ASSEMBLER PIPELINE I/O

(Proposed Amendments for HUMAN)

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 specifications for the input and output of the assembler. The document describes the messages in order of their introduction along the assembler pipeline, and provides an Appendix with concrete examples of each input/output message.

Note: The addition of Scaffold Link (SLK) messages, the addition of internal ID fields to the defining message for all data types; and the changes to Contig messages and the addition of Degenerate Scaffold messages in support of unscaffolded contigs.

1 Conventions

The Assembly Group has adopted an organic software development strategy, rapidly prototyping pipeline components and then evolving them into robust components. To this end, the individual modules are engineered to communicate via a simple ASCII-based encoding of the pipeline messages, which can be switched over to a more compact and efficient binary representation in production situations. The requirement for the ASCII encoding was that it be easy to read by a human (aiding debugging), while also being trivial to parse. The result is the 3-code formatted messages described within this document.

1.1 3-code format

The format of all messages 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 solely 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, or in subsequent lines in the case of multi-line fields.

1.2 External & Internal Accession Numbers

All input and output data is uniquely identified by accession numbers -- 64-bit unique identifiers. The assembler uses its accession numbers for each object type in the form of a 32-bit unsigned integer. In many of the message types described here, both the internal and external accession numbers are included for tracking purposes. The field names for external accession names start with an 'e', whereas the field names for an internal accession number start with an 'i'.

To facilitate analysis of assemblies, the internal accession number should be loaded into the appropriate DB tables.

1.3 Data formats

1.3.1 Numeric and character data

For the messages specified in this document, the Pascal data structure is given (at left) along with the corresponding 3-code (in shaded area at right). The encoding of the data for each field is given by specifying the *scanf* UNIX format that would correctly read the input. Thus, for example, '%lu' reads a long and '%1[AD]' reads one character, from the two characters in square brackets. By convention, scalar values are encoded as a single capital letter and the sequence of letters corresponds directly to the sequence of value names in the scalar definition.

1.3.2 String Data

Arbitrary length strings, as needed for 'Quality', 'Sequence' and 'Source' fields, are encoded as a series of new-line terminated strings ending with a line containing a '.' in column 1. In the *scanf* translation specification of such message components, we take the liberty of using regular expression syntax for these fields, with the symbol '↵' denoting a new-line. The encoded string is the concatenation of all the characters save the new-lines and the terminating period.

1.4 Sequence Intervals

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, while (2,2) specifies the position between the second and third symbols.

```
SeqInterval: record bgn, end: int32 end
```

1.5 Alignment Positions

For every fragment in a contig/unitig alignment, a complete record of its alignment to the gapped consensus is supplied. The endpoints of the fragment in the alignment are specified as a SeqInterval, while the detailed alignment is given by a 'delta encoding'. This delta is a series of positive integers indicating the positions within the fragment's clear range at which to insert a dash to align to the consensus sequence. 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 complete layout of the contig/unitig, and a precise representation from which to reconstruct the multi-alignment with just a bit of additional effort.

1.6 Quality Values

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.

1.7 ProtoIO File Termination

All valid ProtoIO files shall be terminated with an "End of File" (EOF) message whose purpose is to signal that the ProtoIO file was written properly and in its entirety. The EOF message contains fields specifying at what time the message was written and the status of the run producing the message. In keeping with UNIX

conventions, a status of 0 is normal and any other value is an error code. Any file which ends in an EOF message with a non-zero status, contains no EOF message, or contains multiple EOF messages is considered invalid and should be used only for debugging purposes. Finally, the message also contains space for a text comment.

The EOF message also contains two fields to support checksum verification of the file. The first field is a boolean flag; a non-zero value indicates that a checksum has been computed for the field and is in the subsequent checksum field. A value of 0 indicates that the value in the subsequent checksum field should be ignored.

```
EndOfFileMesg:
record
   status: int32
   created: time_t
   has_check: int32
   checksum: uint64
   comment: string(char)
end
```

```
{EOF
sta:%d
crt:%d
hck:%d
chk:%lu
com:↲(%[^\n] ↲)*.
}

acc:                %lu

acc:        %lu

acc: %lu
```

2 Software Support

This specification is supported by the proto IO library, that is documented elsewhere.

3 Inputs

The input to the assembler is prefixed with an “audit record” for tracking purposes. Following the audit record, there are three main components to the Assembler input: fragments, links and distance records, and screen and repeat items.

3.1 Audit and Batch Messages

Every pipeline transmission batch will have a batch record as its first item and an audit record as its second item. A batch record will consist of the name of the batch, the time the batch was created, a unique identifier for the batch, and a possibly empty comment field. 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:

Batch_ID: **uint64**

```
BatchMesg:
record
 name:     string
 created:  time_t
 eaccession: Batch_ID
 comment:  string
```

```
{BAT

bna:%s
crt:%d
acc:%lu
com:%s
```

```
end                                                    }

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                                                    }
```

3.2 BACs

Each BAC that is the source of finished or unfinished pseudo-reads (shredded reads), unshredded sequence, light-shotgunned reads or BAC end reads is defined by a BAC message. The reads that derive from the BAC will reference the BAC through their locale field. There are three possible levels at which a BAC or portion there of are identified in the BAC message: bac_id, seq_id, and (num_bactigs , bactig_list). A distance record which contains the mean and standard deviation of the BAC's length is referenced in this record, and a text field describing the origin of the BAC is also included.

