Unofficial Format Specification of the IDL "SAVE" File

Craig Markwardt[1]

Last Modified
2011-12-21

Abstract

I describe the file format of IDL SAVE files, and a set of accessor routines written in IDL which can read and write SAVE files directly. Data and compiled functions in an IDL session can be stored on a disk file, and restored to a new session at a later point in time. The data are stored using a tag-based format to encourage forward and backward compatibility. Programmers who wish to implement the ability to read, write and interrogate SAVE files, and people who are just interested in "how it works" should read this document.

1 Introduction

IDL[2] is the Interactive Data Language, a proprietary data processing and visual analysis language published by Research Systems Incorporated. Its primary strengths are easy and efficient manipulation of arrays, and convenient visualization tools. One feature of IDL is the ability to save any number of IDL "variables" into a file. At any later point those variables can be restored to memory. This is a convenient way for a user to save all or part of their current work session, and the session can later be resumed by restoring the file. The files produced by the `SAVE` command, and reconstituted by the `RESTORE` command, are called "save" files by RSI. (See `SAVE` and `RESTORE` commands in the IDL Reference Manual.)

The format if IDL save files is not officially documented by RSI. They presumably do not document it because it is their "proprietary" information, but also because the reserve the right to change the format at any time, and thus do not wish to be constrained by an officially documented position. Indeed, as I detail below, I have found that IDL save file formats have changed slightly over time. Still, for the most part the file format has remained remarkably constant, and any extensions to the

[2]IDL is a trademark of Research Systems Incorporated (RSI)

format that RSI has introduced in new versions of IDL, have mostly attempted to remain compatible with previous versions of IDL.

I have developed a library of IDL routines which is able to interrogate, read and write IDL save files. To be clear, this library is not sanctioned or supported by RSI in any way. Also, there are no guarantees that RSI will keep the format of save files the same over time, however my intent is to support any changes that come up.

This is my attempt to document the library functions in a rudimentary way, and also to document the file format itself. Hopefully this will help other people trying to understand the format of the files, especially programmers of other scripting languages who are trying to construct read/write filters for IDL save files.

What follows is an overview of the IDL save format, followed by a brief discussion of the various levels of library functions, and finally a reference section describing in depth the format of each record found in a save file.

1.1 Scope

This set of documentation, and the accompanying library routines, are known to work with IDL save files produced by IDL version 4.0, and versions 5.0–5.5. They work with save files smaller than 2 gigabytes, that are uncompressed and do not contain objects. Compressed save files will not be accessible at all, and cannot be generated by the library. It will be possible to access and scan save files containing objects, but not possible to restore objects.

1.2 Disclaimer

Please note again that RSI has *not* sanctioned this specification, and I have discovered this information purely by examining IDL save files from known inputs on my own time. I believe that this information is correct, but cannot guarantee it, especially because the file format is vulnerable to change by RSI in the future. Use this information at your own risk.

2 File Format Overview

The IDL save file format is an extensible format. By this I mean that new record types can in principle be added without disturbing the overall format of the file, which enhances the probability that even an older version of IDL will be able to read the file, even if some information is lost.[3] This is possible because IDL files are stored in a standard "tagged record" format.

[3]However, even RSI did not appear to follow their own prescription when they released IDL version 5.4. In that version the record header (RECHEADER) format was changed incompatibly, so that earlier versions of IDL could not read files produced by IDL 5.4. This situation is apparently be corrected in the release of IDL version 5.5.

By "tagged record" format, I mean that the components of an IDL save file are each stored as a separate record type. Each record type is "tagged" or marked with a code which identifies the format of the record. Thus, the reader of the file can decide whether it is able to read a record based on its format code.

At its core level, the format of an IDL save file is very simple. It starts with a *signature* block which identifies the file as an IDL savefile, followed by a stream of records:

Format: **SAVEFMT** `(overall save file format)`

Type	**Name**	**Description**
BYTE×2	SIGNATURE	IDL SAVE file signature (=BYTE(['S','R']))
BYTE×2	RECFMT	IDL SAVE record format (=['00'xb, '04'xb])
Record 1		
Record 2		
⋮		
Record n — END_MARKER		

The meaning of the above table is is that the first two bytes of the file are a "signature," followed by two bytes which identify the record format of the file, followed by a series of standard records, the last of which is a standard END_MARKER record (described below).

These tables will appear throughout this document. The first column contains the *format or structure* of the element being described. In this case "BYTE×2" means 2 consecutive bytes. As more complex data structure formats are described, these may also appear in the first column. The second column contains the *name* of the element as used in this document. The final column contains any descriptive comments.

It should now be clear how an IDL save file should be read. The first process is to open the file and read the first four bytes. If the first four bytes do not match the expected signature, then the file is not in a recognized format and must be rejected. If the match succeeds, then the reader proceeds to read each record in turn, until the END_MARKER record is reached.

