BioWare Aurora Engine

Generic File Format (GFF)

1. Introduction

The Generic File Format (GFF) is an all-purpose generic format used to store data in BioWare games. It is designed to make it easy to add or remove fields and data structures while still maintaining backward and forward compatibility in reading old or new versions of a file format.

The backward and forward compatibility of GFF was important to the development of BioWare's games because file formats changed rapidly. For example, if a designer needed creatures in the game to have a new property to store their Last Name, it was easy to add that field to the creature file format. New versions of the game and tools would write out the new field, and old versions would just ignore it.

In Neverwinter Nights, a BioWare game, most of the non-plain-text data contained in a module is in GFF, although the compiled scripts are a notable exception.

The following file types within a module are all in GFF:

- Module info file (ifo)
- Area-related files: area file (are), game object instances and dynamic area properties (git), game instance comments (gic)
- Object Blueprints: creature (utc), door (utd), encounter (ute), item (uti), placeable (utp), sound (uts), store (utm), trigger (utt), waypoint (utw)
- Conversation files (dlg)
- Journal file (jrl)
- Faction file (fac)
- Palette file (itp)
- Plot Wizard files: plot instance/plot manager file (ptm), plot wizard blueprint (ptt)

The following files created by the game are also GFF:

- Character/Creature File (bic)

2. File Format Conceptual Overview

2.1. General Description

A GFF file represents a collection of data. C programmers can think of this collection as a `struct`. Pascal programmers can think of it as a `record`. Each item (properly called a Field) of data has: a text Label , a data type, and a value. C programmers can think of Fields as member variables in a C struct.

The labelling of Fields is the key to GFF being able to maintain backward and forward compatibility between different file formats.

The above concepts are best illustrated by an example.

Example of GFF usage

Let us use a simplified example of a Waypoint object. A Waypoint has a Tag that is used for scripting purposes, and it has a location. In C++ code, a Waypoint might be defined as follows:

```
class TWaypoint
{
  TString m_sTag;
```

```
    float m_fPositionX;
    float m_fPositionY;
    float m_fPositionZ;
};
```

(Notes: In the above declaration, we have purposefully omitted member function declarations because they are irrelevant to this example. Also, assume that a string class called TString has already been defined)

Suppose we have an instance of a Waypoint tagged "ShopEntrance", located at (3.2, 5.7, 0.0) in some Area.

When saved to GFF, it would "look" like this:

```
"Tag"        string  "ShopEntrance"
"PositionX"  float   3.200
"PositionY"  float   5.700
"PositionZ"  float   0.000
```

Now suppose we decide to add a MapNote property, which is a string that appears when the player mouses over a Waypoint in the game's minimap, and a HasMapNote property to specify whether the Waypoint appears on the minimap in the first place. The C++ declaration for the Waypoint object might now be:

```
class TWaypoint
{
  TString m_sTag;

  bool m_bHasMapNote;      // this was added
  TString m_sMapNoteText;  // this was added

  float m_fPositionX;
  float m_fPositionY;
  float m_fPositionZ;
};
```

If we try to load our old Waypoint instance from before, the program would still correctly load the Tag and X, Y, Z coordinates because it reads them from the GFF file by finding their Labels and reading the associated data. The fact that we just inserted some extra data inbetween the Tag and coordinates is inconsequential. Reading Fields from a GFF file does not depend at all on the physical position of Fields' bytes relative to each other within the file.

In this example, it is a simple matter from a programming perspective to notice that the old Waypoint instance does not have a MapNoteText or HasMapNote property stored on it in the GFF file. On seeing this, it is equally simple to set default values for those properties if their GFF Fields were not found.

2.2. Field Data Types

Allowed GFF Field types are listed in the table below.

Table 2.2a: Field Type Descriptions

Field Type	Size (in bytes)	Description
BYTE	1	Unsigned single byte (0 to 255)
CExoLocString	variable	Localized string. Contains a StringRef DWORD, and a number of CExoStrings, each having their own language ID.
CExoString	variable	Non-localized string
CHAR	1	Single character byte
CResRef	16	Filename of a game resource. Max length is 16 characters. Unused characters are nulls.
DOUBLE	8	Double-precision floating point value
DWORD	4	Unsigned integer (0 to 4294967296)
DWORD64	8	Unsigned integer (0 to roughly 18E18)
FLOAT	4	Floating point value
INT	4	Signed integer (-2147483648 to 2147483647)
INT64	8	Signed integer (roughly -9E18 to +9E18)
SHORT	2	Signed integer (-32768 to 32767)
VOID	variable	Variable-length arbitrary data
WORD	2	Unsigned integer value (0 to 65535)
Struct	variable	A complex data type that can contain any number of any of the other data types, including other Structs.
List	variable	A list of Structs.

