**ASSEMBLER PIPELINE I/O CONVENTIONS**

The assembly group rapidly prototypes pipeline components and then evolves them into robust components, essentially an organic software development strategy. To this end, the various modules can communicate via either a simple ASCII-based encoding of the pipeline messages or a more compact and efficient binary representation under production situations. The idea for the ASCII encoding is that it is easy to read by a human (aiding debugging), but of such a form that it is be trivial to parse.

Note new introduction. Document is now defining document for message passing substrate of entire system.

This document is the **defining document** for the precise information content of every message that flows through the Celera Assembler pipeline**.** As such it contains precise message specification for the input and output of the assembler given in the Celera Assembler document. Often the messages here will contain precise implementations of loosely defined items such as lists in the overview document, whose purpose is only to convey the information content of the external input and outputs. The document describes the messages in order of their introduction along the assembler pipeline.

Note carefully that we adopt the convention that sequence intervals are specified as a pair of positions within a sequence and positions are the points between symbols of the sequence. The leftmost position is numbered 0 so that for example, (0,4) specifies the first 4 symbols of a sequence, (2,2) specifies the position between the second and third symbols, and so on.

# 1. External Inputs:

We start with the simple message for distance and work our way up. The format is called 3-code because every field name and message type name is compressed to a 3 letter abbreviation, with the added convention that type names are all capital-letters and field names are all lower-case letters. A record is encoded across several lines of input where the first line has a '{' in column 1 and the last line consists entirely of a '}' in column 1. The 3-code for the message type name is in columns 2-4 of the header line, followed immediately by a new-line. The lines between the header and tail encode the fields of the record. Each field-line has the 3-code for the field in columns 1-3 and a ':' in column 4, followed immediately by the relevant data in columns 5 to the end-of-line. For the distance message below, the Pascal data structure is given at left and the corresponding 3-code is given at right. The encoding of the data for the field is given by specifying the *scanf* UNIX format that would correctly read the input. Thus, for example, '%ld' reads a long and '[AD]' reads any of the two characters. By convention scalar values are encoded as a single capital letter and the sequence of letters correlates with the sequence of value names in the scalar definition. Thus, for example, 'A' denotes 'Add', and 'D', 'Delete'.

Distance_ID, Fragment_ID, Screen_Item_ID: **int64**

| | |
|---|---|
| DistanceMesg: | {DST |
| **record** | |
| action: **scalar** (AS_ADD,AS_DELETE) | act:[AD] |
| accession: Distance_ID | acc:%ld |
| mean: **float32** | mea:%f |
| stddev: **float32** | std:%f |
| **end** | } |

So an example of a distance message might look like:

```
{DST
act:A
acc:1000803
mea:2000.
std:333.3
}
```

Moving on to the next example message, we consider the screen item record. The new feature illustrated by this record is the way in which arbitrary length strings, needed for the 'Source' and 'Sequence' fields, are encoded. In the translation specification at right for these fields, the symbol '↲' denotes a new-line symbol and we take the liberty of using regular expression syntax to express that the input is a series of new-line terminated strings ending with a line containing a '.' in column 1. The encoded string is the concatenation of all the characters save the new-lines and the terminating period. The relevance field is used to determine which functions should or should not be performed with each screen item or fragments matching the screen item. For ubiquitous repeats, a relevance value of 1 instructs the overlapper to take regions of matching fragments into account when computing overlaps.

| | |
|---|---|
| ScreenItemMesg: | {SCN |
| **record** | |
| type: **scalar** (AS_UBIQREP,AS_CONTAMINANT) | typ:[UC] |
| accession: ScreenItem_ID | acc:%ld |
| repeat_id: [0…Num_of_Repeat_Types-1] | rpt:%d |
| relevance: **int32** | rel:%d |
| source: "description of data source" | src:↲(%[^\n]↲)*. |
| sequence: **string(char)** | seq:↲(%[^\n]↲)*. |
| variation: **float32 in [0,.1]** | var:%f |
| min_length: **int32** | mln:%d |
| **end** | |

So an example of a screen item message might look like:

```
{SCN
typ:C
acc:1099993
rpt:3
rel:0
src:
Anything you'd like to put here is cool!
.
seq:
aaaaaaaaaaaaaaaaaaaaaaaa
ccccccccccccccccccccccc
gggggggggggggggggggggggg
ttttttttttttttttttttttt
.
var:3.2e-2
mln:40
}
```

We conclude with the remaining three primary input types, fragment records, link records, and repeat records, given immediately below. The only remaining matter to discuss is the encoding of quality values in the fragment records. Phred quality values are integers in the range [0,60]. An encoding based on a series of, say blank-separated integer constants, is too space consumptive for even our modest prototyping requirements, so we choose to encode these numbers as a series of printable ASCII characters. To wit, a value *i* is mapped to the ASCII symbol *'0'+i*. Thus a sequence of Phred numbers is mapped to a sequence of characters and encoded in a 3-code record as a string. Note carefully that the string terminating symbol '.' is less than '0' in the ASCII collating sequence and thus cannot occur in the Phred encoding sequence.

| | |
|---|---|
| RepeatItemMesg: | {RPT |
| **record** | |
| repeat_id: [0…Num_of_Repeat_Types-1] | rpt:%d |
| which: **string(char)** | wch:%s |
| length: **int32** | len:%d |
| **end** | } |

The primary input to the assembler is a fragment message, either for a read or a guide. Guides are further categorized as either being (a) BAC-ends, (b) pseudo-reads from unfinished BACs, (c) pseudo reads from finished BACs, and (d) STSs. The difference between the messages for reads versus guides are as follows.

