Binary Sequence Formats

The Xdna, SnapGene, and Gck sequence formats

Damien Goutte-Gattat <dgouttegattat@incenp.org>

2019/08/20

Abstract

This document intends to document binary sequence formats used by some molecular biology programs and which are, as far as I know, not documented anywhere else.

Table of Contents

1. Introduction 1
    1.1. History and Rationale for this Document 1
    1.2. Conventions Used in this Document 2
    1.3. Legal Notice 2
2. The Xdna Format 2
    2.1. Base Structure 3
    2.2. Extended Xdna Files 3
3. The SnapGene Format 5
    3.1. General Structure 5
    3.2. The Cookie Packet 5
    3.3. The DNA Packet 6
    3.4. Other Packets 6
4. The Gck Format 7
    4.1. The Header and the Sequence Packet 8
    4.2. The Features Packet and Associated Strings 9
    4.3. The Sites Packet and Associated Strings 10
    4.4. The Versions Section 10
    4.5. The Name and Flags Section 11

1. Introduction

1.1. History and Rationale for this Document

When I first started to work in a molecular biology lab, most of my then colleagues were using Apple computers and a particular, Mac OS-specific plasmid editor called DNA Strider. I was regularly given some sequence files in the native binary format of that program, which none of the applications available on my GNU/Linux desktop were able to read.

Between two experiments, I reverse-engineered the DNA Strider format and wrote a couple of small C programs to read a DNA Strider file and convert it into a FASTA or GenBank file, or conversely to read a FASTA file and convert it into a DNA Strider file. This was my Xdna2 project [https://incenp.org/dvlpt/xdna2.html].

A few years later, in another lab, I received from external collaborators some sequence files in the native format of another sequence editor called SnapGene. I envisioned for a while to add support for that format to my Xdna2 project, but quickly decided against it, mostly on the grounds that, given the complexity of the SnapGene format, implementing a parser in C code would have been too time-consuming.

Instead, I chose to develop the parser in Python, and since I was doing that, I also re-wrote my original DNA Strider parser in Python as well. I wrote both parsers in such a way that they integrate seamlessly with the SeqIO framework of Biopython. This was my BinSeqs [https://incenp.org/dvlpt/binseqs.html] project.

Later again, I came across files in another proprietary format, this time generated by a program called Gene Construction Kit. I again reverse-engineered the format and added a parser for it to the BinSeqs project. At around the same time, I submitted my parsers to the Biopython project. At the time of this writing (mid-August 2019), they have been merged into the master branch and will be part of the upcoming Biopython 1.75 release.

The code of the parsers may be read by anyone wishing to understand the formats—I tried to put enough explanatory comments for that purpose. However, I felt that it could be useful to have a standalone description of the formats in plain English, which would not require the reader to be fluent in Python. This is thus the purpose of this document.

I document here the DNA Strider, SnapGene, and Gene Construction Kit native formats, as I understood them. This is *not* an exhaustive documentation, but it should allow anyone wishing to read files in those formats to do so and to extract most of the information the files contain.

1.2. Conventions Used in this Document

In the schematics used to describe data structures, the following conventions are used:

- When two rows are depicted, the top row contains byte offsets from a reference position (e.g. the beginning of the file), while the bottom row indicates the nature of the data found at the corresponding offset. All offsets are expressed in hexadecimal.
- `B` denotes a single byte; an expression like `B(`*x*`)` denotes a block of *x* bytes.
- `H` and `I` denote short (16-bit) and long (32-bit) integers, respectively; a preceding `>` sign indicates that the byte order is big-endian.
- `Pad(`*x*`)` denotes *x* bytes of padding.
- `P(`*name*`)` denotes a 8-bit Pascal string.
- In offsets or in expressions indicating a number of bytes, a value prefixed by a * means the value is an offset and should be replaced by the value stored at that offset (this is analog to dereferencing a pointer in C).

Unless otherwise noted, all “Pascal strings” (sometimes abbreviated as P-strings) referred to in the text are 8-bit Pascal strings, meaning that the length of the string is stored in a single byte ahead of the first character of the string. In some places, Pascal string variants in which the length is stored in 4 bytes (in big-endian order) are also encountered, they will always be explicitly referred to as “32-bit Pascal strings”.

