

27th February 2008

Generic Format for Sequence Data

Version 1.3.1

Contributions

This standard was developed using an open process. An archive of meeting minutes and discussions is available [here](#):

The major contributors are (in no particular order):

- James Bonfield (Sanger)
- Gabor Marth (Boston College)
- Toby Bloom (Broad)
- Martin Shumway (NCBI)
- Vladimir Alekseyev (NCBI)
- Paul Flicek (EBI)
- Darren Platt (23andme)
- Mike Attili (Helicos)
- James Knight (Roche)
- Clive Brown (Sanger)
- Dirk Evers (Illumina)
- Jonathan Manning (ABI)
- Asim Siddiqui (BC Genome Sciences Centre)

Background

This document describes a machine and technology independent format for storing DNA sequence data and associated quality values. This document is **preliminary** and **subject to change**.

Contacts

Change history

The following table shows the change history for this format.

Version	Date	Author	Comments
0.1	15 Oct 2006		Original content.
0.1.5	29 th Oct 2006		
0.2	13 th Nov 2006		Modified per comments received at Nov 2 nd telecon.
0.2.1	26 th Nov 2006		Modified per comments received by e-mail. Incorporated Read Header and ZTR format.
0.3	11 th Dec 2006		Modified per comments received at Dec 7 th telecom
1.0 DRAFT	9 th Jan 2007		Version 1.0 for approval
1.0 DRAFT	28 th Jan 2007		Version 1.0 revised for approval
1.0	11 th Feb 2007		Approved version
1.1.1	11 th Jul 2007		First draft of next version. Changes to container structure, indexing and read header
1.1.2	29 th Jul 2007		Minor changes following comments received at the NCBI meeting. Changes made to readFlags in read header, XML block, data header block, and index block. Some additional cosmetic changes.
1.2	6 th August 2007		Version 1.2
1.3	18 th Dec 2007		Changes to read id format and minor changes to index format
1.3.1	6 th Feb 2008		Description of containers and multiple files. Acknowledgements added.

Table of contents

1. Overview	7
1.1 Scope and purpose	7
2. References	8
3. Terms, definitions, and notation	9
3.1 Conformance levels	9
3.2 Glossary of terms	9
4. Abbreviations and acronyms	10
5. Requirements	11
6. Format Specification	12
6.1 General	12
6.1.1 Strings	13
6.2 Container Header	14
6.3 XML Block	14
6.4 Data Block Header	14
6.5 Data Block	15
6.5.1 Read Header	15
6.5.2 ZTR Blob	15
6.5.3 Unique Read Id	16
6.5.4 Defining Read Pairs and Other Read Features	18
6.6 Index Block	19
6.6.1 Hash Function	20

List of figures

Figure 6.1—A single container	12
Figure 6.2—Multiple containers	13

List of tables

Table 5.1—Format Requirements	11
Table 6.1—Block format.....	13
Table 6.2—Container Header Format	14
Table 6.3—XML Block Format	14
Table 6.4—Data Block Header Format.....	15
Table 6.5—Read Header Format.....	15
Table 6.6— Read Identifier formatting characters	16
Table 6.6— Unique Read Identifier	17
Table 6.7—Index Block Format.....	19

1. Overview

1.1 Scope and purpose

Scope: This document describes a format for storing nucleic acid sequence information. The format is designed to be machine and technology independent.

Purpose: The format has been developed to provide a single, uniform format for DNA sequence data. The format has primarily been designed to support data archival, data exchange. Its secondary purpose is to support data submission to the NCBI Short Read Archive.

2. References

- [R1] IUPAC Nucleotide Code - IUB (Nomenclature Committee, 1985, Eur. J. Biochem. 150; 1-5).
- [R2] Brent Ewing, LaDeana Hillier, Michael C. Wendl, and Phil Green. Base-calling of automated sequencer traces using PHRED. I. Accuracy assessment. 1998. Genome Research 8:175-185.
- [R3] Brent Ewing and Phil Green Base-calling of automated sequencer traces using PHRED. II. Error probabilities. 1998. Genome Research 8:186-194
- [R4] Extensible Markup Language (XML) 1.0 (Fourth Edition) W3C Recommendation 16 August 2006, Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, François Yergeau eds.
- [R5] ZTR: a new format for DNA sequence trace data . James K. Bonfield and Rodger Staden. Bioinformatics 18:3-10

3. Terms, definitions, and notation

3.1 Conformance levels

Several keywords are used to differentiate between different levels of requirements and optionality, as follows:

3.1.1 expected: Describe the behavior of the hardware or software in the design models assumed by this specification. Other hardware and software design models may also be implemented.