1. Every guide has an associated locale whereas this field is undefined for reads. The interpretation of the locale is different for each kind of guide, but is always expected to be a 64-bit UID produced by the Celera database. For the BAC-based guides it is a UID assigned to the particular BAC from which the guide came. The assembler does not care about the nature of this UID other than that a distinct integer be given to each BAC. If over time, end reads, then contigs, and finally finished sequence for a BAC become available, the same BAC number should be given to the associated guides for that BAC. For an STS guide, a distinct locale number should be assigned to each bin formed when a sufficiently high LOD score is used to order STSs. We suggest 6.
2. Every read has a quality vector and a clear range whereas guides need not. In such a case the quality vector field is NULL and the clear range is the entire fragment. By convention, all fragment sequence coordinate and interval references generated by the assembler are relative to the clear range.
3. Each contig of an unfinished BAC and each finished BAC is assumed to have been partitioned into a set of neatly overlapping pseudo-reads that are given as guides to the assembler. For these pseudo-reads, the assembler will expect to be given the interval of the underlying BAC from which the pseudo-read was excised in the locpos interval of the record. This field is defined only for unfinished and finished BACs. In the case of the several unordered contigs for a given unfinished BAC, simply assign a disjoint interval to each contig and then give the position of each pseudo-read of a contig with respect to the assigned interval.

Note carefully that we adopt the convention that intervals are specified as a pair of positions within a sequence and positions are the points between symbols of the sequence. The leftmost position is numbered 0 so that for example, (0,4) specifies the first 4 symbols of a sequence, (2,2) specifies the position between the second and third symbols, and so on.

After the relevant fragments have been added to the system one may then add (or delete) pairwise distance constraints or *links* between them. A link message always contains the type of link being added or deleted and the two fragments involved. If a link is being added then one also needs to specify the time of entry, a reference to the distance record specifying the distance range between the fragments, and a scalar indicating whether the fragments are in the same, opposite orientation, or unknown orientation. Note that mates and BAC guides are always in the opposite orientation with respect to each other. The distance constraint always refers to the distance between the 5' end of the two fragments, regardless of orientation. Links are divided into six categories according to the source of the link. AS_MATE links are for mated pairs of 2K, 10K, 50K, and transposon library end reads from the Celera sequencing pipeline, from external sources of whole genome shotgun sequence, or from UBAC fragments that were sequences from opposite end of subclones. AS_BAC_GUIDE links are between end-sequenced BACs and AS_STS_GUIDE links are between paired STSs. The AS_REREAD link, permits one to specify that two reads were sequenced from the same end of an insert. These are rereads that were resequenced for some reason, e.g. the PCR-prep encountered a mononucleotide repeat and stuttered, and thus was resequenced with a plasmid prep. In this case neither the distance or orientation fields convey any information. The last two link types model user input constraints and may be between any pair of fragments in the system. The MAY_JOINs represent single links that may be incorporated if there is not conflicting information, and the MUST_JOINs represent infinitely weighted links that will be followed if at all possible. Finally, note fully that there should be exactly one distance record for each insert library, even if the library was designed to have insert sizes equal to that of another library. The reason for this is that the assembler will be empirically determining a distribution of observed mate distances and these distributions will be different, even for libraries designed to have the same mean distance

The distance between guides and mates are specified in distance records that are passed to the assembly system as records requesting that a given distance entity be added or deleted. The message record specifies the action, the ID of the distance entity, and (in the case of insertion) the normal distribution from which the distances were sampled. The field mean gives the mean of the distribution and stddev gives the standard

deviation. Thus, for example, 99% of all links referring to a particular distance message will be of length in `[mean-3stddev,mean+3stddev]`.
The orient field specifies the relative orientation of the two fragments. Links representing sequence of opposite ends of some type of insert (AS_MATE or AS_BAC_GUIDE) must specify an AS_INNIE orientation (3' ends are adjacent) except in the case of mated pairs of reads from a transposon library, which must specify an AS_OUTTIE orientation. Specifying an AS_UNKNOWN orientation is equivalent to specifying 4 links, each with one of the possible orientations.

```
FragMesg:                                          {FRG
record
 action:   scalar (AS_ADD,AS_DELETE)               act:[AD]
 accession: Fragment_ID                            acc:%ld
 variant of action:
  AS_ADD:
   record
      type: scalar (AS_READ,AS_EXTR,AS_TRNR,       typ:[RXTELUFS]
                    AS_EBAC,AS_LBAC,AS_UBAC,
                    AS_FBAC,AS_STS)
    locale:   Locale_ID                            loc:%ld
    locpos:   SeqInterval                          pos:%d,%d
    source:   "description of data source"         src:↲(%[^\n]↲)*.
    entry_time: time_t                             etm:%d
    sequence:  string(char)                        seq:↲(%[^\n]↲)*.
    quality:   string(bytes)                       qlt:↲(%[^\n]↲)*.
    clear_rng:  SeqInterval                        clr:%d,%d
   end
 end
end                                                }

  SeqInterval: record bgn, end: int32 end

LinkMesg:                                          {LKG
record
 action: scalar (AS_ADD,AS_DELETE)                 act:[AD]
 type:   scalar (AS_MATE,AS_BAC_GUIDE,             typ:[MBSRYT]
                 AS_STS_GUIDE, AS_REREAD,
                 AS_MAY_JOIN,AS_MUST_JOIN)
 frag1: Fragment_ID                                fg1:%ld
 frag2: Fragment_ID                                fg2:%ld
 variant of action:
  AS_ADD:
   record
    entry_time:  time_t                            etm:%d
    distance:   Distance_ID                        dst:%ld
    orientation: scalar (AS_NORMAL, AS_ANTI
                 AS_INNIE,AS_OUTTIE,
                 AS_UNKNOWN)          ori:[NAIOU]
   end
  end
end
```