Distance_ID, Locale_ID: **uint64**

```
{BAC
record
action: scalar (AS_ADD, AS_DELETE,
                AS_REDEFINE)                           act:[ADR]
ebac_id:          Locale_ID                            bid:%lu
type: scalar  (AS_ENDS, AS_LIGHT_SHOTGUN, AS_UNFINISHED,
                AS_FINISHED)                           typ:[ELUF]
eseq_id:          Locale_ID                            sid:%lu
entry_time: time_t                                     etm:%d
elength:          Distance_ID                          len:%lu
num_bactigs:  int16                                btg:%d
bactig_list:  list of BactigRec                    (<BTG-record>↵)*
source:           "description of data source"         src:↵(%[^\n]↵)*.
end                                                    }

BactigRec: record                                      {BTG
eaccession:   Locale_ID                        acc:%lu
Length:           int32                                len:%d
end                                                    }
```

There are four types of BACs, specified as follows:

AS_ENDS:

Only the ends of the BAC have been sequenced to use as BAC guides. . The num_bactigs

field and the bactig_list are undefined, and will not appear in the ASCII version of this message.

AS_LIGHT_SHOTGUN:

A low coverage shotgun has been performed and the reads have not been assembled. The BAC contents will appear as LBAC fragment reads. The num_bactigs field and the bactig_list are undefined, and will not appear in the ASCII version of this message.

AS_UNFINISHED:

A medium coverage shotgun has been performed and the reads have been assembled into contigs (which we will here after refer to as bactigs to represent contigs derived in this fashion from a BAC). The BAC contents will appear as either/both Bactig reads, or as shredded UBAC fragments. The (num_bactigs , bactig_list) fields are only relevant for this type of BAC.

AS_FINISHED:

The entire sequence of the BAC has been sequenced and assembled into a single contig. The BAC contents will appear as either/both a single FullBAC read or shredded FBAC fragments. . The num_bactigs field and the bactig_list are undefined, and will not appear in the ASCII version of this message.

The **bac_id** field is relevant for all four types of BAC message and specifies the unique identifier for the BAC which is persistent in the database even if features or sequence of the BAC change.

The **seq_id** field is only relevant for AS_UBAC and AS_FBAC types and specifies a unique identifier for the sequence of the BAC which will change when the sequence changes.

The **redefine** action is used to redefine a BAC to a more finished state. The allowed transitions are from type AS_ENDS to the other three types, from AS_LIGHT_SHOTGUN to AS_UNFINSHED or AS_FINISHED, or from AS_UNFINISHED to AS_UNFINISHED or AS_FINISHED. These redefinitions do not require the deletion of previous fragments which reference this BAC via the Locale_ID. Note that a redefine can only go toward a more finished state so, for example, once a BAC has been defined as an AS_FINISHED, BAC end or light shotgun fragments can not be input for that BAC. However, BAC end and/or light shotgun fragments can be input for a BAC before it is redefined as an AS_FINISHED so the order of messages is important. The reason for allowing a redefinition from type AS_UNFINISHED to AS_UNFINISHED is to reflect a better intermediate assembly for the same BAC presumably due to more sequence data for that BAC without requiring that all data referencing the BAC be deleted first. When redefining type AS_UNFINISHED to type AS_UNFINISHED an entirely new set of bactigs should be defined in the bactig_list. A redefine does not delete the Bactigs and Fragments associated with old incarnations of the BAC. These may still be referenced by subsequent messages.

3.3 BINs

This section is an attempt to anticipate how STS information could be defined for input to the

assembler. This should be treated as speculative until it is better defined.

Each BIN represents a bin formed by a LOD score of 6.

```
BinMesg:                                                {BIN
record
action: scalar (AS_ADD, AS_DELETE)                      act:[AD]
eaccession:      Locale_ID                          acc:%lu
entry_time: time_t                                      etm:%d
source:             “description of data source”        src:↵(%[^\n]↵)*.
end                                                     }
```

3.4 Fragments (reads and guides)

The primary input to the assembler is a **fragment message**, which can describe a Celera read (AS_READ and AS_TRNR), an external whole genome shotgun read (AS_EXTR), or a guide. Guides are sub-categorized as either being (a) BAC-ends (AS_EBAC), (b) pseudo-reads from unfinished BACs (AS_UBAC), (c) pseudo reads from finished BACs (AS_FBAC), (d)reads from lightly-shotgunned BACs (AS_LBAC), and (e) STSs (AS_STS). In addition, for the overlay assembler much longer sequence guides are allowed which are the sequence of unshredded bactigs from unfinished BACs (AS_BACTIG) and possibly the complete sequence of finished BACs (AS_FULLBAC). The differences between read and guide fragment messages are as follows.

1. **Accession**: Every read or guide has an accession number. The interpretation of the accession number is always expected to be a 64-bit UID produced by the Celera database. For AS_READ, AS_EXTR, AS_TRNR, AS_EBAC, AS_LBAC, AS_UBAC, AS_FBAC, and AS_STS this is the fragment UID. For AS_BACTIG this is the bactig UID. For AS_FULLBAC this is the BAC’s Seqiemce UID.
2. **Quality**: Every read has a quality vector and sequence clear range, whereas guides need not. In the absence of quality values, the quality field is NULL and the clear range is the entire fragment.
3. **Locale UID**: 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.. This UID must have been previously defined as a BAC. If over time, end reads, then bactigs, and finally finished sequence for a BAC become available, the same locale 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 (we suggest 6). is used to order STSs. Each bin must have been previously defined as a BIN message**. This field is not defined for celera reads, does not appear in the ascii version of the message, and will have an undefined value**
4. **Sequence UID**: This field is **only** defined for unfinished BACs (AS_UBAC), unfinished and unshredded BACs (AS_BACTIG), finished BACs (AS_FBAC) and finished and unshredded BACs (AS_FULLBAC) (It is not defined for AS_EBAC, AS_LBAC, AS_READ, AS_EXTR, AS_TRNR, and AS_STS, its value is not defined, and it will not appear in the ASCII version of the message). This is a 64-bit UID produced by the Celera database to track the sequence of a BAC.