3 Overview of Library Routines

This CMSAVE library allows interactive users and programmers to read, write and interrogate IDL SAVE files.

Interactive users will probably appreciate the ability of CMSAVEDIR to list the contents of a SAVE file without restoring it. They will also probably use the /APPEND keyword of CMSAVE to append additional data to any SAVE file.

Programmers will probably use the library to write their own data, and to read user's existing data. The library provides higher-, middle- and lower-level routines for reading writing and querying a SAVE file.

For a complete documentation of the procedures, please see the individual files. Only an overview is given here.

3.1 High Level Routines

The high-level routines are meant to provide completely contained procedures which interact with save files. These routines are to be used either on the command line, or within other programs.

CMSAVE and CMRESTORE are drop-in, but more featureful replacements to the built-in IDL SAVE and RESTORE procedures. CMSAVEDIR allows you to print the contents of an existing save file without actually restoring it, and also to interrogate the contents of the file programmatically.

Unlike the built-in versions of RESTORE, CMRESTORE and CMSAVEDIR allow a variety of ways to access auxiliary data from a save file without actually restoring the variables from disk. The concept is that a program can scan the file and decide which actions to take based on the contents of the file, without restoring it first. This may be important for example, to prevent crucial variables from being overwritten, or to compile a class before restoring an object of that class.

For example, before restoring a file, here is a way to determine whether it contains system variables, which may be dangerous to restore:

```
; Extract variable names from save file
cmsavedir, 'myfile.sav', n_var=nvar, var_names=vnames, /quiet
; Find any variable names with exclamation mark '!'
sb = strpos(vnames, '!')
wh = where(sb GE 0, count)
if count GT 0 then $
  message, 'WARNING: myfile.sav contains system variables'
```

It is also possible to pass data to CMSAVE and CMRESTORE using the DATA keyword. Data can be passed as pointers, handles or structures. This allows for maximum flexibility and convenience for the programmer.

3.2 Mid Level Routines

The mid-level routines provide two procedures to read from and write variables to a save file. These routines are most useful to programmers who wish to provide the ability to save and restore IDL variables in a format native to IDL, with a minimum of fuss.

CMSVREAD and CMSVWRITE provide simple routines that read and write data from a save file. These routines are as easy to use as READF and PRINTF.

For example, to write the three variables A B and C to a save file, the following code is all that is necessary.

```
openw, 50, 'test.sav'       ;; Add /STREAM under VMS !
cmsvwrite, 50, a, name='a'
cmsvwrite, 50, b, name='b'
cmsvwrite, 50, c, name='c'
close, 50
```

Similar code is used to read variables from a file. However, save files can in principle contain pointers and objects, which CMSVREAD and CMSVWRITE cannot handle. For those cases it is best to use either the high- or low-level routines. The mid-level routines are best when it is known that a save file will have basic variables only, usually in a predetermined order.

3.3 Low Level Routines

There are a number of low-level routines available to advanced programmers. CMSV_OPEN is used to open save files for reading and writing (the normal CLOSE is used to close the file when finished). CMSV_RREC and CMSV_WREC read and write most record types from a save file. CMSV_RVTYPE and CMSV_RDATA read a variables type and data respectively; CMSV_WVTYPE and CMSV_WDATA perform the equivalent writing function.

It is best to examine the upper level routines such as CMSVREAD or CMSAVEDIR to understand how these high level routines are to be used.

Programmers who wish to use the CMSAVE library should first include the following code in their initialization routine. This will guarantee that the CMSAVE library is fully functional.

```
catch, catcherr
if catcherr EQ 0 then lib = cmsvlib(/query) else lib = 0
catch, /cancel
if lib EQ 0 then $
  message, 'ERROR: The CMSVLIB library must be in your IDL path.'
```

In place of the MESSAGE call, any graceful failure action can be taken.

4 File Format

This section and the following sections contain a detailed description of the IDL save file format. As described above, and IDL file consists of a stream of tagged records:

Format: **SAVEFMT** `(overall save file format)`

Type	Name	Description
BYTE×2	SIGNATURE	IDL SAVE file signature (=BYTE(['S','R']))
BYTE×2	RECFMT	IDL SAVE record format (=['00'xb, '04'xb])
Record 1		
Record 2		
⋮		
Record n — END_MARKER		

However, these records must appear in a certain order, which also depends on which version of IDL is being used. These records might be described as the save file "preamble," which give important metadata about the information. For IDL 4 this preamble consists of only one record:

Format: **PREAMBLE4** `(preamble records for IDL 4 save files)`

Type	Name	Description
TIMESTAMP	TIMESTAMP	Save file history information

The first record must be of type TIMESTAMP (see below for record types). Beyond that, save files produced by IDL 4 do not contain heap data. For IDL version 5 and beyond, the preamble is extended to include several more records after the TIMESTAMP record, which describes the file version information, such as which version of IDL was used to create the file:

Format: **PREAMBLE5** `(preamble records for IDL 5 save files)`

Type	Name	Description
TIMESTAMP	TIMESTAMP	Save file history information
VERSION	VERSION	Save session IDL information
HEAP_HEADER	HEAP_HEADER	(optional) Heap index information
—	PROMOTE64	(optional IDL 5.4 only) if file has ULONG64 offsets
NOTICE	NOTICE	(optional) Disclaimer notice

The HEAP_HEADER record only appears if the save file contains heap data (i.e., pointers). In IDL 5.4, the PROMOTE64 record indicates the presence of ULONG64 record offsets for storing files larger than 4 gigabytes (IDL 5.4 only).