3.1.2 may: Indicates a course of action permissible within the limits of the standard with no implied preference (“may” means “is permitted to”).

3.1.3 shall: Indicates mandatory requirements strictly to be followed in order to conform to the standard and from which no deviation is permitted (“shall” means “is required to”).

3.1.4 should: An indication that among several possibilities, one is recommended as particularly suitable, without mentioning or excluding others; or that a certain course of action is preferred but not necessarily required; or that (in the negative form) a certain course of action is deprecated but not prohibited (“should” means “is recommended to”).

3.2 Glossary of terms

3.2.1 byte: Eight bits of data, used as a synonym for octet.

3.2.2 Base caller: A program used to identify the bases in a sequence read from the data generated by a sequencing machine.

3.2.3 Big endian: Indicates that the byte order for an integer stored in multiple bytes is most significant byte first.

3.2.4 Read: A sequence of nucleic acids. Usually refers to a sequence generated by a sequencing machine.

3.2.5 Flow traces: Sequence read information generated by certain types of sequencing machines (for example the Roche/454 FLX sequencer).

3.2.6 ZTR: A block based format originally developed to store ABI trace files [R-5].

4. Abbreviations and acronyms

This document contains the following abbreviations and acronyms:

CH	Container Header
DB	Data Block
DBH	Data Block Header
DNA	Deoxyribose Nucleic Acid
IB	Index Block
ID	identifier
NCBI	National Centre for Biotechnology Information
PHRED	
RH	Read Header
URL	Uniform Resource Locator
UTF-8	8-bit Universal Character Set/Unicode Transformation Format
XML	eXtensible Markup Language

5. Requirements

The standard has been developed to fulfill the following requirements

Table 5.1—Format Requirements

Requirement Id	Requirement	Reference
RQ-1	The standard shall be open.	
RQ-2	The standard shall have a streamable format	
RQ-3	The standard should allow data to be stored in an efficient manner (i.e. consume the least amount of disk space).	
RQ-3	The standard shall support random access to the data file.	
RQ-4	The standard shall not require experimental information.	
RQ-5	The standard shall support individual reads and sets of multiple reads.	
RQ-6	The standard will support a unique identifier for each read.	
RQ-7	Multi-byte order shall be big endian.	
RQ-8	The standard shall support the storage of reads from different platforms in the same file.	
RQ-9	The standard shall not require image data (and their equivalent) to be stored.	

6. Format Specification

6.1 General

A data file is comprised of one or more containers. Each container is comprised of several blocks.

Container Header (CH) – The first block in a container is the Container Header. There is only one such block per container and it contains general information about the container.

XML Block – The CH may be followed by an XML (R-4).

Data Block Header (DBH) – The Data Block Header is followed by one or more Data Blocks. It contains information common to the Data Blocks that follow it.

Data Block (DB) – The Data Blocks contain the actual sequence data and associated quality values. Each Data Block contains information about a single read. Data Blocks utilize the ZTR format (R-5) and comprise a Read Header (RH) and a ZTR blob.

The last block in a file is the optional index block.

Index Block (IB) – The index block is an optional block that contains a hash-based index. The index enables a fast lookup of the location of every read in every container in the file. The format of the index block allows it to be stored in a separate file to the containers.

The last 8 bytes in the file are used to store the size of the index block (in bytes). If the index block is not present, the size of the index is given as zero bytes.

The following figure provides an example of a file containing a single container.