5. **Bactig UID**: This field is **only** defined for unfinished BACs (AS_UBAC), and unfinished and unshredded BACs (AS_BACTIG). This is a 64-bit UID produced by the Celera database to track the sequence of a bactig. (Elsewhere, it is not defined and it will not appear in the ASCII version of the message).

6. **Locale Position**: Each contig of an unfinished BAC (AS_UBAC) and each finished BAC (AS_FBAC) 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 requires the interval of the underlying BAC (for AS_FBAC) or bactig (for AS_UBAC) from which the pseudo-read was excised, communicated in the locpos field. This field is defined **only** for unfinished and finished BACs (AS_UBAC and AS_FBAC). Elsewhere, it is not defined, and it will not appear in the ASCII version of the message. In the case of several unordered "bactigs" (contigs of an unfinished BAC) for a given unfinished BAC, each bactig should have been assigned a unique identifier (passed in the bactig field), defined by the associated BAC message, and the locpos interval is defined relative to the position of the pseudo read within the bactig.

```
FragMesg:                                          {FRG
record
 action:    scalar (AS_ADD,AS_DELETE)              act:%1[AD]
 eaccession: Fragment_ID                           acc:%lu
 variant of action:
  AS_ADD:
   record
   type: scalar (AS_READ,AS_EXTR,AS_TRNR,          typ:%1[RXTELUFSBC]
                 AS_EBAC,AS_LBAC,
                 AS_UBAC,AS_FBAC,AS_STS,
                 AS_BACTIG,AS_FULLBAC)
    elocale:     Locale_ID                         loc:%lu
    eseq_id:     Locale_ID                         sid:%lu
    ebactig_id:  Locale_ID                         btd:%lu
    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                                                }
```

3.5 Links and associated distance records

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 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. The last two link types model user input constraints and may be between any pair of fragments in the system.

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 and external sources of whole genome shotgun sequence, or from AS_LBAC fragments that were sequences from opposite end of subclones.
- **AS_BAC_GUIDE** links are between end-sequenced BACs.
- **AS_STS_GUIDE** links are between paired STSs.
- **AS_REREAD** links permit 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.
- **MAY_JOIN** links represent single links that may be incorporated if there is not conflicting information.
- **MUST_JOIN** links represent infinitely weighted links that will be followed if at all possible.

The distance between mates is specified in **distance records** that are passed to the assembly system as records requesting that a given distance entity be added or deleted. The distance 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 fields `mean` and `stddev` give the mean standard deviation of the distribution. Thus, for example, 99% of all links referring to a particular distance message will be of length in `[mean-3stddev, mean+3stddev]`. Note 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 for BAC Guides is specified in a BAC message so for type AS_BAC_GUIDE the Distance_ID in the LKG message must be the same as the Distance_ID in the corresponding BAC message. The type AS_STS_GUIDE has not been implemented yet so the defining message for Distance_ID in this case has not been defined.

The orientation 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.

```
Distance_ID, Fragment_ID, Screen_Item_ID: uint64

LinkMesg:                                          {LKG
record
 action: scalar (AS_ADD,AS_DELETE)                 act:%1[AD]
 type:   scalar (AS_MATE,AS_BAC_GUIDE,             typ:%1[MBSRYT]
                 AS_STS_GUIDE, AS_REREAD,
                 AS_MAY_JOIN,AS_MUST_JOIN)
 efrag1: Fragment_ID                               fg1:%lu
 efrag2: Fragment_ID                               fg2:%lu
 variant of action:
  AS_ADD:
   record
    entry_time:  time_t                            etm:%d
    edistance:    Distance_ID                      dst:%lu
    orientation: scalar (AS_NORMAL, AS_ANTI        ori:%1[NAIOU]
                         AS_INNIE,AS_OUTTIE,
                         AS_UNKNOWN
```

```
   end
  end
end
```

```
}
```

```
DistanceMesg:
record
 action:    scalar (AS_ADD,AS_DELETE)
 eaccession: Distance_ID
 mean:      float32
 stddev:    float32
end
```

```
{DST

act:%1[AD]
acc:%lu
mea:%f
std:%f
}
```

3.6 Screen Items

Screen items are used to specify vector, contaminant, or repeat sequences that should be masked or tagged for the purposes of assembly. The repeat_id field indicates a user determined classification of the item, item, while the relevance bit-vector field is used to indicate how matches should be handled within the screener or in subsequent programs. For example, in the case of ubiquitous repeats, a relevance value that binary-ands with AS_OVL_HEED_RPT (1) instructs the overlapper not to base overlaps on sequences in matching intervals. Simple repeats (and heterochromatin) should have relevance fields that binary-and with AS_URT_IS_SIMPLE (8) to instruct the screener to consolidate matches more effectively.