5 Basic Record Structure

Each record has a variable length, but the beginning of the record has a standard format up until IDL version 5.4:

Format: **RHDR** `(format of every record header in save file)`

Type	**Name**	**Description**
LONG	RECTYPE	Numerical record format code
ULONG	NEXTREC0	Offset to next record, relative to start of file, low order 32-bits
ULONG	NEXTREC1	Offset to next record, relative to start of file, high order 32-bits
LONG×1	—	Unknown
—	—	Remainder of record

Thus, every record header contains at least four long words, but is often longer, depending on the record type. The first word contains the format type code. The type codes are described in more detail below, but there is only one format per code. The second and third words contain the offset of the *next record* measured from the beginning of the file. This is how a reader walks through an IDL save file: it must read each record header to determine the record type, and the offset to the next record. When the reader is finished with the current record, it must *seek* (or POINT_LUN) to the position of the next record.

IDL is inconsistent in how it handles file sizes of greater than 4 gigabytes. In IDL version 5.3 and earlier, there is no support for large files (i.e. `NEXTREC1` must be 0). IDL version 5.4 had a special way of handling such large files, with a different record header format, as described in the next paragraph. After IDL version 5.4, I believe that IDL added support for large files by allowing `NEXTREC1` to be non-zero. The position of the next record would be computed as: `NEXTREC = NEXTREC0 + NEXTREC1 *` 2^{32}.

Version 5.4 of IDL changed the record format in an incompatible way for large files. When dealing with a large file, IDL version 5.4 inserts a special record (`PROMOTE64`). If the `PROMOTE64` record is present in the file, then the record header format changes to the following hybrid:

Format: **RHDR64** `(format of every record header in save files after PROMOTE64 record appears)`

Type	**Name**	**Description**
LONG	RECTYPE	Numerical record format code
ULONG64	NEXTREC	Offset to next record, relative to start of file
LONG×2	—	Unknown

Note the change of LONG to ULONG64. This change increases the length of the record to 5 32-bit words instead of 4 32-bit words, and thus makes all records produced by IDL 5.4 incompatible with previous versions of IDL. This situation is dealt with using the PROMOTE64 keyword of the save library.

NOTE: where "RHDR" appears below as a data type, the actual data type is either RHDR or RHDR64, depending on whether the PROMOTE64 record has appeared in the record stream or not. Before PROMOTE64 appears, the normal record header is used (RHDR) and after, the hybrid header is used (RHDR64).

Obviously, the PROMOTE64 method is pretty crazy and should not be written by new software. However, software readers may need to be aware of such issues for compatibility with older versions of IDL software.

5.1 Record Types

Here is a table containing the record type codes, the record name, and a short description of the record contents.

0	`START_MARKER`	Start of SAVE file
1	`COMMON_VARIABLE`	Block contains a common block definition
2	`VARIABLE`	Block contains variable data
3	`SYSTEM_VARIABLE`	Block contains system variable data
6	`END_MARKER`	End of SAVE file
10	`TIMESTAMP`	Block contains time stamp information
12	`COMPILED`	Block contains compiled procedure or function
13	`IDENTIFICATION`	Block contains author information
14	`VERSION`	Block contains IDL version information
15	`HEAP_HEADER`	Block contains heap index information
16	`HEAP_DATA`	Block contains heap data
17	`PROMOTE64`	Flags start of 64-bit record file offsets
19	`NOTICE`	Disclaimer notice

All of these record types, except for VARIABLE, SYSTEM_VARIABLE and HEAP_DATA, are read using the CMSV_RREC low-level procedure.

6 Primitives

Having defined the general properties of records in an IDL save file, we will now proceed to a finer level of detail: the definition of primitive data types within the records themselves.

The most important data types are long integers and strings, referred to throughout this document as LONG and STRING. These quantities are naturally quite the same as the LONG and STRING data types found within IDL, but because in this document they represent names of data types as stored on disk, it is required to define their exact layout. The library procedures CMSV_RRAW and CMSV_WRAW are used to read and write these types of data values.

Please NOTE: the formats of quantities stored in IDL *variables* can be somewhat different than these raw quantities. This section only refers to how the fundamental units of the file are formatted. See Sec. 9 for the format of IDL variable data.