1.3. Legal Notice

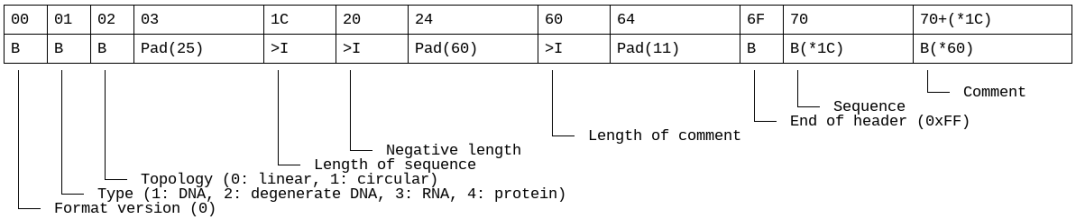
2. The Xdna Format

The *Xdna* format is the native format used by Christian Marck’s DNA Strider program for Mac OS. It is also used by Serial Cloner [http://serialbasics.free.fr/Serial_Cloner.html].

2.1. Base Structure

A basic Xdna file comprises three sections in a fixed order: a header, the sequence, and a comment (the comment being optional). The general structure is depicted in Figure 1.

Figure 1. Structure of a Xdna file



The header has a fixed size of 112 bytes. It starts with a byte giving the version number of the format, which seems to always be zero (I have never seen a Xdna file with a different version). It ends with a byte which seems to always be 0xFF and whose meaning is unknown (it may be simply to mark the end of the header).

After the version byte come two bytes (byte 2 and 3) indicating the type of sequence stored in the file (1 denotes a DNA sequence, 2 a degenerated DNA sequence, 3 a RNA sequence, and 4 a protein sequence) and the topology of the sequence (0 denotes a linear sequence and 1 a circular sequence), respectively.

The header contains three lengths which are all stored as big-endian long integers (4 bytes). There is the length of the sequence (bytes 29–32), the length of the comment (bytes 97–100), and the *negative length* (bytes 33–36). The “negative length” is the length of the part of the sequence before the base considered as the “origin” (base number 1, which in DNA Strider is not always the first base).

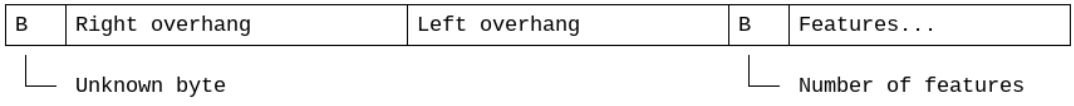
Serial Cloner has no such concept of a sequence origin and always generates files with a negative length of zero.

The sequence itself starts immediately at the first byte after the header (byte 113), and runs for as many bytes as indicated in the header’s sequence length field. The sequence is followed by the optional comment, without any separating byte(s)—the first byte after the sequence is the first character of the comment. The comment may be empty, in which case bytes 97–100 contain zero and the file ends after the last byte of the sequence (unless it is an “extended” Xdna file with an annotation section, see below).

2.2. Extended Xdna Files

Some Xdna files contain an additional annotation section after the comment (or immediately after the sequence if the comment is empty). This is typically the case of files generated by Serial Cloner (actually, I believe this additional section might be an extension of the format, created by Serial Cloner; I have never seen a DNA Strider-generated Xdna file containing such a section).

Figure 2. Structure of the Xdna annotation section



The annotation section (Figure 2) starts with a single byte whose meaning is unknown (it might be there solely to indicate the presence of the annotation section), then con-

tains two variable-length fields describing optional right-side and left-side overhangs (described in the following section). Then, a single byte gives the number of sequence features, which are then stored in as many feature structures (described in a later section) until the end of the file.

2.2.1. Overhang Specifications

Each overhang (right and left) is represented by a Pascal string containing the text representation of the overhang length, followed by the actual sequence of the overhang.

A length of zero indicates there is no overhang; a strictly positive length denotes a 5' overhang, and a strictly negative length denotes a 3' overhang.