A few of these data types merit additional explanation.

Void data

The Void type is used to store a stream of raw bytes as a single item of data under a single label. Possible applications include storage of raw image or sound data. It is never a good idea to use the void type to directly dump a data structure (such as a user-defined C struct) from memory to disk. The other primitive data types should be used to store each structure element instead.

CExoString

The CExoString type is a simple character string.

The suggested maximum allowed length is 1024 characters, which is a limit defined primarily to conserve network bandwidth in the case of strings that need to be passed from server to game client.

CExoStrings are generally not used for text that the player might see, since it would be the same text regardless of the player's native language. Instead, CExoStrings are used largely for developer/designer-only text, such as Tags used in scripting.

CExoLocString

A CExoLocString is a string type used to store text that may appear to the user and has a number of features designed to allow users to see text in their own language.

1) StringRef: A CExoLocString always contains at least a single integer called a StringRef (or StrRef). A StrRef is an index into the user's dialog.tlk file, which contains text that has been localized for the user's language.

2) Embedded Strings with Language IDs: If the StrRef is -1, then dialog.tlk is not used, and instead, the localized text must be embedded within the CExoLocString. A CExoLocString may contain zero or more normal strings, each paired with a language ID that identifies what language the string should be displayed for.

In the presence of both embedded strings and a valid StringRef, the behaviour is up to the application. In the toolset, embedded strings take precedence, assuming that there is an embedded string for the user's language. As with CExoStrings, embedded strings in CExoLocStrings should be no more than 1024 characters.

The following is a list of languages and their IDs:

Table 2.2b: Language IDs for LocStrings

Language	ID
English	0
French	1
German	2
Italian	3
Spanish	4
Polish	5
Korean	128
Chinese Traditional	129
Chinese Simplified	130
Japanese	131

3) Gender: In addition to specifying a string by Language ID, substrings in a LocString have a gender associated with them. 0 = neutral or masculine; 1 = feminine. In some languages, the text that should appear would vary depending on the gender of the player, and this flag allows the application to choose an appropriate string.

The Struct Data Type

In addition to the primitive data types listed above, GFF has a compound data type called a Struct. A Struct can contain any number of data Fields. Each Field can be any one of the above listed Field types, including a Struct or List.

Regardless of what Fields are packed into a Struct, there is always a single integer value associated with the Struct, which is the Struct ID. This ID is defined by the programmer who is writing out GFF data, and can be used to identify a Struct without having to check what Fields it contains. In many cases, though, the programmer already knows--or can assume--the Struct structure simply by the context in which the Struct was found, in which case the ID is unimportant.

Every GFF file contains at least one Struct at the top level of the file, containing all the other data in the file. The Top-Level Struct does not have a LabelThe Top-Level Struct has a Struct ID of 0xFFFFFFFF in hex.

The List Data Type

A List is simply a list of Structs. Like all GFF Fields, a List has a Label.

Structs that are elements of a List do not have Labels.

2.3. Field Labels

Every Field has a Label, a text string that identifies it. This Label can have up to 16 characters.

When a program writes out a structure in memory to disk as a GFF file, each field in that memory structure is written out using the most appropriate GFF Field data type, and with a GFF Label associated with it.

It is this labelling of data elements that makes GFF useful for storing data structures that are subject to change. When new information is added to a data structure, old GFF files can still be read because the old data is fetched by Label rather than by offset. That is, no assumptions are made as to where one variable is relative to another in the file.

3. File Format Physical Layout

This section describes what the actual bytes are in a GFF file, where they are, and what they do.

