CALF (Compact ALignment Format)

Version 0.081113

(Comments & suggestions appreciated – please send to Phil Green <phg@u.washington.edu>).

Main features:

(i) In addition to the read-to-reference (or read-to-consensus) multiple sequence alignments (including strand, mate pair and RNAseq spliced alignment information), a CALF file records the base qualities and mapping qualities of the aligned reads, and unaligned read data. In particular all of the information (except prior probabilities) needed for SNP detection and computation of genotype probabilities at each reference position is present. Since the reads & base qualities can readily be recovered from the CALF file, arguably it could be the only file one needs to keep following a next-gen sequencer run.

(ii) It is compact, requiring only about $2 + c$ bytes per reference sequence position, where c is the average read depth. For $c > 2$ this is actually smaller than a FASTQ file of the reads (which is possible by virtue of encoding a read base and its quality value in a single byte), and for $c < 10$ it is smaller than a GLF file constructed from the same data.

(iii) The context-dependencies in the CALF read & alignment representations are all local in the sense that, starting from a particular alignment column, one can reconstruct the part of the alignment surrounding that column, including the mate-pair and spliced alignment connections and all base and mapping qualities, by reading relatively short distances forwards and backwards in the CALF file. By avoiding the need to read in the entire alignment this allows fast access to regions of interest even with very large files, making large alignments such as deeply resequenced human genomes manageable. Paired mates and spliced alignments are pointed to using byte offsets, making it possible to jump to the relevant location in the file (using *fseek*, for example) without reading the intervening region.

Conceptually, the format is straightforward: Think of an ordinary text representation of a multiple alignment, with all sequences written horizontally, and the reference as the first row. Subsequent rows contain the reads in their aligned locations, with many reads (separated by spaces) generally present in each row. Both the reference and the reads can contain gap characters. We convert each alignment column to a (variable-length) 'record' by stripping out spaces and using one byte to represent each alignment character (base or gap), together with its quality value, that occurs in the column. A 0 byte terminates the record. Four additional bytes per read are required to indicate when a read alignment is starting or ending and to record the strand and mapping quality of the alignment. (The start and end markers allow unambiguous reconstruction of the alignment from its 'stripped out' reduction). When there is a mate pair or spliced alignment continuation, additional bytes (usually two or four per read) are needed to point to it.

Detailed specification:

A CALF file consists of an initial free-form ASCII (plain text) section, and a data section which may include a record-based representation of one or more multiple alignments and unaligned reads.

1. ASCII section

This can have an arbitrary number of (\n-terminated) lines, and it is terminated by a 0 byte (which must appear even if there is no ASCII text). 0 bytes are not allowed within the ASCII text. Normally this section would specify (at least) the sample from which the reads were generated (including whether it is cDNA or genomic DNA), library characteristics (e.g. insert size distributions used for assessing mate-pair consistency), the names & versions of the reference sequences, and the program used to produce the alignments. Some groups will want to impose a specific format, but it is probably not desirable to make this part of the specification because relevant information will vary by project.

2. Data section

This immediately follows the 0 byte that terminates the ASCII section, and consists of a series of variable-length 'records' representing one or more multiple alignments, each record corresponding either to a single alignment column, or to an uncovered segment of the reference (i.e. a block of alignment columns in which the reference but no aligned reads occur). Uncovered segments can be represented in either of two ways (size only, or packed sequence). Records are byte structured, i.e. parsing of the record occurs byte by byte (although several types of information can be present within a single byte). Each record is terminated by a 0 byte. Each alignment is assumed to correspond to a single contiguous reference sequence. To include more than one alignment (e.g. for the different chromosomes in a eukaryote genome), simply concatenate them. As described in section 2.6 below, unaligned read data are included within alignment records when the read has an aligned mate, and following the last alignment otherwise.

An empty record (consisting only of a 0 byte) must follow the 0 byte that terminates the last record in the last alignment. This empty record must appear even when no alignments are given. Any data that follow it are assumed to represent unaligned reads.

The record byte structures are as follows:

2.1. First byte of a record ('record header byte'). This byte is parsed as (p, s, t) where $0 \le p \le 15$ is the value represented by the four highest-order bits, $0 \le s \le 3$ is the value represented by the next two lower-order bits, and $1 \le t \le 3$ is the value represented by the two lowest-order bits. t indicates the record type (and should never be 0):

type 1: a single alignment column
type 2: an uncovered reference segment for which the size (but not sequence) is given
type 3: an uncovered reference segment for which the sequence (but not size) is given, in packed format.

s indicates the type of the previous record; this information is needed for the purpose of reading backwards in the file. $s = 0$ when the current record is the first in an alignment (indicating the beginning of a new reference sequence).