Here are some examples of overhang specifications:

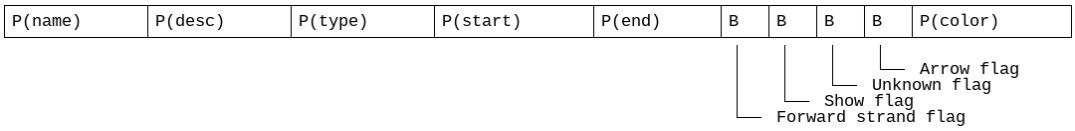
- `0x00 0x30`: indicates no overhang (`0`, encoded as a Pascal string);
- `0x01 0x32 0x41 0x41`: denotes a 5' AA overhang (P-string `2`, then `AA`);
- `0x02 0x2D 0x31 0x43`: denotes a 3' C overhang (P-string `-1`, then `C`).

2.2.2. Features

After the left overhang specification comes a single byte indicating the number of features (a Xdna file therefore cannot contain more than 255 features). If there's no features, that byte is zero and the file ends here.

If there are features, they are stored one after the other after that last byte. Each feature contains 6 fields and 4 flags (Figure 3).

Figure 3. Structure of a Xdna feature



All fields are stored as Pascal strings. They are, from the first to the last:

- the displayed name of the feature;
- a description of the feature;
- the type of the feature, which may be any of the features type supported by GenBank (e.g., `misc_feature`, `CDS`, `exon`, etc.);
- the start position of the feature (counting from 1, not 0), stored as text;
- the end position of the feature, stored as text;
- the text representation of a RGB triplet (3 comma-separated numbers from 0 to 255, e.g. `127,127,127,`) indicating the color used to paint the feature on a sequence or plasmid map.

Since the start and end positions of the feature are stored as text, the format could theoretically support fuzzy locations (but not compound locations), by using a notation similar to the one used in GenBank flat files (e.g., `<5` to denote a start position located anywhere before the fifth base). However neither DNA Strider nor Serial Cloner support such notation and only exact locations may be used.

The *description* field may contain a simple free-form comment on the feature, but may also contain GenBank-like qualifiers. In that case, the field is structured in lines (separated by carriage returns, `\r`), the first line being a free-form comment, and the following lines containing key-value pairs. Here is an exemple of a formatted description field (line feeds inserted for clarity):

```
Free-form comment on the first line\r
gene="bla"\r
product="beta-lactamase TEM"\r
function="ampicilin resistance"
```

The four flags are stored between the fifth and sixth fields, each flag being a single byte. The first flag indicates the strand the feature is on, for DNA sequences (reverse strand if the flag is not set, forward strand if it is); the second flag indicates whether the feature should be displayed on a sequence or plasmid map; and the fourth flag indicates whether the feature should be decorated with an arrow. The meaning of the third flag is unknown.

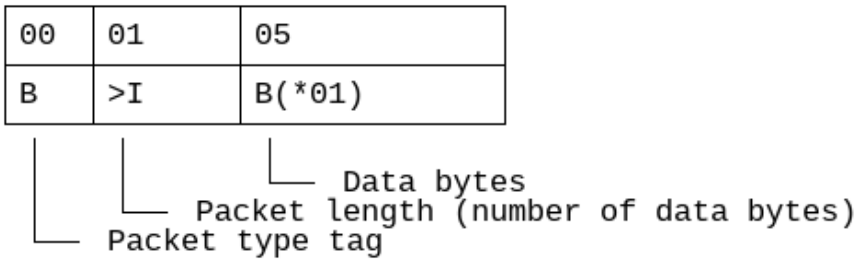
3. The SnapGene Format

The *SnapGene* format (typical file extension: `.dna`) is the native format of the SnapGene [https://www.snapgene.com/] program from GSL Biotech.

3.1. General Structure

A SnapGene file is a succession of *packets*. A packet starts with a single byte indicating the type of the packet (and the nature of the data it contains). Then a big-endian long integer gives the number of data bytes in the packet, and the actual data bytes start immediately after that (Figure 4).

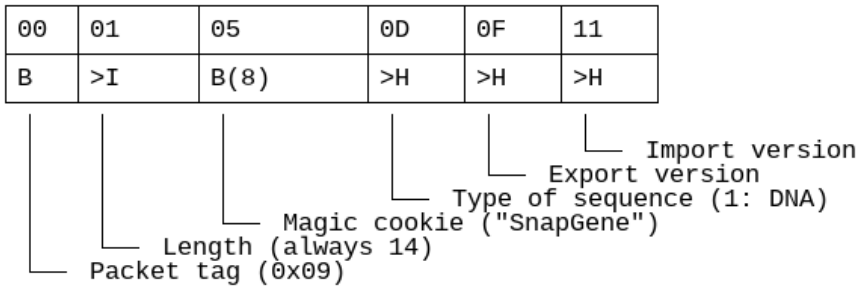
Figure 4. Structure of a SnapGene packet



Packets in a SnapGene file may be encountered in any order, except for the Cookie Packet which must always be the first packet in the file.