The file format descriptions in this section use the following terminology:

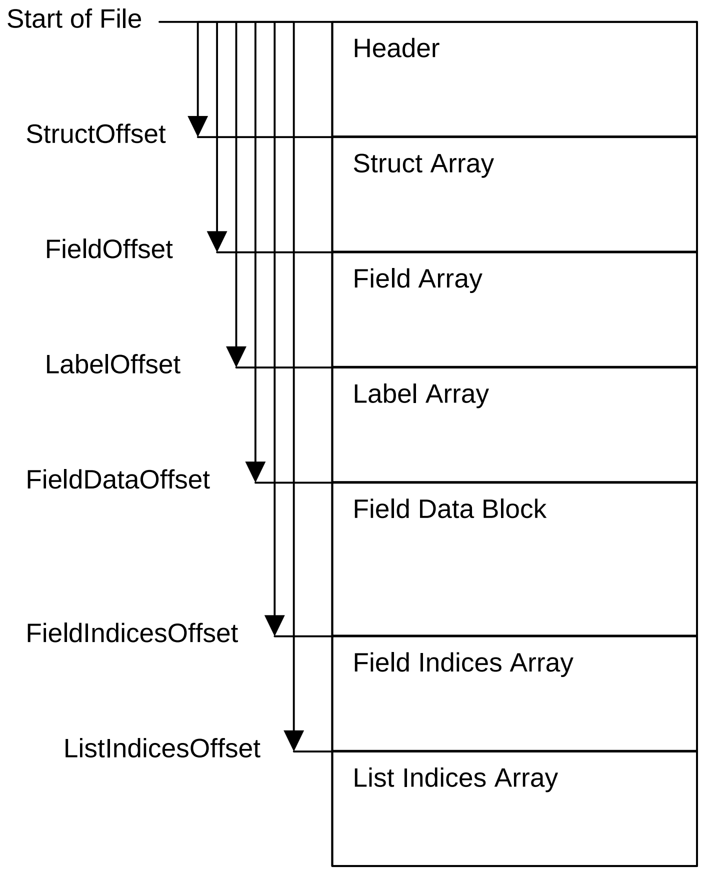
- BYTE: 1-byte (8-bit) unsigned integer
- CHAR: 1-byte (8-bit) character
- DWORD: 4-byte (32-bit) unsigned integer

Note that GFF byte order is little endian, which is the format used by Intel processors. If an integer value is more than 1 byte long, then the least significant byte is the first one, and the most significant byte is the last one. For example, the number 258 (0x0102 in hex) expressed as a 4-byte integer would be stored as the following sequence of bytes within the file: 0x02, 0x01, 0x00, 0x00.

3.1. Overall File Layout

A GFF file contains 7 distinct sections--1 fixed-size header, 5 arrays of fixed-size elements, and 1 block of raw data. The offset to each section is stored in the header, as is the number of elements in each array.

Figure 3.1: GFF File Structure



3.2. Header

The GFF header contains a number of values, all of them DWORDs (32-bit unsigned integers). The header contains offset information for all the other sections in the GFF file. Values in the header are as follows, and arranged in the order listed:

Table 3.2: Header Format

Value	**Description**
FileType	4-char file type string
FileVersion	4-char GFF Version. At the time of writing, the version is "V3.2"
StructOffset	Offset of Struct array as bytes from the beginning of the file
StructCount	Number of elements in Struct array
FieldOffset	Offset of Field array as bytes from the beginning of the file
FieldCount	Number of elements in Field array
LabelOffset	Offset of Label array as bytes from the beginning of the file
LabelCount	Number of elements in Label array
FieldDataOffset	Offset of Field Data as bytes from the beginning of the file
FieldDataCount	Number of bytes in Field Data block
FieldIndicesOffset	Offset of Field Indices array as bytes from the beginning of the file
FieldIndicesCount	Number of bytes in Field Indices array
ListIndicesOffset	Offset of List Indices array as bytes from the beginning of the file
ListIndicesCount	Number of bytes in List Indices array

The FileVersion should always be "V3.2" for all GFF files that use the Generic File Format as described in this document. If the FileVersion is different, then the application should abort reading the GFF file.

The FileType is a programmer-defined 4-byte character string that identifies the content-type of the GFF file. By convention, it is a 3-letter file extension in all-caps, followed by a space. For example, `"DLG "`, `"ITP "`, etc. When opening a GFF file, the application should check the FileType to make sure that the file being opened is of the expected type.