The integer types are stored in IEEE format, and are automatically converted by the CMSV_RRAW and CMSV_WRAW procedures. Note that *every* integer value is encoded according the following table, including those within record headers, and stored in string length values.

Format: **INTEGER_TYPES** `(list of integer data types)`

Type	**Name**	**Description**
BYTE	BYTE	Single native byte
BYTE×4	LONG	32-bit signed word, IEEE format
BYTE×8	ULONG64	64-bit unsigned word, IEEE format

The string type is another fundamental type. A string contains any variable-length stream of bytes. The first quantity is a LONG integer which describes the length of the string in bytes, followed by the bytes themselves.

Format: **STRING** `(format of string data type)`

Type	**Name**	**Description**
LONG	LENGTH	Length of the string in bytes
BYTE×LENGTH	CHARS	Characters of string, expressed as bytes. If the string is empty, i.e., if LENGTH is zero, then no CHARS are present. There is no zero termination.
BYTE×N	PAD	Padding bytes, to align stream to next 32-bit boundary. If the stream is already on a 32-bit boundary, or if LENGTH is zero, then no PAD bytes appear.

Thus, an empty (zero-length) string would simply be composed of a single LONG integer with the value of zero.

7 Record Descriptions

The following sections describe the actual formats of the records which have been outlined above. The "preamble" records (records which contain metadata) appear first, followed by the "content" records (which contain user data), and finally the end-of-file record is described.

7.1 Preamble Records

Preamble records must appear first in the file and typically describe the properties of the *file*, rather than the data itself. This metadata includes when the file was saved and by whom, and which version of IDL wrote it. By observation there appears to be an established order for these records to appear in a file, but it is not known whether this ordering is required or merely the default. The order appears to be:

1. TIMESTAMP — time and user information
2. VERSION — information about the version of IDL
3. PROMOTE64 — (optional) signals presence of large data

Format: **TIMESTAMP (10)** `(timestamp and user information)`

Type	**Name**	**Description**
RHDR	HDR	Block header record
LONG×256	—	Unknown (= LONARR(256))
STRING	DATE	Date session was saved, as a string.
STRING	USER	User name of saved session.
STRING	HOST	Host name of saved session.

NOTES: The TIMESTAMP record always appears to come first. Depending on the host operating system, the "user" and "host" fields may be empty or contain unreliable data. The format of the DATE field appears to be "WWW MMM DD hh:mm:ss YYYY" where the WWW is an abbreviation of the name of the day of the week, MMM is an abbreviation of the month name, DD is the day of the month, YYYY is the calendar year, and "hh:mm:ss" is the 24-hour time.

Format: **VERSION (14)** `(version information)`

Type	**Name**	**Description**
RHDR	HDR	Block header record
LONG	FORMAT	Save file format version number
STRING	ARCH	Architecture of saved session (= !VERSION.ARCH)
STRING	OS	OS of saved session (= !VERSION.OS)
STRING	RELEASE	IDL version of saved session (= !VERSION.RELEASE)

NOTES: The VERSION record type can be used to determine which version of IDL, and which OS was used to write a particular save file. It does not appear in files generated by IDL 4. The FORMAT entry is a file format version number, which appears to be incremented when the file format is changed. The following table describes the IDL software version where a given FORMAT value first appeared.

FORMAT value	IDL Version
5	5.3(?)
6	5.4(?)
7	5.6
8	6.0
9	6.1

Format: **PROMOTE64 (17)** `(signals start of RHDR64 record headers)`

Type	Name	Description
RHDR	HDR	Block header record

NOTES: Any records which appear *after* this record will have RHDR64 record headers in place of RHDR records. This record is not mandatory, and only appears in files generated by IDL version 5.4 or later.

Format: **IDENTIFICATION (13)** `(provides author information)`

Type	Name	Description
STRING	AUTHOR	Author of save file
STRING	TITLE	Title of the save file
STRING	IDCODE	Other author information

NOTES: This record is optional, and appears to provide more definitive information about who saved the IDL file. [This may be more useful for SAVE files which contain compiled procedures and functions.] There appears to be no direct way for users to specify this information from the built-in IDL SAVE command, but there does seem to be some connection to the EMBEDDED keyword to SAVE, and the creation of embedded-license save files.

Format: **HEAP_HEADER (15)** `(heap data table of contents)`

Type	Name	Description
LONG	NVALUES	Number of heap values in SAVE file
LONG× NVALUES	INDICES	Indices of heap values

NOTES: This record appears only if the file contains heap data, which is data *pointed to* by IDL pointers. Pointers are available only in IDL version 5.0 or later. The HEAP_HEADER record is merely a table of contents record which describes how many heap values there are, and lists there numerical index values. The use of the heap index values is described later in this document.

Format: **NOTICE (19)** `(disclaimer notice)`

Type	Name	Description
STRING	TEXT	ASCII text of disclaimer notice

NOTES: This record type is optional, and appears only in later versions of IDL.