## Audit Records:

Every pipeline transmission batch will have an audit record as its first item. An audit record will consist of a list of the agents that produced the batch in the sequence they were applied, and for each agent the name of the agent, time of completion, version number, and a possibly empty comment are specified:

```
AuditMesg:                                         {ADT
 list of AuditLine                                 (<ADL-record>↲)*.
                                                   }
```

| | |
|---|---|
| AuditLine: | {ADL |
| **record** | |
| name: **string** | who:%s |
| complete: **time_t** | ctm:%d |
| version: **string** | vsn:%s |
| comment: **string** | com:%s |
| **end** | } |

# 2. GateKeeper:

The assembler modules require consecutive IDs beginning at 1 for efficient indexing of internal and disk-based data structures. These 32-bit "IID's" are assigned and added to every record containing a UID supplied by the external DMS with the exception of Repeat_IDs which already have this property. Thus the GateKeeper module augments all input messages -- FRG, LKG, SCR, and DST -- with internal ID assignments and passes them on as IFG, ILK, ISN,and IDT messages. These messages are identical to the input counterparts save that:

All `acc-`fields that contain external references are converted to (External,Internal) accession number pairs, encoded in 3-format as "(%d,%d)". In the corresponding C-structs, the single field, say "<X>" to the external ID, is replaced with two fields "e<X>" and "i<X>" to the appropriate sized ints.

All other fields that contain external references are converted to internal accesion numbers encoded in 3-format as "%d" and their field name, "X" becomes "i<X>".

The Gatekeeper further checks all input for semantic consistency as described in the defining document for that stage.

# 3. Repeat Tagger/Contaminant Screener:

The URT/URC module consumes ISC messages, passes IDT messages through unalterred, and adds to the IFG message. The consumed ISC messages are recorded in a screen index store on disk. To avoid any ambiguity the augmented fragment records are called "ScreenedFragMesg" and their 3-code header is "SFG". The new information is a sequence of ScreenMatch records incorporated into the fragment message as follows:

IntScreen_ID: **int32**

| | |
|---|---|
| ScreenedFragMesg: | {SFG |
| **record** | |
| "As before" | |
| clear_rng: SeqInterval | |
| screened: **sorted list of** ScreenMatch | scn:↲<ISM-record>*. |
| "As before" | |
| **end** | } |

Each match instance is represented by an IntScreenMatch record that is defined below. Note that IntScreenMatch items are always enclosed within the FragMesgScreened record, therefore a Fragment_ID field is not needed.

| | |
|---|---|
| IntScreenMatch: | {ISM |
| **record** | |
| where: SeqInterval in fragment | whr:%d,%d |
| iwhat: IntScreen_ID | wht: %d |
| repeat_id: [0…Num_of_Repeat_Types-1] | rpt:%d |

```
 relevance:  int32                                  rel:%d
 portion_of: SeqInterval                            pof:%d,%d
 direction:  scalar (AS_FORWARD,AS_REVERSE)         dir:[FR]
end                                                 }
```

# 4. Overlapper:

The Overlap module stores screened fragments in a fragment store, passes on relevant fragment information to the subsequence stages in an OFG fragment message, and adds overlap messages (OVL) to the stream. An OFG fragment message is exactly an SFG save that:

(1) the type name is OFG instead of SFG

(2) the seq and `qlt` fields are absent.

Overlaps between fragments are described in "OverlapMesg" records as follows. It would be preferrable if the overlaps for a given fragment followed its OFG message and if that fragment were the A_frag for the relevant overlap records.

```
IntFrag_ID: int32

OverlapMesg:                                        {OVL
record
 aifrag:     IntFrag_ID                             afr:%d
 bifrag:     IntFrag_ID                             bfr:%d
 orientation:  scalar (AS_NORMAL,AS_INNIE,
              AS_OUTTIE,AS_ANTI)                    ori:[NAIO]
 overlap_type:
   scalar (AS_DOVETAIL,AS_CONTAINMENT,AS_SUPERREPEAT)  olt:[DCM]
 a_hang:     int32                                  ahg:%d
 b_hang:     int32                                  bhg:%d
 quality:    float32                                qua:%f
 min_offset,                                        mno:%d
 max_offset:  int32                                 mxo:%d
 polymorph_ct: int32                                pct:%d
 delta:      string(int)                            del:↲((%d)*↲)*.
end                                                 }
```

We add Anti's here for uniformity throughout.