3.3. Structs

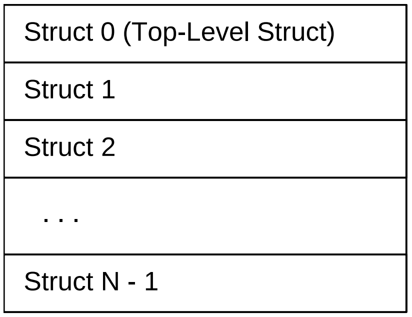
In a GFF file, Struct Fields are stored differently from other fields. Whereas most Fields are stored in the Field array, Structs are stored in the Struct Array.

The very first element in the Struct Array is the Top-Level Struct for the GFF file, and it "contains" all the other Fields, Structs, and Lists. In this sense, the word "contain" refers to conceptual containment (as in Section 2.1) rather than physical containment (as in Section 3). In other words, it does *not* imply that all the other Fields are physically located inside the Top-Level Struct on disk (in fact, all Structs have the same physical size on disk).

Since the Top-Level Struct is always present, every GFF file contains at least one element in the Struct Array.

The Struct Array looks like this:

Figure 3.3: Struct Array



Struct 0 is always present

N = Header.StructCount

N is always greater than or equal to 1

Physically, a GFF Struct contains the values listed in the table below. All of them are DWORDs.

Table 3.3: Struct Format

Value	Description
Struct.Type	Programmer-defined integer ID.
Struct.DataOrDataOffset	If Struct.FieldCount = 1, this is an index into the Field Array. If Struct.FieldCount > 1, this is a byte offset into the Field Indices array, where there is an array of DWORDs having a number of elements equal to Struct.FieldCount. Each one of these DWORDs is an index into the Field Array.
Struct.FieldCount	Number of fields in this Struct.

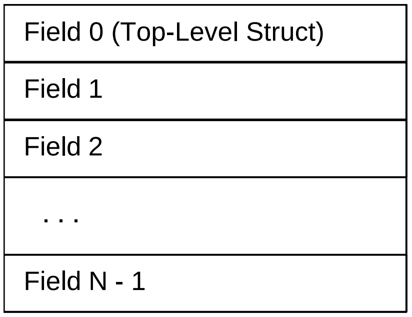
The above table shows that the Fields that a Struct conceptually contains are referenced indirectly via the DataOrDataOffset value.

Struct 0, which is the Top-Level Struct, always has a Type (aka Struct ID) of 0xFFFFFFFF.

3.4. Fields

The Field Array contains all the Fields in the GFF file except for the Top-Level Struct.

Figure 3.4: Field Array



N = Header.FieldCount

Each Field contains the values listed in the table below. All of the values are DWORDs.

Table 3.4a: Field Format

Value	Description
Field.Type	Data type
Field.LabelIndex	Index into the Label Array
Field.DataOrDataOffset	If Field.Type is a simple data type (see table below), then this is the value actual of the field. If Field.Type is a complex data type (see table below), then this is a byte offset into the Field Data block.

The Field Type specifies what data type the Field stores (recall the data types from Section 2.2). The following table lists the values for each Field type. A datatype is considered complex if it would not fit within a DWORD (4 bytes).

Table 3.4b: Field Types

Type ID	Type	Complex?
0	BYTE	
1	CHAR	
2	WORD	
3	SHORT	
4	DWORD	
5	INT	
6	DWORD64	yes
7	INT64	yes
8	FLOAT	
9	DOUBLE	yes
10	CExoString	yes
11	ResRef	yes
12	CExoLocString	yes
13	VOID	yes
14	Struct	yes*
15	List	yes**

Non-complex Field data is contained directly within the Field itself, in the DataOrDataOffset member. If the data type is smaller than a DWORD, then the first bytes of the DataOrDataOffset DWORD, up to the size of the data type itself, contain the data value.

If the Field data is *complex*, then the DataOrDataOffset value is equal to a byte offset from the beginning of the Field Data Block, pointing to the raw bytes that represent the complex data. The exact method of fetching the complex data depends on the actual Field Type, and is described in Section 4.

*As a special case, if the Field Type is a Struct, then DataOrDataOffset is an index into the Struct Array instead of a byte offset into the Field Data Block.

**As another special case, if the Field Type is a List, then DataOrDataOffset is a byte offset from the beginning of the List Indices Array, where there is a DWORD for the size of the array followed by an array of DWORDs. The elements of the array are offsets into the Struct Array. See Section 4.9 for details.

