the per-scan data (**ms_read_per_scan**) and do something with it, and

(d) clear the per-scan data structure(s) (**ms_init_per_scan**, with the *clear_flag* set to one).

(9) Clear global data structures (**ms_init_global** with the *clear_flag* set to one).

(10) Close the file (**ms_close**).

14.1.1 The code fragment in Fig. 1 provides an example of reading a protocol interchange file.

14.2 *Writing a Protocol Interchange File*—Writing interchange files is similar to reading them; most of the same steps are followed, but instead of reading information and removing it from data structures, information is placed in data structures and then written to the interchange file. The following steps should be followed (again, it does not matter whether scan data or instrument data is written first):

(1) Declare data structure variables (**MS_Admin_Data**, etc.).

(2) Open the interchange file for writing (**ms_open_write**).

(3) Initialize global data structures (**ms_init_global**, with *clear_flag* set to zero).

(4) Fill the global data structures with information; there are two ways to handle character strings: either allocate space for the strings in the MS data structures and copy the string values into them, or simply load pointers to statically allocated strings. Which choice is taken determines whether to *initialize* or *clear* the data structures later.

(5) Write the global information (**ms_write_global**).

(6) If instrument data is available and the application wants to write it, then for each instrument component, do:

(a) initialize the instrument component data structure (**ms_init_instrument** with the *clear_flag* set to zero),

(b) set the *inst_no* field of the **MS_Instrument_Data** structure to the index of the component,

(c) fill the remaining instrument data structure fields with data; the same consideration with respect to strings applies here,

(d) write the component data (**ms_write_instrument**), and

(e) clear (if string copies were loaded) or initialize (if pointers to static strings were loaded) the instrument component data structure (**ms_init_instrument**).

(7) Optionally, if the experiment contains selected ion detection data:

(a) write global group information (**ms_write_group_global**), then for each group do:

(b) initialize the per-group data structure (**ms_init_per_group** with the clear_flag set to zero),

(c) set the group_no field to the index of the group,

(d) fill the remaining per-group structure field with data, making sure that all arrays are dimensioned to the maximum number of masses in any group,

(e) write the group data (**ms_write_per_group**), and

```
#include    <stdio.h>
#include    <stdlib.h>
#include    "netcdf.h"        /* only because of NC_VERBOSE */
#include    "ms10.h"

/* data structure declarations                                */

static MS_Admin_Data          admin_data;
static MS_Sample_Data         sample_data;
static MS_Test_Data           test_data;
static MS_Raw_Data_Global     raw_global_data;
static MS_Instrument_Data     inst_data;
static MS_Raw_Per_Scan        raw_data;
static MS_Raw_Library         lib_data;

extern int ncopts = NC_VERBOSE;     /* change to 0 when code is
                                       delivered as product */

main()
{
   long                 nscans;
   long                 ninst;
   long                 index;
   int                  file_id;
   int                  err_code;
   ms_admin_expt_t      expt_type;

   /* open the interchange file for reading                   */

   if ( MS_ERROR == (file_id = ms_open_read( "test.cdf" )) ) {
      fprintf( stderr, "Failed to open interchange file\n" );
      exit( 1 );
   }

   /* initialize global data structures                       */

   ms_init_global( 0, &admin_data, &sample_data,
                  &test_data, &raw_global_data );

   /* read global information                                 */

   if ( MS_ERROR ==
       ms_read_global( file_id, &admin_data, &sample_data,
                      &test_data, &raw_global_data ) ) {
      fprintf( stderr, "ms_read_global failed\n" );
      exit( 1 );
   }

   (do something with the raw data)
```

FIG. 1 Code Fragment

(f) clear (if data arrays were allocated) or *initialize* (if pointers to static arrays were copied) the per-group data structure (**ms_init_per_group**).

(8) For each scan, do:

(a) initialize the raw per-scan (and library, if the file contains library data) data structure(s) (**ms_init_per_scan**, with the *clear_flag* set to zero),

(b) set the *scan_no* field of the **MS_Raw_Per_Scan** (and **MS_Raw_Library**) data structure(s) to the scan index,

(c) fill the remaining raw data (and library) fields with information,

(d) write the per-scan data (**ms_write_per_scan**), and

(e) clear or *initialize* the per-scan data structure(s) (**ms_init_per_scan**).

```
nscans = raw_global_data.nscans;
ninst = admin_data.number_instrument_components;
expt_type = admin_data.experiment_type;

/* initialize the instrument data structure              */

ms_init_instrument( 0, &inst_data );

/* For each instrument component, read its data, do something
   with it, then clear the data structure                */

for ( index = 0; index < ninst; index++ ) {
   inst_data.inst_no = index;

   if ( MS_ERROR == ms_read_instrument( file_id, &inst_data ) ) {
      fprintf( stderr, "ms_read_instrument (component %ld) failed\n",
               index );
      exit( 1 );
   }

   (do something with the instrument component data)

   ms_init_instrument( 1, &inst_data );
}

/* read the total chromatogram                           */

if ( MS_ERROR == ms_read_TIC ( file_id, &nTic, &tic_values ) )
   fprintf ( stderr, "ms_read_TIC failed\n" );
   exit ( 1 );

}

(do something with the TIC data)

/* free the TIC values array (it was allocated during the call to
   ms_read_TIC)                                          */

free ( tic_values );

/* initialize the per-scan data structures               */

ms_init_per_scan( 0, &raw_data, &lib_data );

/* for each scan, read its data, do something with it, then
   clear the data structure                              */

for ( index = 0; index < nscans; index++ ) {
   raw_data.scan_no = index;

   if ( expt_library == expt_type ) {
      lib_data.scan_no = index;
      err_code = ms_read_per_scan( file_id, &raw_data, &lib_data );
   }
   else
      err_code = ms_read_per_scan( file_id, &raw_data, NULL );
```

```
   if ( MS_ERROR == err_code ) {
      fprintf( stderr, "ms_read_per_scan (scan %ld) failed\n",
               index );
      exit( 1 );
   }

   (do something with the raw and library data)

   ms_init_per_scan( 1, &raw_data, &lib_data );
}

/* clear the global data structures                    */

ms_init_global( 1, &admin_data, &sample_data, &test_data,
               &raw_global_data );

/* close the file                                      */

ms_close( file_id );

exit( 0 );
```

(9) Clear or *initialize* global data structures (**ms_init_global**).