Repeat_ID: **uint64**

```
ScreenItemMesg:
Record
 Action:    (AS_ADD, AS_UPDATE, AS_DELETE)
 Type:       scalar(AS_UBIQREP,AS_CONTAMINANT)
 eaccession:  ScreenItem_ID
 erepeat_id: Repeat_ID
 Relevance:  int32
 Source:     "description of data source"
 Sequence:   string(char)
 Variation:  float32 in [0,.1]
 Min_length: int32
End
```

```
{SCN

act:%1[AUD]
typ:%1[UC]
acc:%lu
rpt:%lu
rel:%d
src:↵(%[^\n]↵)*.
seq:↵(%[^\n]↵)*.
var:%f
mln:%d
}
```

```
RepeatItemMesg:
record
 erepeat_id: Repeat_ID
 which:     string(char)
 length:    int32
end
```

```
{RPT

rpt:%lu
wch:%s
len:%d
}
```

3.7 Sequencing Plate and Library Information

Sequencing plate messages and well messages convey information necessary to correct and detect plate

tracking errors that occur in the lab and/or data tracking system such as mislabeling and plate rotation. The sequencing plate message specifies the forward and reverse sequencing plate unique identifiers for the pair of plates derived from the same clone plate as well as the library that the clones on the clone plate were selected from. The well message specifies for each fragment which sequencing plate and which well on the sequencing plate the fragment was sequenced from.

The library message contains information about the library from which the fragment was derived. In particular, it contains the ID of the donor, thus aiding in polymorphism detection, and the distance record specifying the size distribution of the type of clone from which the fragment was derived. This will allow a fragment to be associated with a particular type of clone even if its mate is not present in the input.

Library, plate, and well messages do **not** need to be provided for fragments. However, if there is a well message for a particular fragment, it must reference valid and existing plate and library messages.

```
Library_ID: uint64
Donor_ID: uint64
Plate_ID: uint64
Well_ID: uint16
```

```
LibraryMesg:
Record
   eaccession: Library_ID
   donor: Donor_ID
   distance: Distance_ID
   source: string(char)
end
```

```
{LIB
acc:%lu
don:%lu
dst:%lu
src:↵(%[^\n] ↵)*.
}

acc: %lu

acc: %lu

acc: %lu
```

```
SeqPlateMesg:
Record
 eseq_plate_for:  Plate_ID
 eseq_plate_rev:  Plate_ID
 elibrary:        Library_ID
end
```

```
{SQP

spf:%lu
spr:%lu
lib:%lu
}
```

```
WellMesg:
Record
 efrag:       Fragment_ID
 eseq_plate:  Plate_ID
 ewell:       Well_ID
end
```

```
{WEL

frg:%lu
sqp:%lu
wel:%lu
}
```

4 Outputs

The assembler output consists essentially of the Extended Unitig Graph, the Extended Contig Graph and the Scaffold. The nodes of the unitig graph are output as Unitig messages, and the edges are output as UnitigLink messages. Similarly, the contig graph is output as Contig and ContigLink messages. The scaffold is repesented by a series of Scaffold messages. These basic snapshot components are accompanied by a) Augmented Fragment messages, which present the assembler's determination of each fragment's screen matches, mate status, chimeric status, chaff status, and clear range, and b) Mate Distribution messages, which summarize the mean and standard deviation of the mate distances *AS ASSEMBLED* (versus as input).

4.1 The Augmented Fragment Messages

Assembler annotation of individual fragments is provided by augmented fragment messages (`AFG`), emitted for each fragment that the assembler has processed. The accession indicates the Celera UID assigned to the fragment being augmented. Any Screen Matches found during the assembler's screening phase are provided, along with the assembler's evaluation of the fragment's mate status, chimerism status (all chimeric fragments will appear in singleton unitigs – currently set to 0), and chaff status (singleton fragments are chaff). (. The mate status field should be interpreted as follows.

- 'G' GOOD_MATE: the mate is confirmed by the assembly
- 'B' BAD_MATE: the mate is inconsistent with the assembly
- 'N' NO_MATE: no mate for the fragment was input into the assembler
- 'U' UNRESOLVED_MATE: mate is neither confirmed nor inconsistent with the assembly

The clear range field reflects any changes to the clear range made within the assembler.

```
AugFragMesg:                                    {AFG
record
 eaccession:     Fragment_ID                    acc:(%lu, %u)
 iaccession:    int32
 screened:      list of ScreenMatch             scn:↵<SMA-record>*.
 mate_status:   scalar (GOOD_MATE, BAD_MATE,
                 NO_MATE,UNRESOLVED_MATE)       mst:%1[GBNU]
 chimeric:      Boolean                         chi:%d
 chaff:         Boolean                         cha:%d
 clear_rng:     SeqInterval                     clr:%d,%d
end                                             }
```

Each Screen Match record provides the affected interval of the fragment ('where'), which Screen Item was matched ('what'), the repeat identifier and relevance (described in) and the matching interval and orientation within the Screen Item

```
ScreenMatch:                                    {SMA
record
 where:      SeqInterval in fragment            whr:%d,%d
 what:       Screen_ID                          wht:%lu
 repeat_id:  RepeatID                           rpt:%lu
 relevance:  int32                              rel:%d
 portion_of: SeqInterval                        pof:%d,%d
 direction:  scalar (AS_FORWARD,AS_REVERSE)     dir:%1[FR]
end                                             }
```

4.2 The Extended Unitig Graph

A Unitig message provides a unique accession number assigned by the assembler, and communicates several calculated statistics on the unitig, as well as its multi-alignment. The assembler passes along the unitig coverage statistic, which is used internally to determine whether a unitig should be classified as unique. The unitig status should be interpreted as follows.