3.2. The Cookie Packet

Figure 5. Structure of the SnapGene cookie packet



This packet (Figure 5) is identified by the tag `0x09`. The data bytes start with 8 bytes encoding the string `SnapGene`, which acts as a magic cookie allowing to identify a SnapGene file.

The cookie is followed by three big-endian short integers, the first of those indicating the type of sequence stored in the file (which is always 1 for a DNA sequence).

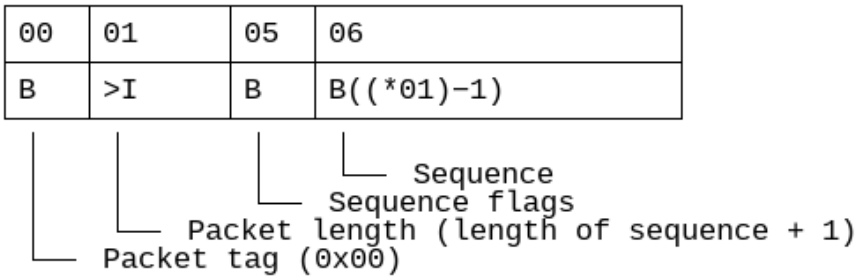
Given its fixed structure, the Cookie Packet always contains 14 data bytes and therefore a SnapGene file always starts with the following sequence of bytes:

```
0x09 0x00 0x00 0x00 0x0E 0x53 0x6E 0x61 0x70 0x47 0x65 0x6E 0x65
  |  \_________________/   |    |    |    |    |    |    |    |
  |   Length (14 bytes)   'S'  'n'  'a'  'p'  'G'  'e'  'n'  'e'
  +- Packet tag
```

3.3. The DNA Packet

This packet (Figure 6) has the tag `0x00`. A SnapGene file cannot contain more than one packet of this type.

Figure 6. Structure of a SnapGene DNA packet



The first data byte is a flag byte. The second bit of that byte is the *Topology flag*, which indicates a circular sequence (if set) or a linear sequence (if not set).

The remaining data bytes contain the sequence itself, encoded in ASCII. Therefore the length of the sequence is the number of data bytes in the DNA Packet minus one byte for the flag byte.

3.4. Other Packets

3.4.1. The Notes Packet

This packet has the tag `0x06`. It contains some metadata about the sequence. The data is a text representation of a XML tree starting with a root node named `Notes`.

Interesting nodes in that tree may include:

- the `Type` node, whose value can be `Synthetic` or `Natural`;
- the `Created` and `LastModified` nodes, which both contain a date in the *`YYYY.M.D`* format;
- the `AccessionNumber` node;
- the `Description` and `Comments` nodes, which contain free-form remarks about the sequence.

The value of the `Description` and `Comments` nodes contains XML-escaped HTML tags, as in the following example:

```
<Comments>&lt;html>&lt;body>Sample comments&lt;/body>&lt;/html></Comments>
```

3.4.2. The Features Packet

This packet has the tag `0x0A`. It contains the text representation of a XML tree starting with a root node named `Features` which lists the features found in the sequence.

Each feature is represented by a XML node named `Feature`. That node may contain the following attributes:

- a `name` attribute, containing a free-form name given to the feature;
- a `type` attribute, containing a GenBank-like feature type;
- a `directionality` attribute, whose value may be 0 for a non-directional feature (in that case, the attribute is generally absent altogether), 1 for a feature on the forward strand, 2 for a feature on the reverse strand, and 3 for a bi-directional feature).

A `Feature` node contains one or several `Segment` child node(s) giving the sequence coordinates of the feature. Each `Segment` node has a `range` attribute whose value is of the form *XXX*`-`*YYY*, where *XXX* is a 1-based start coordinate and *YYY* is the end coordinate.

After the `Segment` node(s), the `Feature` node may also contain `Q` nodes representing feature *qualifiers*. Each `Q` node has a `name` attribute giving the name of the qualifier, and a `V` child node for the value of said qualifier. The value itself is stored in a `text` or `int` attribute depending on the type of the qualifier. Textual values contain XML-escaped HTML tags.