3.5. Labels

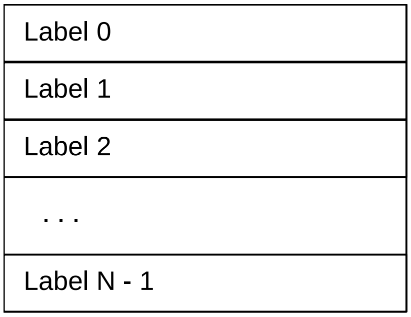
A Label is a 16-CHAR array. Unused characters are nulls, but the label itself is non-null-terminated, so a 16-character label would use up all 16 CHARs with no null at the end.

The Label Array is a list of all the Labels used in a GFF file.

Note that a single Label may be referenced by more than one Field. When multiple Fields have Labels with the exact same text, they share the same Label element instead of each having their own copy. This sharing occurs regardless of what Struct the Field belongs to. All Labels in the Label Array should be unique.

Also, the Fields belonging to a Struct must all use different Labels. It is permissible, however, for Fields in two *different* Structs to use the same Label, regardless of whether one of those Structs is conceptually contained inside the other Struct.

Figure 3.5: Label Array



N = Header.LabelCount

3.6. Field Data Block

The Field Data block contains raw data for any Fields that have a complex Field Type, as described in Section 3.4. The two exceptions to this rule are Struct and List Fields, which are not stored in the Field Data Block.

The FieldDataCount in the GFF header specifies the number of BYTEs contained in the Field Data block.

The data in the Field Data Block is laid out according to the type of Field that owns each byte of data. See Section 4 for details.

3.7. Field Indices

A Field Index is a DWORD containing the index of the associated Field within the Field array.

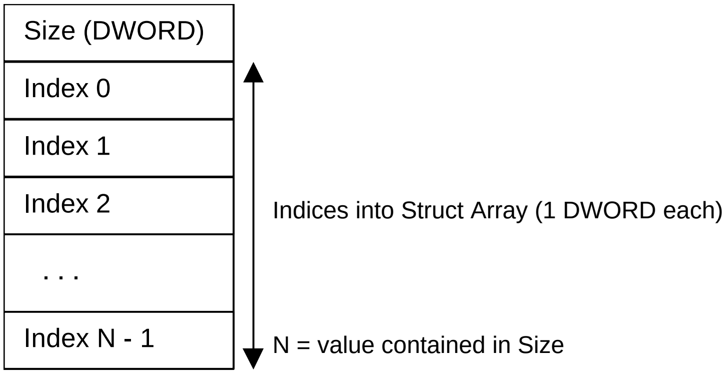
The Field Indices Array is an array of such DWORDs.

3.8. List Indices

The List Indices Array contains a sequence of List elements packed end-to-end.

A List is an array of Structs, and being array, its length is variable. The format of a List is as shown below:

Figure 3.8: List Format


Size (DWORD)
Index 0
Index 1
Index 2
. . .
Index N - 1
Indices into Struct Array (1 DWORD each)
N = value contained in Size

The first DWORD is the Size of the List, and it specifies how many Struct elements the List contains. There are Size DWORDS after that, each one an index into the Struct Array.

4. Complex Field Data: Descriptions and Physical Format

This section describes the byte-by-byte makeup of each of the complex Field types' data, as they are stored in the Field Data Block.

4.1. DWORD64

A DWORD64 is a 64-bit (8-byte) unsigned integer. As with all integer values in GFF, the least significant byte comes first, and the most significant byte is last.

4.2. INT64

An INT64 is a 64-bit (8-byte) signed integer. As with all integer values in GFF, the least significant byte comes first, and the most significant byte is last.

4.3. DOUBLE

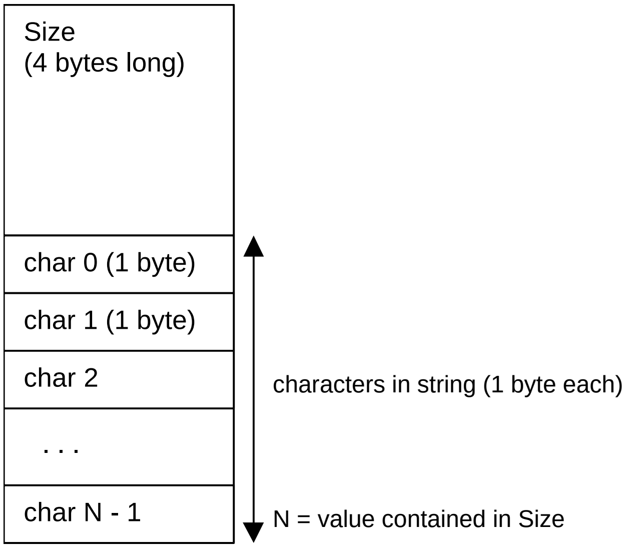
A DOUBLE is a double-precision floating point value, and takes up 8 bytes. It is stored in little-endian byte order, with the least significant byte first.