- 'U' `AS_UNIQUE`: all unique unitigs (defined in terms of high coverage stat and length > 1kbp) and those repeat unitigs (all others) that are not placed in any scaffolds
- 'C' `AS_CHIMER`: represents a single fragment that has been deemed chimeric
- 'N' `AS_NOTREZ`: a repeat unitig that appears in only one scaffold
- 'S' `AS_SEP`: a repeat unitig that has surrogates (i.e., has instances in more than one scaffold)
- 'X' `AS_UNASSIGNED`: status as yet undetermined

An interesting branch point at either end of the unitig is output using the a_branch_point and b_branch_point fields. The magnitude of the field value specifies how far from the respective unitig ends the branch point is located, while the sign indicates the orientation of the branch points. A positive value indicates a branch from repeat into unique, while a negative value from unique into repeat. If no interesting branch point is detected (i.e., none within a certain fixed distance from the end, currently 1000 b.p.), the field will be set to zero.

The remaining fields provide the multi-alignment of the unitig, including the gapped consensus and quality strings (and their length), an indicator of whether any fragments were forced into the alignment, and the complete encoding of the multi-alignment as described in .

```
Chunk_ID: uint64

UnitigMesg:
record
 eaccession:      Chunk_ID
 iaccession       int32
 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 MultiPos
end
```

```
{UTG

acc:(%lu, %u)
src:↲(%[^\n]↲)*.
cov:%f


sta:%1[UCNSX]
abp:%d
bbp:%d
len:%d
cns:↲(%[^\n]↲)*.
qlt:↲(%[^\n]↲)*.
for:%d
nfr:%d
(<MPS-record>↲)*
}
```

```
MultiPos: record
 type: scalar (AS_READ,AS_EXTR,AS_TRNR,
```

```
{MPS
typ:%1[RXTEFUSLuB]
```


07/05/07 $Id: IOSpecDoc_Human.rtf,v 1.1.1.1 2004/04/14 13:41:57 catmandew Exp $ Page 12

```
                AS_EBAC,AS_FBAC,
                AS_UBAC,AS_STS, AS_LBAC,
                AS_UNITIG, AS_BACTIG)
  eident:        Fragment_ID

  source:       "description of data source"
  position:     SeqInterval
  delta_length: int32
  delta:        list of int32
end
```

```
mid:%lu

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

Each Unitig Link Message identifies a pair of unitigs and the orientation of the overlap between them. The orientation is encoded as a scalar with the following interpretation:

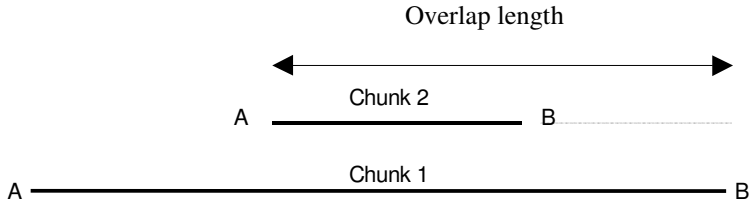
- 'N' Normal
- 'I' Innie
- 'A' Anti-Normal
- 'O' Outie

In the case of the containment overlaps, there are only two possible orientations for each overlap, anti-normal or innie (the unitigs are either aligned in the same direction, or in opposite directions).

The overlap_type specifies the relationship between the unitigs:

- 'N' No overlap
- 'O' Regular overlap
- 'T' Tandem overlap
- 'C' chunk1 contains chunk2
- 'I' chunk2 contains chunk1

For contained unitigs, the overlap distance is specified as if the contained unitig was extended past the B end of the containing unitig.



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 chunk and is also required for the chunks to overlap. If the edge includes a guide, the `includes_guide` is set to TRUE. The `mean_distance` and `std_deviation` fields describe the distribution of the edge distances represented in the link (a negative distance means the unitigs overlap). The number of edges (mates and potentially a chunk 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 link with respect to the assembly. Finally, the `jump_list` gives a list of all pairs

contributng to 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.

```
UnitigLinkMesg:                                    {ULK
record
 eunitig1:              Chunk_ID                   ut1:%lu
 eunitig2:              Chunk_ID                   ut2:%lu

orientation:          scalar {AB_AB, BA_BA,
                              BA_AB, AB_BA}        ori:%1[NAOI]
 overlap_type:         scalar (AS_NO_OVERLAP,
                        AS_OVERLAP,
                        AS_TANDEM_OVERLAP,
                        AS_1_CONTAINS_2,
                        AS_2_CONTAINS_1)           ovt:%1[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:%1[APBCU]
 jump_list:            list of Mate_Pairs          jls:↵(%lu,%lu,%c[MBSYT
                                                   ]↵)*
end                                                }


Mate_Pairs:
record
 in1, in2:             Fragment_ID
 type:                 scalar (AS_MATE,
                        AS_BAC_GUIDE,
                        AS_STS_GUIDE,
                        AS_MAY_JOIN,
                        AS_MUST_JOIN)              %d,%d,%1[MBSYT]

end
```

4.3 Extended Contig Graph

Contigs are ordered collections of Unitigs (and Surrogates, pending placement of all fragments) that cover contiguous regions of the genome. A contig is composed of fragments from the contained Unitigs, as well as fragment-like "surrogates", which are subsets of repeat Unitigs introduced to span gaps in contigs in the absence of complete repeat resolution. A Contig Message provides a unique accession number assigned by the assembler and a representation of the multi-alignment of the contig. The consensus sequence is **gapped**, that is, it will contain dash characters as needed to allow the alignment of all fragments. For each of the 'num_pieces' component fragments, a MultiPos record is given specifying its location in the multi-alignment and a delta encoding of the alignment to the consensus. Further, for each 'num_unitigs' component unitig, an UnitigPos record is given specifying the extent and multi-alignment of the unitig within the gapped consensus. The placed field indicates whether this contig appears in a subsequent

Scaffold message (AS_PLACED) or Degenerate Scaffold message (AS_UNPLACED). The type field in the UnitigPos indicates how this unitig was placed in the scaffold. For unplaced contigs, the unitig status will be AS_SINGLE.