Here is a (simplified) example of the XML that may be found in a `Features` packet:

```
<Features>
   <Feature name="SV40_term" directionality="1" type="terminator">
    <Segment range="400-750" />
    <Q name="note">
     <V text="&lt;html>&lt;body>SV40 poly-adenylation signal&lt;/body>&lt;/html>" />
    </Q>
   </Feature>
  </Features>
```

3.4.3. The Primers Packet

This packet has the tag `0x05`. It is similar to the Features Packet but is specifically designed to contain primer binding data, in a XML tree with a root node named `Primers`.

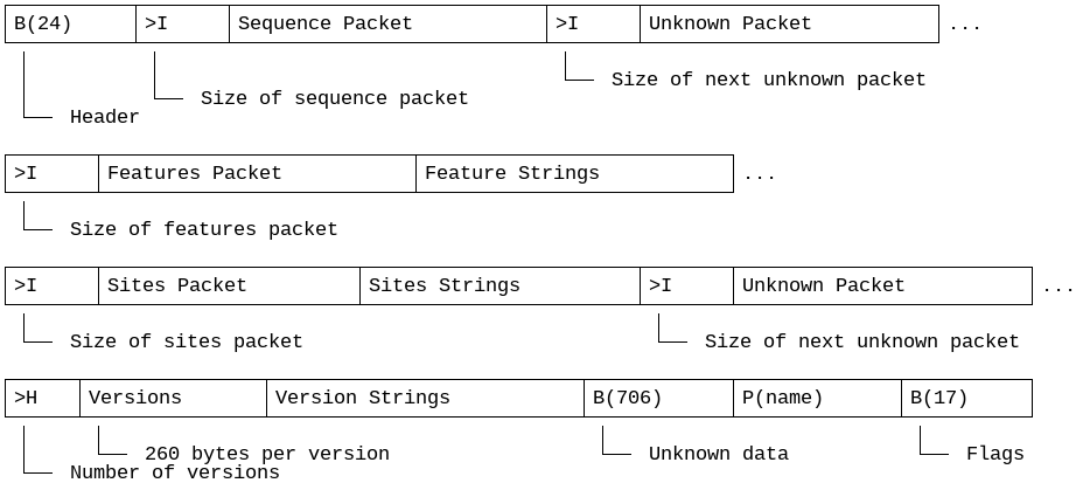
Each primer is represented by a `Primer` node with a `name` attribute (the primer's name, logically enough), a `sequence` attribute (the full sequence of the primer), and a `description` attribute (a free-form description, with XML-escaped HTML tags).

The `Primer` node contains one or several `BindingSite` child node(s) with a `location` attribute (containing a coordinates pair in the *XXX*`-`*YYY* format, as for the `Segment` node found in the Features Packet) and a `boundStrand` attribute whose value is either 0 (primer binding to the forward strand) or 1 (primer binding to the reverse strand).

4. The Gck Format

The *Gck* format is the native format of the Gene Construction Kit [http://www.textco.com/gene-construction-kit.php] program from Textco Biosoftware.

Figure 7. General structure of a Gck file



The overall structure of a Gck file is depicted in Figure 7. This is a somewhat weird mix of fixed-sized blocks (the header, the unknown 706-byte block near the end, and the 17-byte flags section), length-prefixed *packets*, ad-hoc structures (the versions), and lists of strings.

All the types of *packets* start with a big-endian long integer giving the size of the packet (the number of following data bytes). However, and contrary to SnapGene packets as described above, there is no type tag indicating the type of the packet and the nature of its contents. The type of a packet is solely indicated by the packet's position in the overall structure of the Gck file.

All multi-byte numerical values use big-endian order.

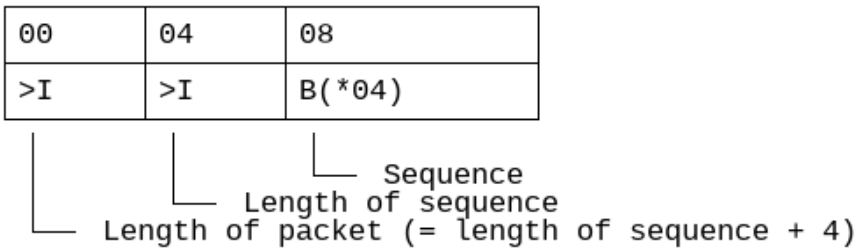
The following sections describe the different components of a Gck file, in the order in which they appear.

