Chapter 3.

Description of Save File Format

This chapter covers the actual mechanics of the save file format. It explains the general file structure, how data is encapsulated, and the types of data written.

There are two types of save files denoted by their extension: save as text (.sat) and save as binary (.sab). The data elements of a .sat and a .sab are identical, except that sab files store it in a binary form. For the sake of clarity, most of the examples in this manual refer to a .sat file, because these files can be viewed and understood with a simple text editor.

Structure of Save File

A save file consists of a one or two line header record, an end marker for the file, and at least one data record between the header and end marker. If history is turned on and applicable, there may be some history records, as well.

```
Header               200 0 1 0
Header               23 ACIS Test Harness - 2.0

Entity Records       -0 body $1 $2 $-1 $-1 #
    .                 .
    .                 .
    .                 .
Entity Records       -25 point $-1 10 0 25 #
End Marker           End-of-ACIS-data
```

Figure 3-1. ACIS .sat File

Header

The first record of the ACIS part save file is a header, such as:

`200 0 1 0`

Integer . . . . An encoded version number. In the example, this is "200".

This value is 100 times the major version plus the minor version (e.g., 107 for ACIS version 1.7). For point releases the final value is truncated. Part save data for the .sat files is not affected by a point release (e.g., 105 for ACIS version 1.5.2).

Integer . . . . The total number of saved data records, or zero. If zero, then there needs to be an end mark.

Integer . . . . A count of the number of entities in the original entity list saved to the part file.

Integer . . . . The least significant bit of this number is used to indicate whether or not history has been saved in this save file.

Starting with ACIS 2.0, another record is part of the save file header.

```
23 ACIS Test Harness – 2.0 8 ACIS 2.0 24 Mon Feb 12 13:59:03 1996 25.4
1e–06 1e–10
```

String . . . . ID for product which produced file, "23 ACIS Test Harness".

String . . . . ACIS version which produced file (may be different from file version), "2.0 8 ACIS 2.0 24".

String . . . . Date file produced (in C ctime format), "Mon Feb 12 13:59:03 1996".

Double . . . . Number of millimeters represented by each unit in model, "25.4".

Real . . . . Value of resabs when file produced, "1e–06".

Real . . . . Value of resnor when file produced, "1e–10".

Entity Record

The header is followed by a sequence of entity records. These records are indexed sequentially starting at 0. Inclusion of the sequence number in the save file is optional. All top level entities must appear before any other entities. Thereafter, the record order is not significant.

```
-0 body $1 $2 $-1 $-1 #
     .
     .
     .
-25 point $-1 10 0 25 #
```

Pointers between entities are saved as integer index values, with NULL pointers represented by the value –1. ACIS pointer indices are preceded by $ in the sat file, or by a binary Tag 12 in the sab file. Refer to a complete breakdown of entity records under the subtitle "*Structure of Entity Record*" in this chapter.

Marker for Begin History Data

When the history save/restore option is turned on, a new section is added to the save file before the `End-of-ACIS-data` marker. This new section comes immediately after the information pertaining to the entities of the active model.

```
Begin-of-ACIS-History-Data
```

The new history section starts with a marker `Begin-of-ACIS-History-Data`. Everything between this section marker and the `End-of-ACIS-History-Section` marker obeys the rules of history save. This is covered in the chapter on classes in the section about history and roll back data.

Marker for End History Data

When the history save/restore option is turned on, a marker `End–of–ACIS–History–Section` immediately follows the history data. Between this marker and the `End–of–ACIS–data` marker may be more entity records. These adhere to the same entity record structure. Refer to a complete breakdown of entity records under the subtitle "*Structure of Entity Record*" in this chapter.

```
End–of–ACIS–History–Section
```

The purpose for this section of the save file is to list entities which may no longer exist at the active state. However, these entities did exist at some point during the creation of the current model and are necessary for roll back and roll forward operations. Entities that were part of pruned branches are not saved.

End Marker

The last entity record is followed by "End-of-ACIS-data" to mark the end of the ACIS save data.

```
End–of–ACIS–data
```

3

Structure of Entity Records

Each entity record consists of a sequence number (optional), an entity type identifier, the entity data, and a terminator.

Top level entities (e.g., body entities) are always the first records in the save file. Thereafter, the data records are in no particular order. When using ACIS tools, a call to api_save_entity_list with a list of *n* top level entities always places these entities in the first *n* records of the save file.

Pointers between data structures are represented by integer indices in the entity records. NULL pointers are represented by the integer –1. The indices are generated as the entities are being saved. The pointers are reconstructed when the entities are restored.

Optional Sequence Number

The sequencing of the indices in the entity record begins with 0. This sequence number itself can be turned on or off for the entity records in the save file. Even when the sequence number is not written to the file, it is implied by the order of the records in the file. Pointers to other records correspond to these implied sequence numbers.

```
-0 body $1 $2 $-1 $-1 #
     .
     .
     .
-25 point $-1 10 0 25 #
```

In the example above from a sat file, "–0" and "–25" are sequence numbers. In the first line, "$1 $2" happen to be pointers to records (not shown) with sequence numbers "–1" and "–2", respectively.