```
ConConMesg:                                          {CCO
record
 eaccession:       Contig_ID                         acc:(%lu, %u)
 iaccession:       int32
placed:          scalar(AS_PLACED, AS_UNPLACED)      pla:%l[PU]
 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 MultiPos                    (<MPS-record>↲)*
 unitigs:         list of UnitigPos                   (<UPS-record>↲)*
end                                                  }
```

```
UnitigPos: record                                    {UPS
 type:            scalar (AS_UNIQUE,                 typ:%1[URSPs]
                          AS_ROCK, AS_STONE,
                          AS_PEBBLE, AS_SINGLE)
 eident:           Chunk_ID                          lid:%%lu
 position:        SeqInterval                         pos:%d,%d
 delta_length: int32                                  dln:%d
 delta:        list of int32                          del:↲((%d )*↲)*
end                                                  }
```

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

```
ContigLinkMesg:                                      {CLK
record
 econtig1:                Contig_ID                  co1:%lu
                                                     co2:%lu
 econtig2:                Contig_ID
 orientation:             scalar {AB_AB, BA_BA,      ori:%1[NAOI]
                          BA_AB, AB_BA}
 overlap_type:            scalar
(AS_NO_OVERLAP,                                      ovt:%1[NORT]
                          AS_OVERLAP,                ipc:%d
                          AS_TANDEM_OVERLAP)         gui:%d
 is_possible_chimera:     boolean                    mea:%f
 includes_guide:          boolean                    std:%f
 mean_distance:           float32                    num:%d
 std_deviation:           float32
 num_contributing:        int32
 status:                  scalar
(AS_IN_ASSEMBLY,                                     sta:%1[APBCU]
                          AS_POLYMORPHISM,
```

```
                                AS_BAD,AS_CHIMERA,
AS_UNKNOWN_IN_ASSEMBLY)
 jump_list:                    list of Mate_Pairs
end
```

```
    jls:↵(%lu,%lu,%c[MBSY
    ↵)*
}
```

4.4 The Scaffold

The Assembler's best choice for the scaffolds is output as "the" assembly. Each scaffold is given as a list of the pairs of adjacent contigs in the scaffold with the chi-squared estimate of the distance and standard deviation between the pair of contigs. The unitig and contig links supporting the scaffold are flagged by their AS_IN_ASSEMBLY status value.

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. A scaffold may consist of a single contig, in which case the num_contig_pairs will be zero and the id of the second contig will repeat the first, with arbitrary orientation and distance.

```
ScaffoldMesg:
record
 eaccession:          Scaffold_ID
 iaccession:          int32
 num_contigs_pairs:  int32
 contig_pairs:       list of ContigPairs
end
```

```
    {SCF

    acc:(%lu, %u)
    noc:%d
    (<CTP -record>↵)*
    }
```

```
ContigPairs:
record
 econtig1:            Contig_ID
 econtig1:            Contig_ID

 mean:               float32
 stddev:             float32
 orientation:        scalar {AB_AB, BA_BA,
                      BA_AB, AB_BA}
end
```

```
    {CTP

    ct1:%lu
    ct2:%lu

    mea:%f
    std:%f
    ori:%1[NAOI]
    }
```

The edges in the scaffold graph are represented by Scaffold Link edges, direct analogs of the Unitig Link Edges in the unitig graph. The main difference is in the objects being related.

```
ScaffoldLinkMesg:
record
 escaffold1:                Scaffold_ID
 escaffold2:                Scaffold_ID
 orientation:               scalar {AB_AB, BA_BA,
```

```
    {SLK

    sc1:%lu
    sc2:%lu
    ori:%1[NAOI]
```

```
                                BA_AB, AB_BA}
 includes_guide:               boolean
 mean_distance:                float32
 std_deviation:                float32
 num_contributing:             int32
 jump_list:                    list of Mate_Pairs
end
```

```
    gui:%d
    mea:%f
    std:%f
    num:%d
    jls:↵(%lu,%lu,%c[MBSYT
    ]↵)*

}
```

For the purposes of the viewer, unscaffolded contigs are also assigned a scaffold ID. This assignment is indicated in a Degenerate Scaffold message (DSC), specified as follows:

```
DegenerateScaffoldMesg:
record
 eaccession:          Scaffold_ID
 econtig    :         Contig_ID

end
```

```
    {DSC

    acc:%lu
    ctg:%lu

    }
```

4.5 Mate-Distance Distribution Messages

These messages are emitted to provide information on the distribution of mate lengths observed for those pairs whose mates lie in the same unitig or contig (and thus whose distance in the assembly is known precisely). For each mate-link distance record provided as input to the assembler with such a contributing pair, a message describing the observed distribution of corresponding mates in the current assembly is produced, with the 'refines' field referencing the original input record. The observed mean and standard deviation for mate pairs of this type in the current assembly are given, as well as the minimum and maximum distances observed for this type. Further, a histogram is provided of the number of contributing pairs within each distance subrange of the entire range from `min` to `max`.