# 5. Unitigger:

The Unitigger absorbs the OFG and OVL records emitted by the Overlapper. It passes ADL, ILK, and IDT messages through unalterred. It chunks fragments into unitigs and maintains a store of all fragment overlaps so that it can do this incrementally. The Unitigger further invokes the Consensus Module through a procedural interface to produce consensus sequences for each unitig and then computes branch points for each unitig. The Unitigger emits the resulting annotated unitig graph, as a series of IntUnitigMesg messages modeling the vertices and as a series of UnitigOverlapMesg messages modeling the edges.

```
IntChunk_ID: int32

IntUnitigMesg:                                      {IUM
record
 iaccession:     IntChunk_ID                        acc:%d
```

```
 source:      "description of data source"
  coverage_stat: float32
  status:           scalar (AS_UNIQUE,AS_CHIMER,
                            AS_NOTREZ,AS_SEP,
                            AS_UNASSIGNED)
 a_branch_point: int32
 b_branch_point: int32
  length:          int32
  consensus:       string(char)
  quality:         string(bytes)
  forced:          boolean
  num_frags:       int32
  f_list:          list of IntMultiPos
end

IntMultiPos: record
 type: scalar (AS_READ,AS_EXTR,AS_TRNR,
               AS_EBAC,AS_LBAC,AS_UBAC,
               AS_FBAC,AS_STS,AS_UNITIG)
 ident:     union(IntFragment_ID,IntChunk_ID)
 source:    "description of data source"
 position:   SeqInterval
 delta_length: int32
 delta:     list of int16
end
```

```
    src:↲(%[^\n]↲)*.
cov:%f

    sta:[UCNSX]
    abp:%d
    bbp:%d
    len:%d
    cns:↲(%[^\n]↲)*.
    qlt:↲(%[^\n]↲)*.
    for:%d
    nfr:%d
    (<IMP-record>↲)*
    }

    {IMP
    typ:[RXTELUFSu]

mid:%d
    src:↲(%[^\n]↲)*.
    pos:%d,%d
    dln:%d
    del:↲((%d )*↲)*
    }
```

I'm recommending we tighten the encoding of these: the type is implied by the list it is in.

A Unitig record contains the following information:

1. A unique accession number assigned by the assembler. These are densely assigned starting at 0, but not necessarily sequential.
2. The coverage statistic used to determine whether a unitig can safely be considered to be from a unique region of the genome.
3. The assembler's current status of the unitig is: AS_UNIQUE if the unitig is unique in terms of having a high coverage stat and being longer than 1kbp or if the unitig is repetitive (low coverage stat or 1kbp or shorter) and not placed in any scaffold; AS_CHIMER if the unitig represents a single fragment that has been deemed chimeric; AS_NOTREZ if the unitig is repetitive and is placed in only one scaffold; and AS_SEP if the unitig is repetitive and has instances in multple scaffolds. Note that a unitig with status equal to AS_SEP is completely separated and might have gaps. This assignment is made by the unitig/chunk graph walker. Initially the Unitigger assigns the value AS_UNASSIGNED to a unitig.
4. Each unitig has arbitrarily assigned ends, imaginatively called "A"/prefix and "B"/suffix. The ordering of the essential edges internal to the unitig starts from the "A" end. Branch points are only interesting if they occur relatively close to the end of a unitig, within AS_CGB_BPT_SEQ_LENGTH (1000) bases. The magnitude of the a_branch_point and b_branch_point fields specify how far from the respective unitig ends the branch point is located, and their sign gives the orientation of the branch points. A value of 0 indicates no branchpoint, a plus sign indicates going from repeat into unique, and a negative number designates transitioning from unique into repeat (never seen so far).
5. The `length` gives the gapped length of the consensus sequence for the unitig encoded in the `consensus` field and also the length of the accompanying quality array, `quality`, unless it is a NULL pointer in which case this field is absent.
6. If the consensus module had problems computing the Unitig's consensus the `forced` field will be set to TRUE to indicate potential quality problems in the consensus.
7. Finally, the multi-alignment of the unitig is encoded by a list of `num_frags` MultiPos records for which the sum of their delta arrays is `delta_total`.

A MultiPos specifies the position of a fragment or unitig within the multi-alignment of a unitig (contig) and a delta for aligning the item with the gapped consensus sequence. The position is specified as a SeqInterval -- an ordered pair of integers. If position.bgn < position.end, then the fragment (unitig) is

oriented along the direction of the unitig (contig), otherwise it has reverse orientation. The exact nature of a delta should be explained here!

A series of independent UnitigOverlapMesg messages specify unitig graph edges. Each edge identifies a pair of unitigs and the orientation of the overlap between them . In this situation, the designers didn't employ the Normal, Anti-Normal, Innie, and Outtie convention of the overlapper, but chose instead the indicated scalar whose interpretation is as follows:

- AB_AB ‘N’ Normal
- BA_BA ‘A’ Anti-Normal
- AB_BA ‘I’ Innie
- BA_AB ‘O’ Outtie

The idea is that the letters on the left side of the underbar depict the orientation of unitig1 (using it’s A and B ends) and the letters on the right side give that of unitig2.