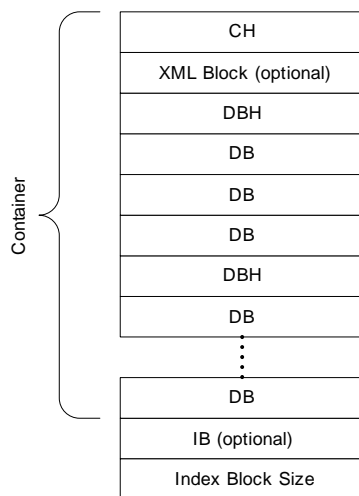
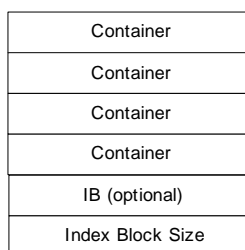


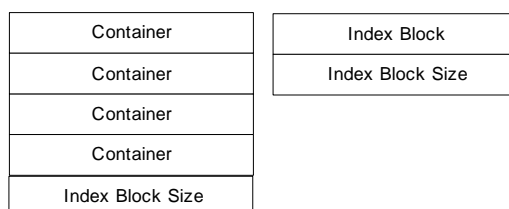
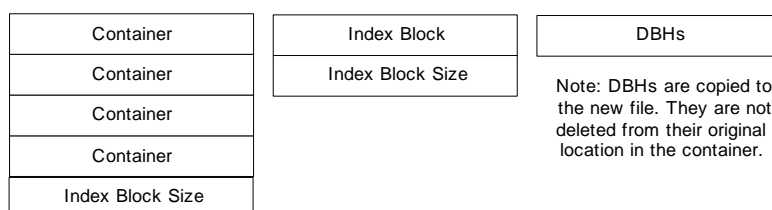
Figure 6.1—A single container

A single file may contain several container structures. Each container is independent and thus containers can be added or removed from the file without disrupting the integrity of the file.

The following figure provides an example of a file containing several containers.

**Figure 6.2—Multiple containers**

The index block may be stored in a separate file to the CH and DB blocks. The DBH blocks may be in the same file as the CH and DB blocks or in a separate file. Note: If the DBH blocks are in a separate file, they are not removed from the container i.e. they are present in both files. Hence other possible configurations include:

**Figure 6.3—Index Block in a separate file****Figure 6.4—Index Block and DBHs in separate files.**

With the sole exception of the index block and container header, all blocks begin with a single character block type and a block size.

Table 6.1—Block format

Field	Description	Type	Value
blockType	The type of block.	char[1]	
blockSize	The size of the entire block (including the blockType and blockSize fields)	unit 32	
Data	Block data		

6.1.1 Strings

Strings are specified either as fixed length (length indicated in square brackets) or as variable length (indicated by an asterix, '*'). For variable length strings, the first byte contains an unsigned int that indicates the number of characters in the string and hence the number of bytes that follow the unsigned int. If a string is optional and no value is to be stored, the length is specified as zero.

6.2 Container Header

The Container Header contains general information about the file. The header includes information about the basecaller used to create call bases of the read. A consequence of this aspect of the format is that a single container may only contain reads called by the same base caller and hence all the reads will (likely) be generated by a single technology. It is not anticipated that the same read will be repeated in multiple containers called by different base callers. Since a single file may contain multiple containers, this limitation on containers is not a great encumbrance to the format.

Table 6.2—Container Header Format

Field	Description	Type	Value
blockType	The type of block.	char[4]	SSRF
blockSize	The size of the entire block (including the blockType and blockSize fields)	unit 32	
version	The version of the standard. Note: the 3 rd number in the version of this document denotes a change to the document that does not change the format itself. Therefore, this field is only comprised of the first two numbers of the document version.	char *	1.3
ContainerType	The type of blob stored in this container. The only supported value is 'Z' for ZTR.	char[1]	'Z'
baseCaller	The name of the base caller used to call the read.	char *	
baseCallerVersion	The version of the base caller	char *	

6.3 XML Block

The XML Block is optional. If present, it is block of UTF-8 characters in XML format (R-4).

Table 6.3—XML Block Format

Field	Description	Type	Value
blockType	The type of block.	char[1]	'X'
blockSize	The size of this block	uint 32	
XML	XML text. The XML record may be used to store additional user defined data. The record "NCBI_submission" is reserved and its form defined by NCBI.	*	

6.4 Data Block Header

The Data Block Header contains information common to the next series of Data Blocks.

Table 6.4—Data Block Header Format

Field	Description	Type	Value
blockType	The type of block.	char[1]	H
blockSize	The size of this block.	uint 32	
subBlockType	Field reserved to allow different types of Data Block Headers in the future. The only type of data block available in the current format is an “E” block or explicit block.	char[1]	E
uniqueIdPrefix	All reads in the data block header have the same unique id prefix. The prefix is a variable length string and may be zero bytes. Refer to section 6.5.3 for more details.	char *	
headerBlob	The header blob. The size of the header blob is deduced from the block size.	*	

6.5 Data Block

Each data blocks is comprised of a Read Header and followed by a ZTR blob.