7.2 Content Records

This section describes data which contain actual content, however, they contain subtypes which will require further elaboration in later sections.

Format: **COMMONBLOCK (1)** `(declaration of common block and variables)`

Type	Name	Description
RHDR	HDR	Block header record
LONG	NVARS	Number of common block variables
STRING	NAME	Name of common block
STRING ×NVARS	VARNAMES	Names of common block variables

NOTES: This record describes the existence, but not the values of, any common block variables. This is equivalent to the declarative COMMON statement in an IDL procedure. The names of the common variables are established, but the values must be defined by another separate record which appears later in the file.

Format: **VARIABLE** `(definition of standard IDL variable)`

Type	Name	Description
RHDR	HDR	Block header record
STRING	VARNAME	Name of IDL variable (upper case ASCII)
TYPEDESC	TYPEDESC	Variable type descriptor
LONG	VARSTART	Indicates start of data (= 7)
VARDATA	VARDATA	Variable data

Format: **SYSTEM_VARIABLE** `(definition of IDL system variable)`

Type	**Name**	**Description**
RHDR	HDR	Block header record
STRING	VARNAME	Name of IDL system variable (upper case ASCII, begins with '!')
TYPEDESC	TYPEDESC	Variable type descriptor
LONG	VARSTART	Indicates start of data (= 7)
VARDATA	VARDATA	Variable data

NOTES: The formats of standard IDL variables and system variables are the same. The only difference is the VARFLAGS flag byte within the TYPEDESC descriptor. The definitions of TYPEDESC and VARDATA are presented in sections 8 and 9. The TYPEDESC segment defines the type of the data, and is read and written by CMSV_RVTYPE and CMSV_WVTYPE. The VARDATA segment actually contains the data, and is read and written by CMSV_RDATA and CMSV_WDATA.

For scalars TYPEDESC is simply the IDL type number, but for arrays and structures the TYPEDESC descriptor can be much more complicated. Because the TYPEDESC descriptor can have a variable size, depending on the complexity of the type being described, it is not possible to parse the data segment of the record without first parsing the type descriptor.

Format: **HEAP_DATA** `(type and data for IDL heap variable)`

Type	**Name**	**Description**
RHDR	HDR	Block header record
LONG	HEAP_INDEX	The heap index of this particular variable. This index refers to the table of contents found in the HEAP_HEADER record.
LONG	—	Unknown ('02'XL or '12'XL)
TYPEDESC	TYPEDESC	Heap variable type descriptor
LONG	VARSTART	Indicates start of data (= 7)
VARDATA	VARDATA	Heap variable data

NOTES: A heap variable has the same format as an IDL standard or system variable, except that, instead of a STRING variable which provides the name of the variable, an integer number is used to identify which variable it is in the heap table of contents.

7.3 Compiled Procedures and Functions

It is outside of the scope of this document to describe the records which contain compiled procedures or functions. The beginning of the record has the following format:

Format: **COMPILED** `(compiled procedure or function)`

Type	Name	Description
RHDR	HDR	Block header record
STRING	PRONAME	Name of procedure or function (upper case ASCII)
LONG	LENGTH	Length of procedure (unknown units?)
LONG	N_VARS	Number of variables used in procedure
LONG	N_ARGS	Total number of parameters and keywords
LONG	PROFLAGS	Bit flags for procedure type, OR'd as follows: `'01'x` Record is a function; else, procedure `'02'x` Procedure accepts keywords `'08'x` Uses _EXTRA keyword parameter `'10'x` Unknown? (defined for every procedure) `'20'x` Record is a method `'40'x` Is a lifecycle method
	⋮ Undefined ⋮	

NOTES: The default is for all flags (except `'10'x`) to be zero, i.e., procedure which does not accept keywords and is not a method. Method-type procedures and functions apply to objects, are only meaningful in IDL 5.0 or greater. A "lifecycle method" is one which is used to create or destroy an object.

7.4 Concluding Record

The final record in an IDL save file is the END_MARKER record. It is empty, other than the record header that every record contains. I currently believe that any data following this record is ignored by IDL.

Format: **END_MARKER (6)** `(signals end of file)`

Type	Name	Description
RHDR	HDR	Block header record

NOTES: Applications can use the presence of this record to terminate processing.

8 Data Type Descriptor Formats

This section describes the format of the TYPEDESC segment, which appears in records of type VARIABLE, SYSTEM_VARIABLE and HEAP_DATA. All of these record types come in three parts. First, the *name* or *index number* of the variable appears, which describes its identity; second, the *type* of the variable is described, such as array dimensions, data type, and whether it is a structure; finally, the data itself appears. This section describes how the data types are stored on disk.

The data types can be divided into several possible cases:

1. Scalar numbers, strings or pointers.
2. Array numbers, strings or pointers.
3. Structures.
4. Objects.

By convention, structures are always arrays, so the data type descriptors for arrays and structures naturally overlap. While objects can be stored in IDL save files, the CMSV library does not support them.