In a type 1 record, *p* indicates the (possibly ambiguous) reference sequence nucleotide (A=1, C=2, G=4, T=8, & combinations thereof; 0 denotes a gap character in the reference). In a type 2 or 3 record, $p = 0$.

The form of the remaining bytes depends upon the record type, as follows:

2.2. Type 1 record. The bytes following the header byte are (usually) parsed as (n, q), where $0 \le n \le 3$ is the value represented by the two high-order bits, and $0 \le q \le 63$ is the value represented by the remaining six bits. (Bytes immediately following a 'start marker' byte are parsed differently, as described below). The following cases occur (depending on the values of q and n):

$1 \le q \le 61$: the byte represents a base & quality pair, with n denoting the base (A = 0, C = 1, G = 2, T = 3), and q = base quality + 1 (adding 1 ensures the 0 byte cannot occur). Base quality values higher than 60 (which essentially never occur in raw reads anyway) must be truncated to 60. (In principle 0 qualities should never occur either, since they imply complete uncertainty i.e. the base could be represented by N; but we don't enforce this).

$q = 0$:
$n = 0$: '0 byte' (terminates record)
$n = 1$: N (the only ambiguity code we allow in reads)
$n = 2$: '- byte' (gap character; also used as mate separator in unaligned data)
$n = 3$: '* byte' (used to delimit unaligned data (see section 2.6 below))

Bytes as above with $0 \le q \le 61$ are (except for the 0 byte) referred to collectively as 'base/qual bytes'. Note that the N, '-', and '*' bytes have values 64, 128, and 192 respectively and are NOT the ASCII bytes representing those characters.

$q = 62, n \ge 0$: 'start marker' byte, indicating that a new read alignment is starting in this column. ('Start' means the beginning, or left-most end, of the read's alignment, not the 5' end of the read). The start marker byte is followed by (in order):
(i) (Optionally) an ASCII header, consisting of a 0 byte, followed by ASCII text (which may not contain an internal 0 byte), followed by another 0 byte. ASCII text prior to the the first white space character (' ', \n, \t, \r, vertical tab, or formfeed) – or the entire field, if no white space character is present – is presumed to represent the read name. Text following the first white space character is arbitrary and may be used to designate the read library or other

relevant information. The read name may be omitted by placing a white space character immediately following the initial 0 byte. Note that the entire ASCII header is optional, as read names are often unnecessary in next-gen sequencing data and can take up a significant amount of space.

(ii) a 'strand/mapqual' byte in which the high-order bit indicates strand of the alignment (0 = top, 1 = bottom), and the low order seven bits represent 1 + mapping quality of the read ($0 \leq$ mapqual ≤ 100) (Mapping qualities are defined in (Li, Ruan, & Durbin, *Genome Res* 2008))

(iii) $2n$ bytes that point to a spliced alignment or mate pair continuation for this read's alignment, using the format described in section 2.5 below ($n = 0$ means there is no continuation)

(iv) an identical copy of the start marker byte (needed for unambiguous parsing when reading backwards in the file)

(v) * bytes and base/qual bytes representing the unaligned portion of the read (and/or unaligned mate) at this end, if any (see section 2.6)

(vi) the base/qual byte corresponding to the first aligned base of the read

The first of the two start marker bytes is called a 'start/start' marker byte.

***q* = 63, *n* > 0:** 'start marker' byte. As above, except that $4n$ rather than $2n$ bytes are reserved for the continuation pointer information, with the extra $2n$ bytes used to represent the offset in reference sequence co-ordinates.

***q* = 63, *n* = 0:** 'end marker' byte, indicating that the preceding byte is a base/qual byte corresponding to the last aligned base of a read. If there is an unaligned segment of the read (and/or an unaligned mate) at this end (section 2.6), it is put between the end marker byte and the last aligned base/qual byte.

2.3. Type 2 record.

There are four bytes following the header byte, giving the size of the uncovered segment as an unsigned positive integer in big-endian format (to ensure cross-platform compatibility). (Size 0 segments are not permitted).

2.4. Type 3 record.

The bytes following the header byte give a packed representation (allowing ambiguous nucleotides) of the uncovered segment's sequence, with each byte representing a successive dinucleotide: bytes are parsed as (p, r) with $1 \leq p \leq 15$ representing the first base (using the same conventions as for a type 1 record header byte) and $0 \leq r \leq 15$ the second base; $r = 0$ only in the last byte for a region that has an odd number of bases. The segment must have at least one base.