The overlap_type specifies the relationship between the unitigs:

- AS_NO_OVERLAP ‘N’ No overlap
- AS_OVERLAP ‘O’ A dovetail overlap between two non-contained unitigs.
- AS_TANDEM_OVERLAP ‘T’ Tandem overlap
- AS_1_CONTAINS_2_OVERLAP ‘C’ unitig1 contains unitig2
- AS_2_CONTAINS_1_OVERLAP ‘I’ unitig2 contains unitig1
- AS_TOUCHES_CONTAINED_OVERLAP 'M' A dovetail overlap between a non-contained unitig and a contained unitig.
- AS_BETWEEN_CONTAINED_OVERLAP 'Y' A dovetail overlap between two contained unitigs.
- AS_TRANSCHUNK_OVERLAP 'X' A transitively inferable dovetail overlap.

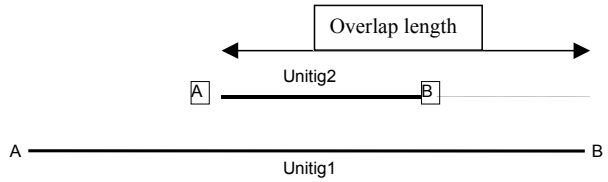
In the case of the containment overlaps, the are two independent orientations for each overlap (the unitigs are either aligned in the same direction, or in opposite directions). By arbitrarily constraining the containing unitig to be in the AB orientation, this boils down to the following constraints on orientation:

- AS_1_CONTAINS_2 overlap
  **AB**_AB (Normal) or **AB**_BA (Innie)
- AS_2_CONTAINS_1 overlap
  AB_**AB** (Normal) or BA_**AB** (Outie)

Unknown
**Formatted:** Bullets and Numbering

Unknown
**Formatted:** Bullets and Numbering

normal and innie, since the two unitigs are either aligned in the same orientation, or aligned with opposite orientation. The overlap distance is specified as if the contained fragment was extended past the B end of the containing fragment.




New Drawing

In addition, the best, minimum, and maximum overlap length between the unitigs is given. The minimum and maximum are not equal in the case of a most edges induced by a small tandem repeat. ~~However, all such edges are not detectable just on overlap, and the CGB transitively infers when the overlapping parts of an edge involve tandem satellites. This is set in the `overlap_type` field.~~ The `overlap_type` field

is also used later, but in this context the AS_NO_OVERLAP and AS_TANDEM_OVERLAP value is never a possibility.

```
UnitigOverlapMesg:                                          {UOM
record
 chunk1:          IntChunk_ID                               ck1:%d
 chunk2:          IntChunk_ID                               ck2:%d
 orient:          scalar (AB_AB, BA_BA,
                  BA_AB, AB_BA)                             ori:[NAOI]
 overlap_type:      scalar (AS_NO_OVERLAP,
                  AS_OVERLAP,
                  AS_TANDEM_OVERLAP,
                AS_1_CONTAINS_2_OVERLAP,
                AS_2_CONTAINS_1_OVERLAP,
                AS_TOUCHES_CONTAINED_OVERLAP,
                AS_BETWEEN_CONTAINED_OVERLAP,
                AS_TRANSCHUNK_OVERLAP
                )                                           ovt:[NOTCIMYX]
 source:            "description of data source"            src:↲(%[^\n]↲)*.
 best_overlap_length: int32                                 len:%d
 min_overlap_length:  int32                                 min:%d
 max_overlap_length:  int32                                 max:%d
 quality:       float32                    qua:%f
 polymorph_ct: int32                       pct:%d
End                                                         }
```

# 6. Chunk Graph Walker:

**Extented Unitig Graph:**

The first task of the unitig/chunk graph walker is to extend the unitig/chunk graph by adding mate edges. These edges connect unitigs that contain linked reads. The resulting graph, termed the Extended Unitig Graph (EUG) consists of vertices that are Unitigs, and undirected edges in the form of Unitig Link Edges. The CGW swallows the UnitigOverlapMesg and LinkMesg messages and emits Unitig Link Edges, as follows:

```
IntUnitigLinkMesg:                                          {IUL
record
 unitig1:         IntChunk_ID                               ut1:%d
 unitig2:         IntChunk_ID                               ut2:%d
 orientation:       scalar {AB_AB, BA_BA,
                  BA_AB, AB_BA}                             ori:[NAOI]
 overlap_type:      scalar (AS_NO_OVERLAP,
                  AS_OVERLAP,
                  AS_TANDEM_OVERLAP,
                AS_1_CONTAINS_2_OVERLAP,
                AS_2_CONTAINS_1_OVERLAP)
                                                            ovt:[NOTCI]
 is_possible_chimera:    boolean                             ipc:%d
 includes_guide:       boolean                               gui:%d
 mean_distance:        float32                               mea:%f
 std_deviation:        float32                               std:%f
 num_contributing:       int32                               num:%d
 status:          scalar (AS_IN_ASSEMBLY,
                    AS_POLYMORPHISM,
                    AS_BAD,
                    AS_CHIMERA,
                              AS_UNKNOWN_IN_ASSEMBLY)       sta:[APBCU]
 jump_list:         list of IntMate_Pairs                    jls:↲(%d,%d↲)*
end                                                         }

IntMate_Pairs: record
```

```
                in1, in2: IntFrag_ID
        type:  scalar (AS_MATE,
                AS_BAC_GUIDE,                          typ:[MBSYT]
                        AS_STS_GUIDE,
                        AS_MAY_JOIN,
                        AS_MUST_JOIN)

    end
```