8.1 Simple Scalar Quantities

The type descriptor for "simple" scalar quantities has the following format. Here "simple" means integers, floating point values (real and complex), strings, and pointers.

Format: **TYPEDESC_SCALAR** `(type descriptor for simple scalars)`

Type	Name	Description
LONG	TYPECODE	IDL variable type code
LONG	VARFLAGS	Bit flags for a type descriptor, OR'd as follows: `'02'x` System variable `'04'x` Array (UNSET for scalar) `'10'x` Unknown? `'20'x` Structure (UNSET for scalar)

NOTES: All of the type descriptor formats begin physically with a TYPECODE and VARFLAGS field. In the case of scalars, those are the only two fields in the descriptor. All scalars have a VARFLAGS value of zero or one, indicating either a system variable or not, but *not* an array or structure.

The IDL TYPECODE is the data type that is used throughout IDL, and by the SIZE function, to describe the basic type of a quantity. The values are contained in the following table:

IDL data type codes

0	Undefined (not allowed)
1	Byte
2	16-bit Integer
3	32-bit Long Integer
4	32-bit Floating Point Number
5	64-bit Floating Point Number
6	Complex Floating Point Number (32-bits each)
7	String
8	Structure (never a scalar)
9	Complex Floating Point Number (64-bits each)
10	Heap Pointer
11	Object Reference (not supported by CMSVLIB)
12	16-bit Unsigned Integer
13	32-bit Unsigned Integer
14	64-bit Integer
15	64-bit Unsigned Integer

The next sections describe the enhancements that are in place for arrays and structures.

8.2 Arrays

Arrays are an extension of simple data types. In addition to the type code and VARFLAGS field, there is an additional segment called ARRAY_DESC, which describes the structure of the array:

Format: **TYPEDESC_ARRAY** `(type descriptor for arrays)`

Type	**Name**	**Description**
LONG	TYPECODE	IDL variable type code
LONG	VARFLAGS	Bit flags for a type descriptor, OR'd as follows: `'02'x` System variable `'04'x` Array `'10'x` Unknown? `'20'x` Structure (UNSET for simple array)
ARRAY_DESC	—	Array descriptor segment

An array descriptor structure is a fixed-length segment describing the dimensions of the array.

Format: **ARRAY_DESC** `(array descriptor)`

Type	Name	Description
LONG	ARRSTART	Indicates start of array descriptor (= 8)
LONG	NBYTES_EL	Number of bytes per element of the array? (string= 12)
LONG	NBYTES	Number of bytes in array
LONG	NELEMENTS	Number of elements in array
LONG	NDIMS	Number of dimensions in array
LONG×2	—	Unknown (= [0,0])
LONG	NMAX	Number of stored dimensions (always 8?)
LONG×NMAX	DIMS	Dimensions of array, padded with 1s to NMAX

NOTES: The ARRAY_DESC descriptor indicates the size of the array in a number of different manners. The total number of elements is given by the NELEMENTS field. The number of dimensions in the array is given by NDIMS. Even if NDIMS is less than 8, the array descriptor always appears to contain NMAX (=8) stored dimensions. The "unused" dimensions are padded with values of 1. The fields NBYTES_EL and NBYTES appear to be unused by readers. There is some evidence that setting these fields to values of 0 does not harm the readers' ability to parse the file.

8.3 Structures

Structures are the most complex form of data stored in an IDL save file, because they can represent any combination of scalars, arrays, and structures. Thus, the format of the structure descriptor must be general enough to store any kind of data.

Format: **TYPEDESC_STRUCT** `(type descriptor for arrays)`

Type	Name	Description
LONG	TYPECODE	IDL variable type code
LONG	VARFLAGS	Bit flags for a type descriptor, OR'd as follows: `'02'x` System variable `'04'x` Array `'10'x` Unknown? `'20'x` Structure
ARRAY_DESC	—	Array descriptor segment
STRUCT_DESC	—	Structure descriptor segment

The array descriptor is described in the previous section. All structures are also arrays, and hence every structure type description will have both an ARRAY_DESC and STRUCT_DESC segment. Even "scalar" structures are considered to be an array of length and dimension equal to 1.

IDL objects are special cases of structures. The values themselves are stored as structures. The definition of the class uses the STRUCT_DESC segments as well, with special fields which give the name of the class and define the class's superclasses. If the INHERITS or IS_SUPER bit fields are set, then STRUCT_DESC segment can be considered to define a class rather than a structure.

The structure (and class) descriptor is defined in the following table.