2.5. Continuation pointers.

The $2n$ bytes used to represent the continuation pointer following a strand/mapqual byte are parsed as follows: In the first byte, the four highest order bits are parsed as (a, b, c) where $0 \leq a \leq 3$ (encoded by the two highest-order bits) indicates the type of continuation: $a = 0$ means spliced alignment, while $a > 0$ means mate pair continuation with the value of a roughly indicating library insert size (1 = small insert, 2 = medium, 3 = large); $b = 0$ if the continuation has orientation and spacing consistent with library expectations, and = 1 otherwise; and *c* indicates which end of the current read's alignment the continuation is from: 0 = start, 1 = end.

The remaining 16*n* - 4 bits of the 2*n* bytes represent a signed integer giving the byte offset to that continuation, computed as the byte location of the start/start marker byte of the continuation, minus the current start/start byte location; thus upstream continuations have negative offsets while downstream continuations have positive offsets. To ensure platform independence and simplify parsing, integers are represented as follows: the highest-order bit (immediately following the '*c*' bit above) is the sign bit (1 for a negative integer and 0 otherwise), while the remaining 16*n* – 5 bits represent the absolute value of the offset, in big-endian format. (Note that for negative integers this is NOT the commonly used 2's complement convention.) Continuations may go into a different alignment (chromosome), or to an unaligned read. In the latter case, the data for the unaligned mate should be included at the appropriate end of the aligned read as described in section 2.6 below, and the byte offset value should be set to 0.

Since up to 44 bits are available to represent the offset, multi-terabyte files can be accommodated.

For start markers of type ***q* = 63**, ***n* > 0** the above 2*n* bytes are immediately followed by another 2*n* bytes representing a signed integer (using the above convention) that gives the offset in reference sequence coordinates rather than bytes. (There are no special bits in these bytes.) Reference coordinates follow the convention defined in section 2.8 below.

2.6. Unaligned reads and read segments. Unaligned segments at the beginning or end of a read may be included by putting a byte sequence that consists of a * byte, followed by the base/qual bytes for the segment, followed by another * byte, just after the copy of the start marker byte, or just before the end marker byte (as appropriate) for the read. Data for the unaligned mate of an aligned read is also included in this way, using 1, 2, or 3 '-' bytes at the appropriate end of the unaligned read's sequence (see below) to designate it as such. One can include both an unaligned portion of the aligned read, and an entire unaligned mate, at the same end of the aligned portion of the read.

Unaligned reads not mated to an aligned read are placed after the empty record that follows the last alignment (these reads may not be pointed to by any aligned read). Such data are given as a sequence of *r* base/qual bytes, where *r* is the read length, followed by a 0 byte. Optionally, an ASCII header may be included by placing the following immediately in front of the *r* base/qual bytes: a start-marker byte (with *n* = 0), followed by a 0 byte, followed by ASCII text, followed by another 0 byte, followed by another start-marker byte. (Note that start-marker bytes are needed only when an ASCII header is included, and that end-marker bytes, mapqual bytes, and continuation pointer bytes are never used.) A pair of unaligned mates are represented as a single sequence in which the (usually uncomplemented) first read is followed by one, two or three '-' bytes (to indicate library insert size category), followed by the (usually reverse complemented) mate read. The '-' bytes thus implicitly represent the (unknown) sequence between the two mates. The 0 byte that terminates one unaligned read's (or read-pair's) data is followed immediately by the data for the next unaligned read.

2.7. Tags. A base of the reference may be 'tagged' by including a 0-length read – i.e. one for which the end marker byte immediately follows the start marker byte copy, and there are no aligned base/qual bytes – at the corresponding location. The text of the tag is placed into an ASCII header field for the read. Similarly a segment of the reference may be tagged by including a mate pair of 0-length reads that flank the segment.

2.8. Reference sequence coordinates. To accommodate offsets between distinct reference sequences while also allowing for gaps in the reference, we adopt the following coordinate convention: bases of the 1st reference sequence (of length *a* bases) have coordinates 1 through *a*, bases of the 2d reference sequence (of length *b*) have coordinates $a + 2$ (rather than $a + 1$) through $a + b + 1$, etc. Gap characters within a reference sequence are assigned the coordinate of the nearest reference base to the left. Gaps past the right end of a reference sequence are assigned the coordinate of the last reference base, while gaps preceding the left end of a reference sequence are assigned a coordinate one less than the first reference sequence base. (Such gaps occur when an aligned read extends beyond the reference sequence to which it is aligned. The reason for 'skipping' a base coordinate position between two successive reference sequences is to allow gaps at the right end of the first reference to have distinct coordinates from those at the left end of the second reference).