(Both the FLOAT and DOUBLE data types conform to IEEE Standard 754-1985).

4.4. CExoString

A CExoString is a simple character string datatype. The figure below shows the layout of a CExoString:

Figure 4.4: CExoString Format



A CExoString begins with a single DWORD (4-byte unsigned integer) which stores the string's Size. It specifies how many characters are in the string. This character-count does *not* include a null terminator.

If we let N equal the number stored in Size, then the next N bytes after the Size are the characters that make up the string. There is no null terminator.

Example: The string "Test" would consist of the following byte sequence:

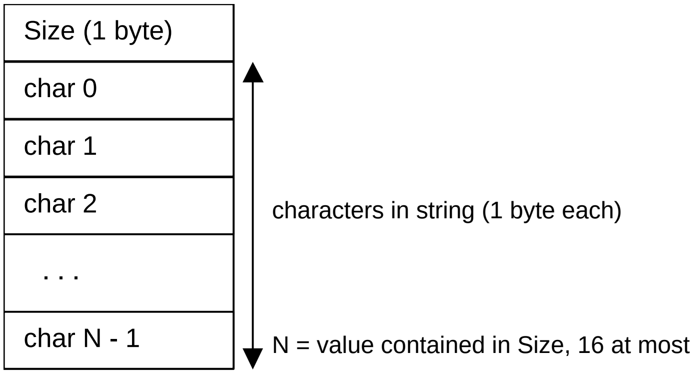
`0x04 0x00 0x00 0x00 'T' 'e' 's' 't'`

4.5. CResRef

A CResRef is used to store the name of a file used by the game or toolset. These files may be located in the BIF files in the user's data folder, inside an Encapsulated Resource File (ERF, MOD, or HAK), or in the user's override folder. For efficiency and to reduce network bandwidth, A ResRef can only have up to 16 characters and is not null-terminated. ResRefs are also non-case-sensitive and stored in all-lower-case.

The diagram below shows the structure of a CResRef stored in a GFF:

Figure 4.5: CResRef Format


Size (1 byte)
char 0
char 1
char 2
. . .
char N - 1
characters in string (1 byte each)
N = value contained in Size, 16 at most

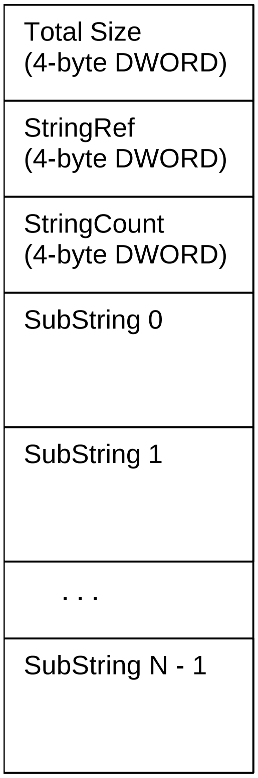
The first byte is a Size, an unsigned value specifying the number of characters to follow. The Size is 16 at most. The character string contains no null terminator.

4.6. CExoLocString

A CExoLocString is a localized string. It can contain 0 or more CExoStrings, each one for a different language and possibly gender. For a list of language IDs, see Table 2.2b.

The figure below shows the layout of a CExoLocString.