Format: **STRUCT_DESC** `(variable structure descriptor)`

Type	**Name**	**Description**
LONG	STRUCTSTART	Indicates start of struct descriptor (= 9)
STRING	NAME	Name of structure (empty string indicates anonymous structure).
LONG	PREDEF	Structure definition bit flags, OR'd as follows: `'01'x` PREDEF - field is predefined struct `'02'x` INHERITS - field is a class `'04'x` IS_SUPER - field is a superclass `'08'x` Unknown? See NBYTES for the meaning of the (`PREDEF AND '01'x`) field.
LONG	NTAGS	Number of tags in structure.
LONG	NBYTES	Number of bytes in structure, using IDL notation. Actual number of bytes may vary from architecture to architecture. If PREDEF bit 1 is set, then this STRUCT_DESC refers to a previous definition, and this is the last field that appears. If PREDEF bit 1 is not set then this is a definition and the following fields appear.
TAG_DESC ×NTAGS	TAGTABLE	Table of tag descriptors, which describe the types of each tag.
STRING ×NTAGS	TAGNAMES	Table of names for tags.
ARRAY_DESC ×NARRAYS	ARRTABLE	If any tag is an array type, then its dimensions are described by the following ARRAY_DESC entries. The entries appear in the order they are in the structure; NARRAYS is determined from the values in TAGTABLE.
STRUCT_DESC ×NSTRUCTS	STRUCTTABLE	If any tag is a structure type, then its tags are described by these STRUCT_DESC entries. The entries appear in the order they are in the structure; NSTRUCTS is determined from the values in TAGTABLE. (Thus, a STRUCT_DESC can contain recursive STRUCT_DESC definitions.)
STRING	CLASSNAME	This field is present if either of the INHERITS or IS_SUPER bits are set (see PREDEF field above). If so, then the structure being defined is a class with name CLASSNAME.
LONG	NSUPCLASSES	This field is present if either of the INHERITS or IS_SUPER bits are set (see PREDEF field above). If so, then NSUPCLASSES is the number of superclasses this class inherits from.
STRING × NSUPCLASSES	SUPCLASSNAMES	This field is present if NSUPCLASSES is present and greater than zero. It is a list of this class's superclass names.
STRUCTDESC × NSUPCLASSES	SUPCLASSTABLE	This field is present if NSUPCLASSES is present and greater than zero. It is a squential list of descriptors for each of this class's superclasses.

NOTES: The length of this segment depends on whether it is *definitional* or *referential*. A definitional structure descriptor describes a structure the first time it appears in the data stream, and in that case the full descriptor is used. After the first time the structure has been defined, other appearances of the same structure may *refer* to the first definition by setting PREDEF to a value of one.

The end of the structure descriptor can itself contain one or more array, structure or superclass descriptors. Any array descriptors appear first, in the order that they appear in the structure; followed by any structure descriptors, in the order they appear in enclosing the structure. Finally, if particular STRUCT_DESC segment is a class definition, the class's superclass definitions, if any, are appended.

The decision of whether a tag is a structure, array or scalar is described by the TAG_DESC segment.

Format: **TAG_DESC** `Descriptor of a structure tag`

Type	**Name**	**Description**
LONG	OFFSET	Figurative "offset" of tag from start of struct. This number is typically meaningless since the true offset is architecture dependent.
LONG	TYPECODE	IDL variable type of tag value.
LONG	TAGFLAGS	Bit flags for variable type, OR'd together: `'20'x` Tag is a structure `'10'x` Tag may be an array (?) `'04'x` Tag is an array

It is not clear what the `'10'x` bit flag of TAGFLAGS means. As noted in the table, the "offset" value is usually not meaningful because it depends on a platform-dependent layout of structures.

9 Data Value Formats

For records of type VARIABLE, SYSTEM_VARIABLE and HEAP_DATA, the final segment of data in the record is the data itself (VARDATA). For all of these record types, the data format is the same.

For "simple" data types (as defined above), and arrays of simple data types, the data *usually* appears in its native format. However, some data types are translated into a slightly different form for the SAVE format. The following table specifies how this transformation is performed.

IDL Type	Storage Type & Format
BYTE	BYTE_DATA (see below)
Signed or Unsigned INT (16-bit)	(U)LONG (32-bit) IEEE
Signed or Unsigned INT (32-bit)	(U)LONG (32-bit) IEEE
Signed or Unsigned INT (64-bit)	(U)LONG64 (64-bit) IEEE
FLOAT (32-bit)	FLOAT (32-bit) IEEE
DOUBLE (64-bit)	DOUBLE (64-bit) IEEE
COMPLEX (32-bit)	2×FLOAT (32-bit) IEEE
DCOMPLEX (64-bit)	2×DOUBLE (64-bit) IEEE
STRING	STRING_DATA (see below)
POINTER	LONG (32-bit) IEEE (see below)

The alignment of the start of all types is on a 32-bit boundary. Array storage formats are packed as closely as possible, with no padding between array elements.

Byte values stored in variables have the following special BYTE_DATA format. Unlike other formats, the byte format appears to have a single 32-bit LENGTH header which indicates the number of bytes, followed by the byte data itself, and finally followed by a number of pad bytes in order to pad the stream to the next 32-bit boundary. The LENGTH information is redundant since length information can also be derived from the type descriptors.