Notes:

(1) Indexing into a CALF file may be needed for rapidly accessing particular sites in the reference sequences or reads. (Note however that sequences, quality values, pointers to mate pairs, and read names or information about particular reads need not be stored separately, as they all can be included in the CALF file; and that ASCII headers, and tags (2.7 above), are flexible enough to accommodate most internal labeling requirements.) We have not yet specified a standard format for index files, but some reasonable principles are

(i) indices should refer to CALF file locations by (absolute) byte number within the file;

(ii) in addition to the byte number, the index should also give the (absolute) reference sequence co-ordinate (using 1 for the first base in the first reference sequence, and 0 for unaligned reads);

(iii) the only types of file bytes that should be referenced are record header or terminator bytes, base/qual bytes not lying within unaligned reads or read segments, start/start or end marker bytes, and bytes internal to a type 3 (uncovered reference region in packed format) record. As discussed in note (4) below, except for type 3 record bytes all of these byte types are mutually distinguishable from each other by examining their low order 6 bits and the previous byte, so the index only needs to indicate whether or not the byte is a type 3 record byte.

One possible format for an index file is an ASCII file in which successive lines give a byte location (as ASCII digits, and for type 3 record bytes only, a suffix _0 or _1 indicating the low order four bits or the high order four bits respectively – the suffix has the dual purpose of indicating that it is a type 3 record byte and specifying the precise reference sequence nucleotide), followed by a reference sequence co-ordinate, optionally followed by a feature (e.g. start of a new chromosome) and any ancillary information regarding the feature.

(2) CALF files consisting solely of a reference sequence in packed format, or solely of unaligned reads, are consistent with the specification. For example, the latter case would be indicated by having two consecutive 0 bytes between the initial ASCII text and the unaligned read data.

(3) Extraneous gap characters should always be deleted from uncovered regions of the reference before conversion to CALF format, to ensure that 0 bytes do not occur within type 3 records and that the region size in a type 2 record is unambiguous. Note that very small uncovered regions may be more efficiently represented using several type 1 records (each consisting of a header byte + terminator) than type 2 or 3.

(4) File parsing issues: The specification is designed to permit unambiguous parsing of the alignment (or unaligned reads) while reading either forwards or backwards in the file, starting from any byte that heads or terminates a record or that is of base/qual, start/start or end marker type (excluding base/qual bytes within unaligned reads or read segments). The type of any such byte can be determined by virtue of (i) the value q of the low-order 6 bits satisfies $q = 62$ or 63 for start and end marker bytes, but $0 \leq q \leq 61$ for record header and base/qual bytes, and (ii) header bytes are always preceded by a 0 byte whereas base/qual bytes outside of unaligned reads or read segments never are. Strand/mapqual bytes, bytes within continuation pointers, bytes within ASCII headers of reads, or interior bytes of type 2 or type 3 records are not internally distinguishable in this manner, but do not cause a problem during forwards or backwards parsing starting from one of the other byte types, because their locations are known by virtue of the surrounding start marker bytes or record type indicators, together with the facts that type 2 records are of fixed size, whereas type 3 records never contain internal 0 bytes. Parsing can also start within a type 3 record provided it is known to be such, using the fact that 0 bytes delimit the record boundaries.

The initial ASCII section is not necessarily internally recognizable but does not need to be, as it is not indexed into from the data section.

Forwards parsing will always succeed all the way to the end of the file, but backwards parsing may only succeed as far as the beginning of the current alignment (because the type of the last record in the preceding alignment is not known). 'Jumping' to another alignment via continuation pointers and further parsing within that alignment always succeeds, in either direction.

(5) Most operations on CALF files are straightforward, and do not require much memory provided one uses an appropriate indexing scheme. As an example, we outline a simple algorithm that can be used to merge several CALF files having the same reference sequence(s) but different sets of aligned reads: The to-be-merged input files are read in parallel record by record while simultaneously writing the new file, keeping in memory essentially just the current merged record as it is being constructed and, for each input file, an index of byte locations of read starts in the input and merged files. At each successive position in the reference sequence,

create an initial merged type 1 (alignment column) record by concatenating the type 1 records at that position from the input files (stripping out the extra copies of the header and terminator bytes), and then modify this record as follows:

(i) for start markers pointing to downstream continuations:

(a) increase the number of allocated continuation pointer bytes if necessary, to be sure they are enough to be able to represent the new pointer value in the merged file once it is known (which won't be until later) – when in doubt, use the maximum of 6 bytes (setting n = 3 in the start-marker byte). Keep all of the other continuation pointer information (including the special bit values, and the numerical value of the byte offset) the same as in the input file.