The first four fields are identical in meaning to the corresponding fields of the UnitigOverlapMesg. The only difference is that now the has_overlap field can take on the value AS_NO_OVERLAP as two unitigs may be connected solely by links. If the number of contributing edges is two, and a single read is required for both edges, then `is_possible_chimera` is set to true. This will happen if a read is part of a mate in the other unitig and is also required for the unitig to overlap. If the edge includes a guide, the `includes_guide` is set to TRUE. The `mean_` and `std_distance` fields give the range of mean and standard deviation of the distance separating the two unitig (a negative distance means the unitigs overlap). The number of edges (read mates and a possible unitig overlap) contributing to the mate edge is given by the field num_contributing. The status field, determined late in the process after a best scaffold has been chosen, gives the status of the edge with respect to this assembly. Finally, the `jump_list` gives a list of all mate pairs of fragments modelled by the edge. The length of the jump_list corresponds to the number of contributing edges if `overlap_type` takes the value AS_NO_OVERLAP. Otherwise the length of the jump_list will be num_contributing – 1.

**Extended Contig Graph:**

The CGW next outputs the extended contig graph. Contigs are ordered collections of Unitigs and Surrogates that cover contiguous regions of the genome. A contig is composed of fragments from the contained Unitigs, as well as "surrogates". Surrogates are subsets of repeat Unitigs that are introduced to span gaps in contigs that result from incomplete repeat fragment resolution in repeat Unitigs. Even where no gaps are introduced in a contig, surrogates may be necessary to provide the necessary overlap 'glue' for consensus. The need for surrogates is determined by the consensus module, and indicated in its output.

Following CGW, consensus expects Unitig and PreContig messages, with the ordering requirement that any Unitig message referenced in a given PreContig message appears in the stream prior to its reference. For each contig in the assembled layout, the Assembly module is expected to emit a PreContig message that specifies the contig identifier, contig length, a count of fragments contained within the contig layout, and a list of ElementPos messages, one for each such item. The position information for each item is encoded as ElementPos.

In the event that consensus requires a surrogate for Unitigs that are separated (status AS_SEP) or unresolved (AS_UNRESOLVED) it should avail itself of the previously computed consensus sequences for the Unitigs. Unitigs marked as (AS_SEP) might need to do that if the contain gaps after being separated/ The use of surrogates should be reflected both in the quality values of the resulting consensus, and in the list of surrogates in the output.

```
PreContigMesg: record                                   {PCM
 iaccession:    IntContig_ID                            acc:%d
 iscaff_id:     IntScaff_ID          sid:%d
 placed:              boolean                           pla:%d
 length:       int32                                    len:%d
 num_frags:     int32                                   nfr:%d
 f_list:       list of IntElementPos                    (<IEP-record>↲)*
 num_unitigs:   int32                                   nou:%d
 unitigs:      list of IntElementPos                    (<IEP -record>↲)*
end                                                     }
```

```
IntElementPos: record                          {IEP
record
 type: scalar (AS_READ,AS_EXTR,AS_TRNR,        typ:[RXTELUFSu]
               AS_EBAC,AS_LBAC,AS_UBAC,
               AS_FBAC,AS_STS, AS_UNITIG)
 ident:    union(IntFragment_ID,IntChunk_ID)  lid:%d
 position: SeqInterval                        pos:%d,%d
end                                           }
```

~~The sid field in the PCM message is a reference to the scaffold in which this contig has been placed, if any. If the contig has not been placed, sid will be set to the sentinel value of -1. This sid is exceptional in that it is the only forward reference to an undefined object in the assembler's I/O spec. The scaffold that is referred to by the sid must appear subsequently.~~

The placed field is true if this contig has been placed in a scaffold.

The edges in the contig graph are represented by Contig Link edges, that are direct analogs of the Unitig Link Edges in the unitig graph. The only differences is in the objects being related.

```
IntContigLinkMesg:                                          {ICL
record
 contig1:          IntContig_ID                             co1:%d
 contig2:          IntContig_ID                             co2:%d
 orientation:       scalar {AB_AB, BA_BA,
                   BA_AB, AB_BA}                            ori:[NAOI]
 overlap_type:      scalar (AS_NO_OVERLAP,
                   AS_OVERLAP,
                   AS_TANDEM_OVERLAP)                       ovt:[NORT]
 is_possible_chimera:  boolean                              ipc:%d
 includes_guide:     boolean                                gui:%d
 mean_distance:      float32                                mea:%f
 std_deviation:      float32                                std:%f
 num_contributing:    int32                                 num:%d
status:           scalar (AS_IN_ASSEMBLY,
                   AS_POLYMORPHISM,
                   AS_BAD,
                   AS_CHIMERA,
                            AS_UNKNOWN_IN_ASSEMBLY)         sta:[APBCU]
 jump_list:         list of IntMate_Pairs                    jls:↲(%d,%d↲)*
end                                                         }
```

**Mate-Distance Distribution Messages:**

These messages are emitted to provide information on the distribution of mate lengths observed for those pairs both of which are in the same unitig or contig (and thus whose distance is known precisely). For each mate-link distance type provided to the assembler, a message describing the distribution mates in the current assembly is produced:

```
IntMateDistMesg:    {IMD
record
 refines:   IntDistance_ID                          ref:%d
 mean:      float       mea:%f
 stddev:    float       std:%f
 min:       int32       min:%d
 max:       int32       max:%d
 num_buckets: int32     buc:%d
 histogram:  list of int32                          his:↲(%d↲)*
end                                                 }
```

`refines` indicates the distance type for which this is the distribution. mean and stddev are the calculated mean and standard deviations for mate pairs of this type. `min` and `max` are the minimum and maximum distances observed for this type. `histogram` is a list of `num_buckets` entries, each of which is a count

of the number of occurrences of a distance in the corresponding subrange of the entire range from `min` to `max`.