(10) Close the file (**ms_close**).

14.2.1 The code fragment in Fig. 2 illustrates writing an interchange file.

15. Keywords

15.1 analytical; data; data communications; good automated laboratory practices (GALP); good laboratory practices (GLP); mass spectrometric data; method: NetCDF; raw data: results

```
#include<stdio.h>
#include<stdlib.h>
#include    "netcdf.h"        /* only because of NC_VERBOSE */
#include    "msl0.h"

/* data structure declarations                                  */

static MS_Admin_Data          admin_data;
static MS_Sample_Data         sample_data;
static MS_Test_Data           test_data;
static MS_Raw_Data_Global     raw_global_data;
static MS_Instrument_Data     inst_data;
static MS_Raw_Per_Scan        raw_data;
static MS_Raw_Library         lib_data;

extern int ncopts = NC_VERBOSE;     /* change to 0 when code is
                                       delivered as product */

main()
{
   long                 nscans;
   long                 ninst;
   long                 index;
   int                  file_id;
   int                  err_code;
   ms_admin_expt_t      expt_type;

   /* open the interchange file for writing; we will assume the
      following:

      experiment_type is expt_library,
      only mass data (no time data) is available,
      there are 5 scans in the library,
      there are two instrument components,
      mass data is in data_short format, and
      intensity data is in data_long format                     */

   nscans = 5;
   ninst = 2;
   if ( MS_ERROR ==
       (file_id = ms_open_write( "test.cdf", expt_library, nscans,
                                 ninst, data_short, data_short,
                                 data_long, TRUE, FALSE )) ) {
      fprintf( stderr, "Failed to open interchange file\n" );
      exit( 1 );
   }
```

FIG. 2 Code Fragment

```
/* initialize global data structures                          */

ms_init_global( 0, &admin_data, &sample_data,
                &test_data, &raw_global_data );


/* write global information, but first fill out some mandatory
   fields. Note that there are four mandatory fields in admin_data
   which are automatically loaded by the API (see the
   documentation of MS_Admin_Data, above). These can be
   ignored. Also, we have chosen to assign static strings
   rather than allocated storage for the two time stamps.
   Unless we manually clear these pointers, we must make
   sure to initialize rather than clear the global data. */

admin_data.experiment_type = expt_library;

/* note: both dates represent 11:48 pm, December 2, 1992 plus
   a few seconds, Greenwich Mean Time. The time differential,
   -0800, tells us that the local machine time is actually
   3:48 pm., a much more reasonable time to be at work. */

admin_data.experiment_date_time = "19921202234829-0800";
admin_data.netcdf_date_time = "19921202234857-0800";

raw_global_data.nscans = nscans;
raw_global_data.mass_format = data_short;
raw_global_data.time_format = data_short;
raw_global_data.intensity_format = data_long;

 (fill out any other global data fields of interest)

if ( MS_ERROR ==
    ms_write_global( file_id, &admin_data, &sample_data,
                     &test_data, &raw_global_data ) ) {
   fprintf( stderr, "ms_write_global failed\n" );
   exit( 1 );
}

/* initialize the instrument data structure                   */

ms_init_instrument( 0, &inst_data );

/* For each instrument component, fill out the data structure,
   write the data, then clear the data structure.             */

for ( index = 0; index < ninst; index++ ) {
   inst_data.inst_no = index;

    (fill out other instrument component fields)

   if ( MS_ERROR == ms_write_instrument( file_id, &inst_data ) ) {
      fprintf( stderr,
               "ms_write_instrument (component %ld) failed\n",
               index );
      exit( 1 );
   }
```

```
      /* this assumes inst_data strings were static...      */

      ms_init_instrument( 0, &inst_data );
   }

   /* initialize the per-scan data structures               */

   ms_init_per_scan( 0, &raw_data, &lib_data );

   /* for each scan, read its data, do something with it, then
      clear the data structure                              */

   for ( index = 0; index < nscans; index++ ) {
      raw_data.scan_no = index;

      (fill out other raw per-scan fields.  The number of points,
       number of flags, masses, (optionally times), and intensities
       are all required fields)

      if ( expt_library == expt_type ) {
         lib_data.scan_no = index;

      (fill out library fields)

         err_code = ms_write_per_scan( file_id, &raw_data, &lib_data );
      }
      else
         err_code = ms_write_per_scan( file_id, &raw_data, NULL );

      if ( MS_ERR0R == err_code ) {
         fprintf( stderr, "ms_write_per_scan (scan %ld) failed\n",
                  index );
         exit( 1 );
      }

      /* It is assumed that the raw per-scan and library data were
         loaded from dynamically allocated strings.  The clear_flag
         argument is therefore set to TRUE.                 */

      ms_init_per_scan( 1, &raw_data, &lib_data );
   }

   /* initialize the global data structures                        */

   ms_init_global( 0, &admin_data, &sample_data, &test_data,
                   &raw_global_data );

   /* close the file                                        */

   ms_close( file_id );

   exit( 0 );

}
```