The optional sequence number represents the index assigned to a record and is intended to improve readability and simplify editing of ACIS save files. The option sequence_save_files controls whether ACIS writes sequence numbers to the part save file.

If sequence numbers are turned on, an entity may be deleted by simply removing its record from the save file. Any references to the removed record's index become NULL pointers when the file is restored by ACIS.

With sequence numbers on, records may also be rearrange within the file. However, if sequence numbers are turned off, a record cannot be simply moved or removed from the save file, because this will create invalid index referencing when the file is restored.

A mixture of records with sequence numbers and records without sequence numbers is permitted within the save file. Any record with no sequence number is given an index one greater than the previous entry in the file. Specified sequence numbers can be in any order. However, care must be taken that no sequence number repeat itself, either through manual specification of sequence numbers or the implied incrementing of other, nonspecified, sequence numbers.

Regardless of what sequence numbers they contain, the entities represented by the first records are assumed to be the top-level entities. Top-level entities are part of the ACIS topology. Also, if the total entity count was written in the header and the last record's sequence number is not one less than that count, a dummy record with a sequence number equal to the count must be added at the end of the file.

Entity Encapsulation in a Record

Each record generally encapsulates its information starting from the base ENTITY class. Because it is known that the encapsulation (and derivation) starts from ENTITY, its identifier is not written. In figure 3-2, this is shown in the middle by the empty double quotation marks, (""). Therefore, data for ENTITY is the first element written.

Starting from inside and going out, the ENTITY data is preceded by identifiers and is followed by the data for the identifiers. The identifiers correspond to classes derived directly from ENTITY. This type of encapsulation continues for all class derivations until the leaf class is reached.

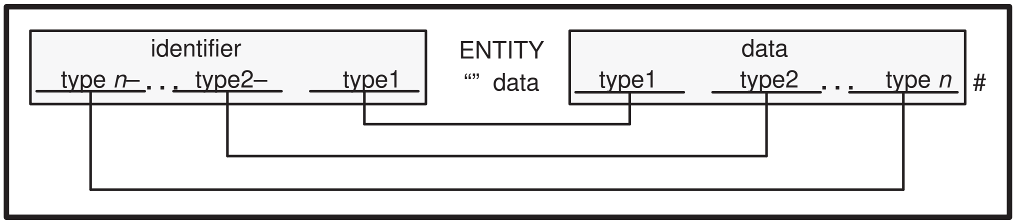

Figure 3-2. Format of Line in Data Record

In figure 3-2 for a sat file , the class identifiers are separated from one another with dashes, while their data fields are separated with white space. The white space is typically a space, but may also be a carriage return or new line. Some records are output with carriage returns for readability.

The last element of the record is the terminator character (#).

Save Identifier

3

The save identifiers are unique strings which denote their classes, and therefore the data associated with them. Chapter NO TAG is sorted alphabetically based on these save identifers. The derivation for a class from the base ENTITY class can be seen by the save identifiers.

For example, “body” is a save identifier for the BODY class, which is derived from ENTITY. Likewise, the save identifiers “plane–surface” might appear in a save file. These have a class derivation consisting of ENTITY, SURFACE, and PLANE. The save identifiers “colour–tsl–attrib” have the class derivation ENTITY, ATTRIB, ATTRIB_TSL, and ATTRIB_COL.

Unique save identifier strings are required for private attributes that a developer may create. This is achieved by deriving a private base class from ATTRIB and giving it a unique name. Then, developers may use any class name for their private attributes, yet the full identifiers are unique in the save file.

If the save identifiers are not completely recognized by ACIS, a data structure from just the *recognized* class(es) is constructed and restored. The remaining data at the end of the record is remembered so that it is not lost by a later save. For unrecognized classes whose derivation is two or more levels removed from ENTITY, such as classes derived from CURVE, SURFACE, or ATTRIB, the minimum is to create a recognizable data structure so that references to the data structure are correct. For example, if a record from a derived class of ATTRIB is not recognized, an ATTRIB record is created in such a way that the chain of attributes remains connected for the entity owning the unrecognized attribute.

Entity Data

The data for an entity is encapsulated in the order of its derivation from the basic ENTITY, from left to right. This is in the opposite order of the derivation of the identifier. The data for ENTITY is written first, followed by the data for the class directly derived from ENTITY, continuing down to the leaf class. For readability in the sat file, each data field is separated by white space: a space, carriage return, or new line.

Terminator

The terminator pound sign (#) marks the end of the entity data in the sat file, while Tag 17 marks it in the sab file. This allows unknown entities and attributes to be read and the start of the next entity to be located.