**Interior Augmented Fragment Messages:**

These messages pass on information about the fragments obtained when building contigs and scaffolds. The chimeric flag is set if the read appears to be a chimera based on the extended unitig graph. The chaff fragment is set if the read is a singleton – not incorporated with other fragments into any scaffold. The clear_rng field gives the clear range of the fragment. This value may be the same as the one input into the assembler, or the assembler may have adjusted it based on examination of the extended unitig graph. The mate_status gives the assembler's determination of the correctness of the mate. The values of this field are described more completely under the in the AugFragMesg section.

```
IntAugFragMesg:                                          {IAF
record
 iaccession:   IntFragment_ID                            acc:%ld
 type:         scalar (AS_READ,AS_EXTR,AS_TRNR,          typ:[RXTELUFS]
                 AS_EBAC,AS_LBAC,AS_UBAC,
                 AS_FBAC,AS_STS)
 chimeric:     Boolean                                   chi:%d
 clear_rng:    SeqInterval                               clr:%d,%d
 mate_status:  scalar (GOOD_MATE, BAD_MATE, NO_MATE,
                 UNRESOLVED_MATE)                        mst:[GBNU]
end                                                      }
```

**Scaffold Messages:**

A series of the assemblers best choices for the scaffolds is output as "the" assembly. This message will actually be produced by the CGW and passed through by Consensus. Each scaffold is given as a list of the pairs of adjacent contigs in the scaffold with the chi-squared consensus estimate of the distance and standard deviation thereof between the contigs. The scaffold and unitig links supporting the scaffold are all known by virtue of having an AS_IN_ASSEMBLY status value.

```
IntScaffoldMesg:       {ISF
record
 num_contigs_pairs: int32                                noc:%d
 contig_pairs:    list of IntContigPairs                 (<ICP -record>↲)*
end                                                      }

IntContigPairs:                                          {ICP
record
 contig1: IntContig_ID                                   ct1:%d
 contig2: IntContig_ID                                   ct2:%d
 orientation:        scalar {AB_AB, BA_BA,
                   BA_AB, AB_BA}   ori:[NAOI]
 mean:     float32                                       mea:%f
 stddev:   float32                                       std:%f
end                                                      }
```

In the list of contig pairs, the contigs are ordered from left to right across the scaffold. For example, if the first three contigs in a scaffold have ids 1, 2, & 3. Then in the list of contig pairs, the first pair of contigs would have contig1 = 1 and contig2 = 2, and the second pair would have contig1 = 2 and contig2 = 3. The orientation field describes the pairwise orientation of the two contigs within the scaffold. (Previously this field always had the implied value of 'N', and all contigs were oriented along the direction of the scaffold.) A scafffold may consist of a single contig, in which case the num_contig_pairs will be zero and the id of the second contig will be the sentinel value –1, and the orientation and distance are arbitrary.

# 7. Consensus:

Consensus intercepts PreContigMesg messages and converts them to IntConConMesgn messages through the action of computing a multialignment for each. In doing so it identifies any unitigs that must be included in the multi-alignment in order to avoid sequence gaps. These unitigs are called surrogates.

```
IntConConMesg:                                  {ICM
record
 iaccession:      IntContig_ID                  acc:%d
length:          int32                         len:%d
 consensus:       string(char)                  cns:↲(%[^\n]↲)*.
 quality:         string(bytes)                 qlt:↲(%[^\n]↲)*.
 forced:          boolean                       for:%d
 num_pieces:      int32                         npc:%d
 num_unitigs:     int32                         nou:%d
 pieces:          list of IntMultiPos           (<IMP-record>↲)*
 unitigs:         list of IntElementPos         IEP-record>↲)*
end                                             }
```

An IntContigMesg record contains the following information:

1. A unique accession number assigned by the assembler.
2. Time of last modification of the contig (if this hasn't changed since the last interim report then the contig is identical to what was specified in the last report).
3. The gapped-consensus sequence determined for the contig. Note carefully the consensus will contain dash characters in order that one be able to align each fragment with it.
4. A list of the fragments and surrogate unitigs assigned to the contig. For each we give in a MultiPos-record their location in the multi-alignment and a delta encoding of the alignment to the consensus. Positions in the consensus are locations *between* characters starting at 0 on the left. The delta is a series of positive integers indicating the positions within the fragment's clear range at which to insert a dash. The delta encoding for the alignment of a unitig in a contig is with respect to the unitig's ungapped sequence coordinates. Note that in aggregate, these records specify the layout of the contig and a precise representation from which to compute a multi-alignment with just a bit of additional effort.