6.5.1 Read Header

Table 6.5—Read Header Format

Field	Description	Type	Value
blockType	The type of block.	char[1]	R
blockSize	The size of the block	unit 32	
readFlags	Each bit may be set to indicate a status for the read. The bits are set to 1 if the flag is true. Bit 1: Is the read bad? Bit 2: Has the read been withdrawn? Bits 3-5: reserved Bits 6-8: user definable	1 byte	\000
readId	The readId is a variable length string. When combined with the uniqueIdPrefix, it yields the unique read id.	char *	
dataBlob	The data blob. The size of this field is deduced from the blockSize.	*	

6.5.2 ZTR Blob

The ZTR Blob is comprised of several ZTR chunks. For details of the ZTR format refer to http://staden.sourceforge.net/manual/formats_unix_12.html. This format support ZTR version 1.3.

MANUFACTURER PARAGRAPHS – VENDORS to provide details on how their data will be organized in ZTR.

6.5.3 Unique Read Id

Each read is assigned a unique read identifier. The unique read id may be globally unique or locally unique. Locally unique ids are unique within a single file, whereas globally unique ids are unique everywhere. The generator of the file is responsible for ensuring read id uniqueness.

When creating the container, the unique read is split into two parts: the unique identifier prefix in the Data Block Header and the readId in the Read Header.

The read id is optional, but its absence prevents the container from being indexed. Local read ids may be utilized to index a file that has no read ids.

The read name is constructed by using a common name prefix stored in the Data Block Header (uniqueIdPrefix) and a suffix stored per-trace in the Read Header (readId). Both of these are strings in the same format used elsewhere; the first byte indicating their length.

The simplest form of trace name is a direct concatenation of the two strings. This is used whenever the uniqueIdPrefix does not contain a '%' character.

To save space it is possible to encode the Read Header name component in binary and specify how this should be decoded by embedded percent expansion rules within the uniqueIdPrefix. The percent expansion rules take the following form:

%<field-width>.<bits-used><format>

"<field-width>" is the minimum number of characters used to format this piece of data. It is optional and defaults to 1. The characters used to pad out a format value that is too small will vary depending on the format used. See below for details.

".<bits-used>" indicates how many bits to use from the readId, starting from bit 7 (and decreasing from there) or the next free bit following a previous %format. It is optional and if not specified generally all bits are used unless indicated otherwise below.

"<format>" controls how the bits are formatted to generate the trace name. It must be one of the following:

Table 6.6— Read Identifier formatting characters

Formatting character	Description
%d	Decimal number. When field-width is longer than the number it is padded out with preceding zeros ('0').
%o	Octal number. The padding character is '0'.
%x	Hexadecimal lowercase; 0-9 and a-f. The padding character is '0'.
%X	Hexadecimal uppercase; 0-9 and A-F. The padding character is '0'.
%j	Base-36 encoding a-z and 0-9 in that order. The padding character is 'a'.
%J	Base-36 encoding A-Z and 0-9 in that order. The padding character is 'A'.
%c	ASCII character. A single character of <bits-used> bits or 8 if no <bits-used> is specified. The field-width value is ignored.

Formatting character	Description
%s	A string - essentially repeated '%c' until all bits are consumed. The field-width value is ignored. Note that a single %s on the end of the uniqueIdPrefix is equivalent to the default concatenation rule when no percent-expansion is found.
%	A literal percent sign. No bits of readId are consumed. The field-width value is ignored. Note: a decoded read id may only contain the characters A-Za-z0-9 and underscore “_”. Hence this coding should not be used.

Note that in the above if the number of bits is specified as more than 32 then it is treated a series of 32-bit blocks. This has no impact on octal and hexadecimal encodings, but may give unexpected results with other formats.

For example, to store a trace name of the format "run_lane_tile_x_y" with x and y being two values between 0-1023 inclusive we can observe that the coordinates need only 12-bits each, thus the pair of them fit in 24-bits, or 3 bytes (plus 1 for the string length). Our uniqueIdPrefix could then be (27) "run_lane_tile_%3.12X_%3.12X". If the X and Y coordinates are 999 and 196 then the readId would consist of hex 03 3e 70 c4 and together this would generate a string for the name as "run_lane_tile_3E7_0C4".