```
MateDistMesg:
record
 erefines:     Distance_ID
 irefines:     int32
mean:         float
 stddev:       float
 min:          int32
 max:          int32
 num_buckets:  int32
 histogram:    list of int32
end
```

```
    {MDI

eref:(%lu, %u)   mea:%f
    std:%f
    min:%d
    max:%d
    buc:%d
    his:↵(%d↵)*

    }
```

5 Intermediate messages

The following table identifies each stage of the pipeline, and the component messages output from that stage. (The input is presumed to be the output from the previous stage.

Note: The definition of these messages is for pipeline design purposes. The content of these messages may be passed in containers other than the messages defined here (e.g., indexed files).

Input .frg	Gatekeeper .inp	Screener .urc	Overlapper .ovl	Unitiger .utg	Scaffolder .cgw	Terminator .asm
BAT	IBA	IBA	IBA	IBA		
ADT	ADT	ADT	ADT	ADT	ADT	ADT
FRG	IFG	SFG	OFG		IAF	AFG
LKG						
DST	IDT	IDT			IMD	MDI
BAC	IBC	IBC				
BIN	IBI	IBI				
SCN	ISN		OVL	IUM	IUM	UTG
RPT	IRP			UOM	IUL	ULK
					ICM	CCO
					ICL	CLK
					ISF	SCF
					IDS	DSC

5.1 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 – BAT, FRG, LKG, SCR, BAC, BIN and DST -- with internal ID assignments and passes them on as IBA, IFG, ILK, ISN, IBA, IBI 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 “(%lu,%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 accession numbers encoded in 3-code format as "%d" with a suitably modified field name, ("xyz" becomes "ixz", for example).

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

5.2 Repeat Tagger/Contaminant Screener

The URT/URC module consumes ISC messages, passes IDT messages through unaltered, 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 component IntScreenMatch records are the internal equivalents of the output ScreenMatch records described in .

```
IntScreen_ID: int32

ScreenedFragMesg:                                    {SFG
Record
 "As Before"
 clear_rng:   SeqInterval
 screened:    sorted list of IntScreenMatch          scn:↵<ISM-record>*.
 "As Before"
end                                                  }
```

5.3 Overlapper

The Overlapper 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 message the 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.

```
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                                                  }
```

5.4 Unitigger

See output messages for description of the Unitig message. The internal version merely substitutes an internal ID for the Celera UID, except for IntMultiPos records, which also include a "contained" field to specify the containment relationship between a contained fragment and it's parent in the reduced graph. A fragment that is not contained has a zero in this field.

A series of independent UnitigOverlapMesg messages specify chunk graph edges. Each edge identifies a pair of chunks and the orientation of the overlap between them. Details of the orient and ovrelap_type fileds are identical to the UnitigLink description of . The UnitigOverlapMesg also provides the best, minimum, and maximum overlap length between the pairs of unitigs. The minimum and maximum equal, except in the case of edges induced by a tandem repeat, where the extent of possible 'slippage' in the overlap is thus indicated. Note that not every tandem edge will be detected by overlap alone, and the CGB will transitively infer when the overlapping parts of an edge involve tandem satellites. This inference will be reflected in the `overlap_type` field. In this internal context, the `AS_NO_OVERLAP` value is not exercised.

```
UnitigOverlapMesg:
record
 chunk1:               IntChunk_ID
 chunk2:               IntChunk_ID
 orient:               scalar (AB_AB, BA_BA,
                               BA_AB, AB_BA)
 overlap_type:         scalar (AS_NO_OVERLAP,
                               AS_OVERLAP,
                               AS_TANDEM_OVERLAP,
                                AS_1_CONTAINS_2,
                                AS_2_CONTAINS_1)
 source:               "description of data"
 best_overlap_length:  int32
 min_overlap_length:   int32
 max_overlap_length:   int32
 quality:              float32

end
```

```
{UOM

ck1:%d
ck2:%d

ori:%1[NAOI]



ovt:%1[NOTCI]
src:↲(%[^\n]↲)*.
len:%d
min:%d
max:%d
qua:%f
}
```

5.5 Scaffolder

The scaffolder reads Internal Unitig Messages, and outputs an internal representation of the extended unitig graph, the extended contig graph and the scaffold, as well as mate distance summaries. The internal versions of these messages correspond directly to their external counterparts.

6 Feedback Messages

The main purpose of feedback messages is to provide a mechanism for the Overlay Assembly Team to request changes in the input data as errors in the data are detected during the course of assembly. The changes would then be made by the Pre-Assembly Team and updates sent to the Overlay Assembly Team using existing mechanisms for updating and adding input data.

In order to avoid polluting the already-overflowing 3-code name space, there is a single message type to encompass all edits. This message, known as a Batch Update Generator (BUG) Message, and its variants

are described below.