4.1. The Header and the Sequence Packet

A Gck file starts with a fixed-size header of 24 bytes. I could not identify any field within that header. Bytes 4 and 8 seem to always contain the value 0x0C, which *may* act as a magic cookie.

Immediately after the header comes the Sequence Packet (Figure 8).

Figure 8. Structure of the Gck sequence packet



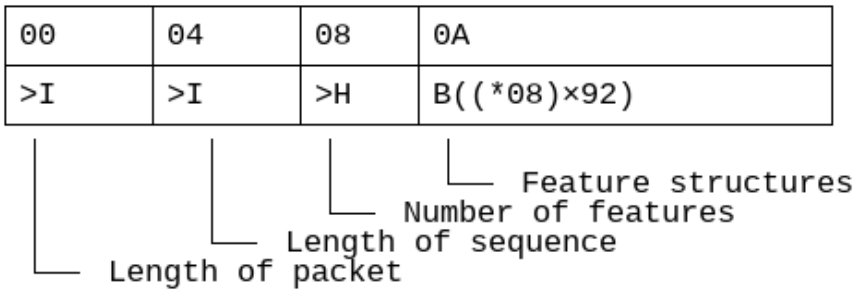
The Sequence Packet, as a normal length-prefixed packet, starts with a long integer giving the size of the packet. Then comes another long integer giving the length of the sequence, and then the sequence itself, in as many bytes as indicated by the previous integer.

The Sequence packet is followed by a packet whose contents is unknown. Being a proper length-prefixed packet, it can however be skipped easily by reading the number of bytes indicated in the first 4 bytes.

4.2. The Features Packet and Associated Strings

Sequence features are described in two consecutive parts of a Gck file: first a Features Packet (Figure 9), then a list of strings associated with the features (thereafter referred to as the Feature Strings List).

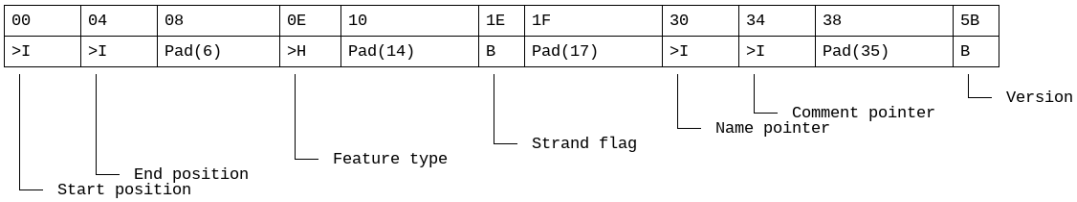
Figure 9. Structure of the Gck features packet



The data bytes of the packet start with a long integer giving the length of the sequence (which is the same as the length indicated at the beginning of the Sequence Packet, I don't know why that information is repeated here), followed by a short integer giving the number of features.

Each feature is then described by a 92-byte block (Figure 10).

Figure 10. Structure of a Gck feature



Two long integers in bytes 1–4 and 5–8 give the 1-based coordinates of the feature. The strand flag in byte 31 can be 0 (no strand specified), 1 (feature on the reverse strand), 2 (feature on the forward strand), or 3 (feature on both strands).

A short integer in bytes 15–16 supposedly indicates the feature's type. This field can take a large range of values, but in all the Gck files I have seen there was actually only two possibilities: a value of zero is a `misc_feature` (which can be anything, really), and any non-zero value denotes a `CDS`.

The last byte of the structure (byte 92) is a version number. It indicates that the feature belongs to the specified version of the file. Versions are numbered in reverse order: the most recent version has the number zero, then the previous version has the number 1, the version before that has the number 2, and so on. (So if you are only interested in the current version of the file, you may skip all features with a version number greater than zero.)

Two long integers, in bytes 48–51 and 52–55, indicate, if they contain a non-zero value, that this feature has respectively a name (stored as a 8-bit Pascal string) and a comment (stored as a 32-bit Pascal string) in the Feature Strings List after the Features Packet.