The read id may contain format characters identified by % (similar to printf statements). When these format characters are present in the prefix, the readId in the RH is interpreted according to the formatting characters.

The following formats for identifiers utilize the type “String”. When these identifiers are encoded into the DBH and DB, they are encoded as type char *.

When decoded, the unique read identifiers adhere to the following form:

Table 6.7— Unique Read Identifier

Field	Description	Type	Value
namespace	Namespace to avoid clashes.	string	
read	Alphanumeric string. Underscores permitted. Allowed character set [A-Za-z0-9_]	string	

6.5.3.1 Preassigned namespaces

Namespaces ‘T’, ‘X’ and ‘L’ are reserved for local read ids.

Vendors are assigned name spaces as follows.

VRO – 454/Roche

VAB – ABI

VHE - Helicos

Sequencing centres wishing to distribute global read ids may do so using their NCBI registered trace repository identifier, prefixed with an ‘N’.

6.5.4 Defining Read Pairs and Other Read Features

A single “read” may be comprised of multiple reads, read pairs or other features (e.g. primers). This information is encoded in the ZTR chunks. The data block header ZTR blob contains a TEXT chunk named REGION_LIST that describes the components of the reads that follow. Hence, the same REGION_LIST applies to all reads that follow. The read header ZTR blob contains a REGN chunk that describes the boundaries between each component of the read. For more details refer to the ZTR specification.

6.6 Index Block

The Index Block provides a lookup hash for every read in the block. Once a DB for a read is identified, the corresponding DBH is the one that precedes the location of the DB in the container.

Table 6.8—Index Block Format

Field	Description	Type	Value
blockType	The type of block.	char[4]	I
version	The version of the index.	char [4]	1.00
indexSize	The size of the index	uint 64	
IndexType	The type of index. This index supports explicit DBHs only.	char[1]	'E'
dbhPositionsStoredSeparately	This byte indicates whether the read entry contains a field indicating the DBH associated with the read. This field is only required if the DBHs are stored in a separate file or if the reads are re-ordered to optimize bulk data reads.	char[1]	0 or 1
numberOfContainers	The number of containers in the file.	uint 32	
numberOfDBHs	The number of DBHs in the file.	uint 32	
numberOfHashBuckets	The number of hash buckets in the file.	uint 64	
dbhFile	If the DBHs are stored in a separate file, the field contains the name of this file.	char *	
containerFile	If the container and reads are stored in a separate file, the field contains the name of this file. The dbhFile and containerFile can be the same file.	char *	
List (n=numberOfContainers)			
bytesToContainer	The number of bytes from the start of the file to the container	uint 64	
End of List			
List (n=numberOfDBHs)			
bytesToDBH	The number of bytes from the start of the file to a DBH	uint 64	
End of List			
List (n=numberOfHashBuckets)			
bucketOffset	Offset relative to the start of the index for the item linked list.	uint 64	
End of List			
List (n=number of reads in the file)			
entryByte	Bits 0-6: Name disambiguation hash. If bit 1 is set, these bits are set to zero, otherwise, the bits are a second short hash. Bit 7: Linked list end. This bit is set to 1 if there are no more entries in the present bucket.	1 byte	
readHeaderPosition	The location of the read header in the file (offset from the start of the file).	uint 64	
dbhIndex	This field is only present if dbhPositionsStored is set to '1'. This field stores the index of the DBH associated with this	uint 32	

Field	Description	Type	Value
	read in the DBH list.		
End of List			
blockType	The type of block.	char[4]	I
version	The version of the index.	char [4]	1.00
indexSize	The size of the index	uint 64	

6.6.1 Hash Function

The top-7 bits are used as a disambiguation hash instead of duplicating the trace name as a space-saving measure. The hash function implemented is lookup3.c from <http://burtleburtle.net/bob/hash/doobs.html>. The number of buckets MUST be a power of two. The disambiguation hash is the top 7 bits from the 64-bit lookup3 hash value. Ie:

```

hval = hash64(HASH_FUNC_JENKINS3, (unsigned char *)tname, strlen(tname));
...
/* Secondary hash is the top 7-bits */
hval >>= 57;

if ((disambig & 0x7f) == hval) {
    /* Potential hit */
    ...

```