# 8. External Output: The Genome Snapshot:

The output from the entire assembly module is a series of messages that collectively comprise a Genome Snapshot, *i.e.*, a description of the current best state of the assembly. This includes information about: individual fragments; the distribution of distances between fragment mates; how the fragments have been assembled into unitigs (chunks); how the unitigs have been combined into contigs; and how the contigs can be laid out across the genome.

**Fragment Messages:**

The information about individual fragments is contained in augmented fragment messages (`AFG`). These are emitted for each fragment that the assembler has processed and contain essentially all the information about that fragment relative to the current assembly. The format of the message is:

| | |
|---|---|
| AugFragMesg: | {AFG |
| **record** | |
| accession: Fragment_ID | acc:%ld |
| screened: **list of** ScreenMatch | scn:↲*<SMA-record>**. |
| mate_status: **scalar** (GOOD_MATE, BAD_MATE, NO_MATE, UNRESOLVED_MATE) | mst:[GBNU] |
| chimeric: **Boolean** | chi:%d |
| chaff: **Boolean** | cha:%d |
| clear_rng: **SeqInterval** | clr:%d,%d |
| **end** | } |

Most of these fields are the same as those already described in previous fragment messages. Of the others:

1. `chimeric` indicates if this fragment has been determined to be chimeric. If so, the fragment will be in a singleton unitig.
2. `chaff` indicates the fragment is a singleton – not incorporated with any other fragments into a scaffold.
3. `clear_rng` is the clear range of the fragment. This may have been altered by the assembler.
4. mate_status is the assembler's determination of whether mates are erroneous of not. NO_MATE indicates that no mate for the fragment was input into the assembler. GOOD_MATE means the mate is confirmed by the assembly. BAD_MATE means the mate is inconsistent with the assembly. UNRESOLVED_MATE means the mate is neither confirmed by the assembly nor inconsistent with the assembly.

**Unitig, Contig, Scaffold and Mate Distance Distribution Messages:**

The assembler also outputs the IUM, IUL, ICM, ICL, ISF, and IMD messages, save that all internal IDs are converted to long external IDs. In this form of the messages the 3-codes become UTG, ULK, CCO, CLK, SCF, and MDI, respectively. Moreover, in all the corresponding datastructure names that begin with "Int" have this prefix removed. Also the internal submessages IMP, IEP, and ICP become MPS, EPS, and CTP, respectively.

**AUTHORS**

Gene Myers

Created: October 14, '98

Revised: Jan. 8, '99 by Gene Myers

Added GateKeeper Module section to design with associated changes to subsequent stages.

Revised: Feb. 1, '99 by Gene Myers

Removed unecessary external Ids from internal message references

Revised: Feb. 8, '99 by Gene Myers

Incorporated chunk graph specifications, with modifications.

Revised: Feb. 22, '99 by Gene Myers

Incorporate consensus module specifications.

Revised: Mar. 5, '99 by Ian Dew

Added relevance field to ScreenItemMesg and ScreenMatch.

Revised: Mar. 10, '99 by Karin Remington

Modified consensus output, mostly to include size info.

Revised: Mar. 22, '99 by Eric Anson

Had to redo definitions of Chunk Mate Edge messages after a copy of this document became somehow corrupted.

Revised: Mar. 30, 1999 by Art Delcher

Changed branch-point message.

Revised: Apr 9, 1999 by Art Delcher

Added Genome Snapshot section.

Revised: April 23, '99 by Saul A. Kravitz

Updates to Fragments, Links, Distance, Chunks and Consensus

Revised: May 11, '99 by Clark Mobarry

Changes to the CHK messages and removed the join messages from the information flow.

Revised: May 11, '99 by Karin A. Remington

Changes to RTG and CTG messages and descriptions thereof.

Revised: June 2, '99 by Gene Myers

Attempt to finalize output of pipeline and sanitize entire document.

Revised: May 11, '99 by Knut Reiner

Changes to the genome snapshot three letter codes.

Revised: June 29,'99 by Eric Anson

Further description of the scaffold messages and inserting the mate status field back into the AugFrag messages.

Revised: July 26,'99 by Saul A. Kravitz

Added type to IntMatePair.

Revised: July 28,'99 by Saul A. Kravitz

Added LinkMesg orientation AS_UNKNOWN and some description.

Revised: August 9,'99 by Saul A. Kravitz (1.55)

Duplicated some text from CeleraAssembler.rtf here

Revised: September 22,'99 by Saul A. Kravitz (1.56)

Proposed changes to contigs and scaffolds, subject to CarlD's and GeneM's approval. Cleaned up some of the historical material. This can always be retrieved from CVS.

Revised: September 22,'99 by Saul A. Kravitz (1.57)

Reflects Gene's Feedback

Revised: September 24,'99 by Saul A. Kravitz (1.58)
Change to UOM message for containment

Revised: September 24,'99 by Saul A. Kravitz (1.59)
Change to IUL message for containment

Revised: October 6,'99 by Saul A. Kravitz (1.61)
Corrections and clarifications to LKG message

Revised: October 6,'99 by Saul A. Kravitz
Corrections and clarifications to LKG message

Revised: December 10, 1999 by Clark M. Mobarry
Modifications to the UOM messages.