Data Elements

The decomposition of class derivations breaks the data elements into control, compound types, and several standard types. This section describes the compound types used by ACIS and the format of data actually written to the save file.

Control

if_cond	Conditional statement that is not written to the save file. Depending on how it evaluates, save data elements indented below the conditional statement are considered for inclusion in the save file.
else	Used as part of the "if" conditional. Not written to the save file.
else if_cond	Used as part of the "if" conditional. Not written to the save file.
repeat	Loop statement that is not written to the save file. Depending on the repeat arguments, save data elements indented below the repeat statement are stored zero, one, or more times in the save file.

Primitive Compound Types

integer	Written as long.
real	Written as double.

boolean	Written as long: if value == 0, then it is (False); if the value != 0, then it is (True).
interval	Low and High bounds are each written as logical(“I”, “F”), indicating whether interval is bound in that direction. If the interval is bound in a given direction, the logical is followed by the bound in that direction, written as a real.
matrix	Written as three vector, one for each row of the matrix.
transform	Written as – a matrix, representing the affine portion of the transformation. – a vector for the translation component, – a real scaling component, and – three boolean, indicating the presence of rotation, reflection, or shear, respectively.
newline	Used to make “save as text” files more readable. Simply adds newline white space.

Primitives for Text File Format (.sat)

Elements in a text file are written using the following format conventions. White space is used to delimit the various items.

char	Written with C printf format “%c”.
short	Written with C printf format “%d ”.
long	Written with C printf format “%ld ”.
logical	*(false_string, true_string, {or any_valid_string})*: Appropriate string written with C printf format “%s ”.
enum <identifier>	The <identifier> specifies which enumeration is active and its valid values. The <identifier> is not written to the file. A valid value only is written to the file. This is a character string or a long value from the enumeration <identifier> written with C printf format “%s ”. Refer to chapter 5.
float	Written with C printf format “%g ”.
double	Written with C printf format “%g ”.
string	Length written as long followed by string written with C printf format “%s”.
ident	The save identifier written with C printf format “%s ”.
subident	The save identifier followed by a dash (–), written with C printf format “%s–”.
position	*x, y, z* coordinates written as real’s.
vector	*x, y, z* components written as real’s.

3

subtype_start	Braces around the subtypes, written as “{ ”.
subtype_end	Braces around the subtypes, written as “} ”.
terminator	Written as “#”.
$rec_num	Pointer reference to a save file record index. Written as “$” followed by index number written as a long.
header	*(version, total, top, flags):* Values written as four long.
sequence	Written as “–” followed by the entity index written as long.

Primititves for Binary File Format (.sab)

Elements in a binary file are generally written as a one byte tag followed by the appropriate data, written with the size and format of the system on which it is written. Binary files are not guaranteed to be portable between platforms due to differences in sizes of data types, byte ordering, and floating point representations.

char	Tag 2, followed by the char value.
short	Tag 3, followed by the short value.
long	Tag 4, followed by the long value.
logical	*(false_string, true_string, {or any_valid_string})*: Tag 10 for a true value, or tag 11 for a false value.
enum <identifier>	The <identifier> specifies which enumeration is active and its valid values. The <identifier> is not written to the file. A valid value only is written to the file. This is Tag 21 followed by character string or a long value from the enumeration <identifier>. Refer to chapter 5.
float	Tag 5, followed by the float value.
double	Tag 6, followed by the double value.
string	Tag 7, followed by a char length value, followed by the specified number of characters, or Tag 8, followed by a short length value, followed by the specified number of characters, or Tag 9 followed by a long length value, followed by the specified number of characters.
ident	Tag 13, followed by a char length value followed by the specified number of characters.
subident	Tag 14, followed by a char length value followed by the specified number of characters.
position	Tag 19, followed by 3 doubles representing *x, y, z* coordinates.
vector	Tag 20, followed by 3 doubles representing x, y, z components.
subtype_start	Tag 15.

subtype_end	Tag 16.
terminator	Tag 17.
$rec_num	Pointer reference to a save file record number index. Written as Tag 12, followed by index cast to a void*.
header	*(version, total, top, flags)*: The characters "ACIS BinaryFile" followed by the supplied values cast to long.
sequence	Sequence numbers are not written to binary files.

Subtypes and References

Subtypes and references are frequently used within any given save file, because they reduce the amount of information that has to be stored in the file. Moreover, they also allow non-entity objects to be shared. When a particular record uses a subtype object which has already been written, the record simply references it rather than writing the entire set of data out again. These references are written out as "{ ref *n* }". The ref indicates that this particular item has already been written out to the save file. The *n* stands for the subtype reference number from the beginning of the file, starting with zero.

All subtype and reference designations are enclosed between subtype_start and subtype_end designators, which are curly braces "{ }" in a .sat file and Tag 15 and Tag 16 in a .sab file. Subtypes definitions are numbered in an index table starting with 0. This table is used when writing the file, but is not saved in the file. References made later in the file use the subtype index table numbering.