(b) put into the in-memory index for this file an entry that indicates the byte locations of the start marker in the input file, and in the merged file (which is now known because the merged file is being concurrently written).

(ii) for start markers pointing to upstream continuations:

(a) compute the input file byte location of the upstream continuation start marker (by adding the input file byte location of the current start marker to the input file byte offset) and use this to look up in the index the corresponding location in the merged file;

(b) compute the new (merged file) byte offset, by subtracting the merged file location of the current start from that of the upstream continuation start. If the new offset is different from the old (input file) offset, then put the new offset into the continuation pointer for the current start (using as few bytes as possible), and rewrite the offset value at the upstream start marker in the merged file to be the negative of the new offset (without changing the number of bytes previously allocated as in (i) (a) above, so as not to invalidate intervening byte offsets). Keep other information at the current start marker the same.

(d) delete the index entry for the upstream start marker.

Then write the merged record into the new file.

There are some additional bookkeeping issues related to uncovered region (type 2 and 3) records, which need to be appropriately altered since these regions will differ among the input files, and to the fact that gaps in the reference may differ among the merged files.

Since an index entry is deleted as soon as the downstream mate is reached, which is usually quite soon, in general the indices do not get very large. However, the above strategy can involve a fair amount of non-sequential disk access in order to rewrite the continuation pointer bytes at the upstream mates. To reduce this, one can keep a few thousand records for the growing merged file in memory before writing, updating their pointer information in memory when possible.

Removing duplicates, or making other changes to the alignment or to the file (for example, reducing the number of bytes used for continuation pointers at upstream mates following the initial construction of a CALF file) can be handled using a similar approach – in such cases one would have a single input file and an output file. Note that if only minor changes to the alignment are involved (for example, removal of one read) the entire process will be quite fast because, with the above algorithm, non-sequential disk accesses are required only for those continuation pointers that bridge all or part of the modified region.

One can use a similar indexing approach in alignment programs to create an initial CALF file from an internal representation of the alignments (roughly speaking, this is a little like merging a large number of separate files each with one aligned read or mate pair).

(6) There is some redundancy/inefficiency in the specification, which conceivably could be removed:

(i) In principle one only needs to specify upstream continuations, or only downstream continuations, rather than both, since the other can be inferred. However the cost of this is context-dependence locality – e.g. if only upstream continuations are specified, one might need to read all the way to the end of the file before knowing the location of a read's downstream continuation.

(ii) If one starts from a (conceptual) alignment in which there is only one read per row, then one always knows how many reads are 'active' in a given alignment column, so the need for a separate start byte could be avoided (any unexpected non-zero/non-end-marker byte would imply the beginning of a new read). One could probably also avoid having a second copy of the start marker byte by cleverer encoding. However, this would complicate parsing backwards (and merging), for relatively little savings.

(iii) Instead of using byte locations to define the continuation pointers, one could instead keep a counter value which increases by 1 each time a start/start or end byte is encountered as the alignments and unaligned reads are processed. Thus the value assigned to a start or end occurring within a column C of the alignment would be 2N + M, where N is the number of 'completed' reads (reads for which the read end lies either entirely to the left of C, or within C but above the position being considered), and M is the number of reads which have started but not yet ended. The continuation pointer would then be computed as a difference in counter values rather than byte values. This method has the advantages over byte offsets of being independent of the specific byte representation of the alignments, and of often requiring fewer bytes, but it forfeits the ability to rapidly access the relevant file location without an index.

Comments & suggestions are appreciated – please send to Phil Green at phg@u.washington.edu.
I thank Heng Li and Richard Durbin for comments on an earlier draft.

Version notes:

0.080914: first to be distributed

0.080915: clarifies byte-parsing notation

0.080922: allow placing of unaligned mates near their aligned mates, up to three mate-pair size-insert libraries, refseq co-ordinate offsets.

0.080925: allow inclusion of unaligned parts of read, simpler format for unaligned read data. Algorithm for merging CALF files (in notes).

0.080926: change mate-separator character in unaligned data to '-' (to avoid conflict with delineators of unaligned segments)

0.081003: allow optional ASCII headers for reads

0.081113: clarifies format for representing negative continuation pointers, and that uncovered reference segments cannot have length 0; imposes conventions regarding location of read names within ASCII headers, use of 0 length reads to ‘tag’ sites or segments in reference, and co-ordinate system for reference sequences.