Format: **BYTE_DATA** `(format of bytes within variable data)`

Type	Name	Description
LONG	LENGTH	Number of bytes stored
BYTE× LENGTH	BYTES	Byte data values
BYTE×N	PAD	Padding bytes, to align stream to next 32-bit boundary. If the stream is already on a 32-bit boundary, then no PAD bytes appear.

String data stored in variables also have a special format, different than the raw STRING format described in previous sections. The layout is given by STRING_DATA below.

Format: **STRING_DATA** `(format of strings within variable data)`

Type	Name	Description
LONG×2	LENGTH	Length of the string in bytes (length repeated)
BYTE× LENGTH	CHARS	Characters of string, expressed as bytes. If the string is empty, i.e., if LENGTH is zero, then no CHARS are present. There is no zero termination.
BYTE×N	PAD	Padding bytes, to align stream to next 32-bit boundary. If the stream is already on a 32-bit boundary, or if LENGTH is zero, then no PAD bytes appear.

The difference between STRING and STRING_DATA is that the LENGTH field is repeated twice rather than appearing only once for the STRING data type. String arrays are also packed together as tightly as possible.

Pointers are another special case. Pointers are stored as long integers. The integer is the same heap index value that appears in the HEAP_HEADER table of contents record, and the HEAP_DATA record. A pointer value of 3, for example, refers to the heap variable whose index value is also 3. Note that it is possible for more than one pointer to refer to the same heap variable.

Structures are obviously composed of more simple data types and (possibly) other structures. The individual components of a structure are stored, in order, packed on 32-bit boundaries.

10 Pointers and Heap Values

Pointer data stored in IDL SAVE files are particularly difficult to manage, because the actual heap variables are stored in separate records which *precede* the record of interest. Thus, if your application requires the reading of pointer data, you must perform special processing in your own code in order to support it. In essence, you must maintain an inventory of heap variables as they are encountered in the file.

If these procedures are not followed then pointer data will not be read, and a LONG integer value appears in the pointers' places. Under IDL 4, pointer data can never be read.

This is accomplished by placing some additional logic in your file processing loop. There are four separate components to this: (1) loop initialization; (2) reading a HEAP_INDEX record; (3) parsing a HEAP_DATA record; and (4) passing extra arguments to CMSV_RDATA. The additional state information is maintained in two variables named PTR_INDEX, which keeps track of the heap variable numbers, and PTR_OFFSETS, which stores the file location of each variable.

1. Loop initialization: is quite simple, use the following code:

```
ptr_index   = [0L]
ptr_offsets = [0L]
ptr_data    = [ptr_new()]
```

2. Reading HEAP_INDEX, which is an array of values indicating the heap variable numbers of each heap variables. These values are stored in PTR_INDEX:

```
CMSV_RHEAP, block, pointer, index, unit=unit
ptr_index   = [ptr_index, index]
ptr_offsets = [ptr_offsets, lonarr(n_elements(index))]
ptr_data    = [ptr_data, ptrarr(n_elements(index))]
```

3. Parse the HEAP_DATA record. Here were are interested in the heap variable number, and the file offset.

```
opointer = pointer
CMSV_RVTYPE, block, pointer, vindex, /heap, unit=unit

vindex = floor(vindex(0))
wh = where(ptr_index EQ vindex)
ptr_offsets(wh(0)) = offset + opointer
```

   Keep in mind that the file offset is OFFSET+POINTER.

4. Pass extra parameters to CMSV_RDATA. The user simply passes these extra variables to the CMSV_RDATA procedure, which automatically recognizes heap data and reads it from the appropriate location.

```
CMSV_RVTYPE, block, pointer, name, size, unit=unit, template=tp
CMSV_RDATA, block, pointer, size, data, template=tp, $
  unit=unit, ptr_offsets=ptr_offsets, $
  ptr_index=ptr_index, ptr_data=ptr_data
```

If this technique is used properly, only those heap variables which are needed are read. Thus, there are never any lost or dangling pointers. Since each bit of heap data is stored in a variable returned to the user, it is not necessary to `PTR_FREE(ptr_data)`; in fact, doing so would corrupt the input data.

11 Significant Changes

1. 2009-09-25 — Document the NOTICE (19) record type; document STRUCT-DESC descriptors which contain classes (and superclasses).

2. 2009-11-22 — Document the special format of byte scalars and arrays when stored in variables; reword the data-in-variables section.

3. 2010-01-11 — Document how large files are now supported by IDL, changing from NEXTREC to NEXTREC0 and NEXTREC1; better document how PROMOTE64 worked with IDL version 5.4; attempt to document which IDL software versions produced which file format versions.

4. 2011-12-20 — Document NBYTES_EL field of ARRAY_DESC (thanks to Yi Qin, Yi.Qin at csiro.au).