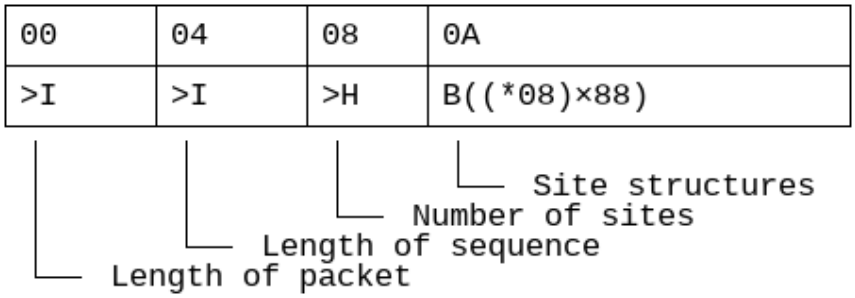
The Feature Strings List has no header, no length indicator, or any marker delimiting the beginning or end of the section. To parse it, or even just to skip it, one needs to know how many strings to expect (and whether they are 8-bit P-strings or 32-bit P-strings), by first parsing the Features Packet and looking at which features have non-zero name or

comment pointers. The strings are stored in the same order as the features in the Features Packet, meaning there is first the name of the first feature (if any), then the comment of the first feature (if any), then the name of the second feature, and so on.

4.3. The Sites Packet and Associated Strings

Restriction sites are described in a similar way as the features, with a Sites Packet (Figure 11) listing the sites, followed by a list of strings (the Site Strings List).

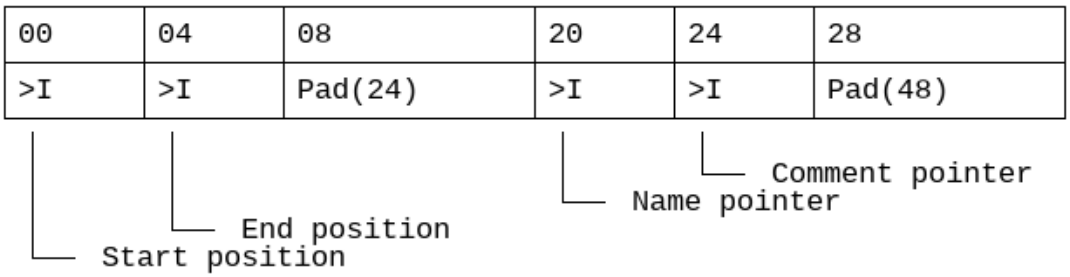
Figure 11. Structure of a Gck sites packet



As for the Features Packet, the data bytes start of the Sites Packet start with a long integer repeating again the length of the sequence, followed by a short integer giving the number of sites.

Each site is then described by a 88-byte block (Figure 12).

Figure 12. Structure of a Gck site



Two long integers in bytes 1–4 and 5–8 give the 1-based coordinates of the restriction sites. Two more long integers, in bytes 32–35 and 36–39, indicate, if they contain a non-zero value, the presence of an associated string in the following Site Strings List.

As for the features, names and comments are stored as 8-bit and 32-bit P-strings, respectively, in the same order as the order of site structures in the Sites Packet. And as for the features, the list of site strings can only be parsed by knowing the strings to expect, by parsing first the Sites Packet.

4.4. The Versions Section

A Gck file can store a kind of history of itself, by keeping track of different versions. Such versions are described in a Versions pseudo-packet followed by a list of associated strings.

The Versions pseudo-packet is not a proper length-prefixed packet. Instead of a long integer giving the size of the entire packet, it starts with a short integer giving only the number of versions, followed by as many 260-byte structures as there are versions.

I could not find any meaning to the contents of a Version structure, save for the last 4 bytes (257–260) which indicate, if they contain a non-zero value, that the version has an associated comment in the following Versions Strings List.

If a comment is present, then it is stored as a 32-bits Pascal string. As for the feature and site strings, the Versions Strings List requires to know how many strings to expect, by parsing first the version structures.

4.5. The Name and Flags Section

After the Version Strings List, there is a fixed-size block of 706 bytes. I have no idea about the contents of this block. It has no length indicator or terminator marker.

That unknown block is followed by a single 8-bit Pascal string containing a free-form name for the sequence described in the file.

Finally, after the name string is a 17-byte block presumably containing flags. I could only make sense of the last byte of the block, which if non-zero indicates that the sequence is circular (it is linear if the flag is unset).