Figure 4.6a: CExoLocString Format



N = value contained in StringCount

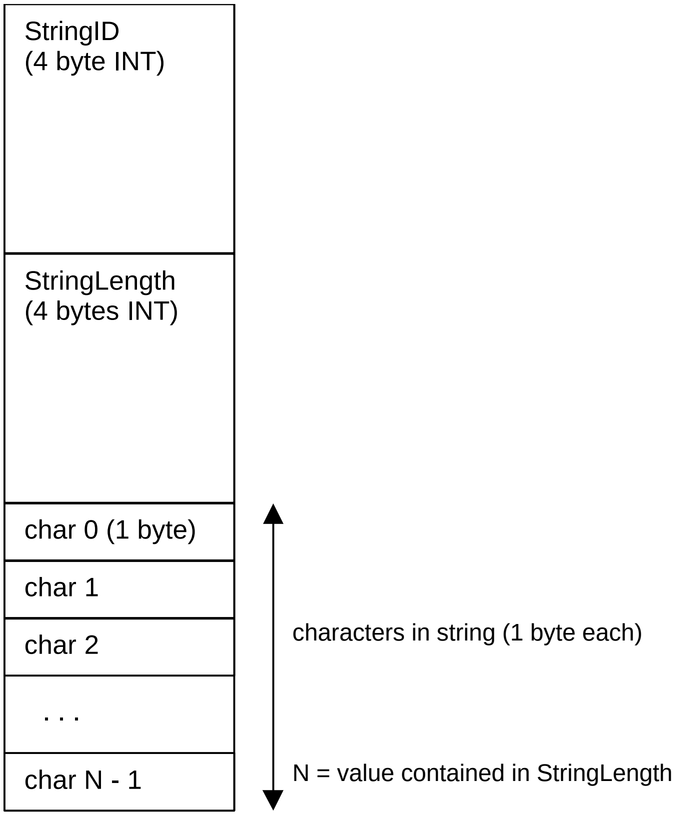
A CExoLocString begins with a single DWORD (4-byte unsigned integer) which stores the total number of bytes in the CExoLocString, not including the first 4 size bytes.

The next 4 bytes are a DWORD containing the StringRef of the LocString. The StringRef is an index into the user's dialog.tlk file, which contains a list of almost all the localized text in the game and toolset. If the StringRef is -1 (ie., 0xFFFFFFFF), then the LocString does not reference dialog.tlk at all.

The 4 bytes after the StringRef comprise the StringCount, a DWORD that specifies how many SubStrings the LocString contains. The remainder of the LocString is a list of SubStrings.

A LocString SubString has almost the same format as a CExoString, but includes an additional String ID at the beginning.

Figure 4.6b: CExoLocString SubString Format



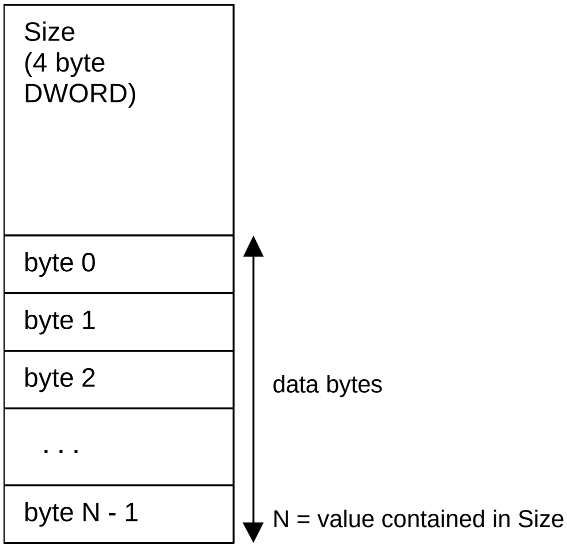
The StringID stored in a GFF file does not match up exactly to the LanguageIDs shown in Table 2.2b. Instead, it is 2 times the Language ID, plus the Gender (0 for neutral or masculine, 1 for feminine).

If we let N equal the number stored in StringLength, then the N bytes after the StringLength are the characters that make up the string. There is no null terminator.

4.7. Void/binary

Void data is an arbitrary sequence of bytes to be interpreted by the application in a programmer-defined fashion. The format is shown below:

Figure 4.7: Void Format



Size is a DWORD containing the number of bytes of data. The data itself is contained in the N bytes that follow, where N is equal to the Size value.

4.8. Struct

Unlike most of the complex Field data types, a Struct Field's data is located not in the Field Data Block, but in the Struct Array.

Normally, a Field's DataOrDataOffset value would be a byte offset into the Field Data Block, but for a Struct, it is an index into the Struct Array.

For information on the layout of a Struct, see Section 3.3, with particular attention to Table 3.3.

4.9. List

Unlike most of the complex Field data types, a List Field's data is located not in the Field Data Block, but in the Field Indices Array.

The starting address of a List is specified in its Field's DataOrDataOffset value as a byte offset into the Field Indices Array, at which is located a List element. Section 3.8 describes the structure a List element.