```
#define AS_MSG_NO_BAC   0

BugMesg:                                                {BUG
record
   type: scalar ( AS_CREATE_BAC, AS_REMOVE_BACTIG,      typ:%1 [CRASGrs]
                  AS_ADD_BACTIG, AS_SPLIT_BACTIG,
                  AS_CHANGE_CLR_RANGE, AS_REMOVE_CONTIG,AS_SPLIT_CONTIG )
   source: string(char)                                 src:↵(%[^\n] ↵)*.
   variant of type:
      AS_CREATE_BAC:
         bac_eaccession: Locale_ID                      bac:%lu
         seq_eaccession: Locale_ID                      seq:%lu
      AS_REMOVE_BACTIG:
         bac_eaccession: Locale_ID                      bac:%lu
         seq_eaccession: Locale_ID                      seq:%lu
         bactig_eaccession: Fragment_ID                 btg:%lu
      AS_ADD_BACTIG:
         src_bac_eaccession: Locale_ID                  sbc:%lu
         src_seq_eaccession: Locale_ID                  ssq:%lu
         dst_bac_eaccession: Locale_ID                  dbc:%lu
         dst_seq_eaccession: Locale_ID                  dsq:%lu
         bactig_eaccession: Fragment_ID                 btg:%lu
      AS_SPLIT_BACTIG:
         src_bac_eaccession: Locale_ID                  sbc:%lu
         src_seq_eaccession: Locale_ID                  ssq:%lu
         bactig_eaccession: Fragment_ID                 btg:%lu
         num_pos: int32                                 nps:%d
         split_array: list of BugSplitPos               (<BSP-record>↵)*
      AS_CHANGE_CLR_RANGE:
         old_frag_eaccession: Fragment_ID               old:%lu
         new_frag_eaccession: Fragment_ID               new:%lu
         clear_rng: SeqInterval                         clr:%d,%d
   end
end                                                     }

BugSplitPos:                                            {BPS
record:
   position: SeqInterval                                pos:%d,%d
   bactig_eaccession: Fragment_ID                       btg:%lu
end                                                     }
```

The AS_CREATE_BAC operation creates a new BAC of type AS_UNFINISHED with 0 bactigs. The source field for the new BAC contains the source field of the BUG message, and hence can be used to indicate the reason for the BAC's creation. The accession number of the new BAC and its associated sequence are pre-specified in the bac_eaccession and seq_eaccession fields respectively of the BUG message.

The AS_REMOVE_BACTIG operation removes a bactig from a particular BAC. The fragment message for the bactig is unchanged, but the source field is appended to the source field for the BAC. It is the responsibility of the Assembly Team to ignore removed bactigs, which can be identified from the fact that they do not appear in the bactig list of any BAC message.

The AS_ADD_BACTIG operation adds a bactig to a particular BAC, specified by dst_bac_eaccession and

dst_seq_eaccession. The bactig must be pre-existing, but its associated fragment message does not need to be part of the input stream. If the bactig is currently part of another BAC, then src_bac_eaccession and src_seq_eaccession are set to the accession of that bac and its sequence. Otherwise, both fields are set to AS_MSG_NO_BAC. Both the BAC message and the bactig's fragment message are updated to reflect the new assignment.

The AS_SPLIT_BACTIG operation splits a bactig into two or more new bactigs. The original bactig is implicitly removed from its associated BAC, which is specified by src_bac_eaccession and src_seq_eaccession, and the new bactigs are inserted in its place. The number of splits to be performed is given by the num_pos field. For each split, there is a pair of entries in the split_array field. More specifically, if the bactig is to be split into intervals *p(0)-q(0), p(1)-q(1), ..., p(M)-q(M)*, where *p(0) < q(0) ,... p(M)<q(M)*, then the resulting bactigs will contain the sequences [*p(0), q(0)*], [*p(1), q(1)*], ..., [*p(M), q(M)*]. The accession numbers of the bactigs are given in the split_array entries.

Finally, the AS_CHANGE_CLR_RANGE operation sets the clear range of a specified fragment to a new value. As part of making the change, the old fragment is deleted (meaning that an AS_DELETE FragMesg must be sent to the Assembly Team) and is supplanted by a new fragment whose ID is specified by the new_fragment_eaccession field. The new fragment is identical to the old save the change in clear range and the addition of the BugMesg's source field to its own.

BUG messages are periodically be batched together into a single file by members of the Assembly Team, who will then send the file to Pre-Assembly for processing. The file must be processed sequentially so that any operation can refer to the results of a preceding operation. Pre-Assembly then sends back the processed data in the form of incremental updates to the Assembly Team using currently-existing ProtoIO messages.

Another use for the BUG messages arises within the frame work of the regional assembler. Here, an internal version of the BUG message, called IBG message , is used to edit contigs. The two types AS_SPLIT_CONTIG and AS_REMOVE_CONTIG are used to indicate that the objects that are being split or removed are not bactigs but rather contigs.

Currently, IBG messages are only implemented under CA but not AS.

A. Input Examples

```
{BAT
bna:celsim batch 0
crt:952112525
acc:0
com:
(No comment)
.
}
{ADT
{ADL
who:Celsim 1.54 (gaussian)  2000/02/22
ctm:952112525
vsn:
com:
.
}
.
}
```

```
{DST
act:A
acc:1
mea:1774.000000
std:191.333328
}
{BAC
act:A
bid:10000000002
typ:E
etm:0
len:1
src:
BAC for Bac Ends 10000000002
.
}
{FRG
act:A
acc:2
typ:E
loc:10000000002
src:
3f
[7347,6897]
J.0.1 (2) [873,1323] [450,0]
.
etm:0
seq:
cccagattccttccctctgctgcggaccacagtagttttcatgtgcagttacacccactgaaggccattc
tcacattaagcatgcaccttatccaggagatgtgcgcacggttgcgaagattctgcggcgccgggggtta
ataccattagacatgaatcaagcggtgtaaagcgtgtcgaagctcagaataaagatgtttaatgccagac
cacgtgtggagtagtgttgtgtattactaacgcgagtaactagacaggcatcgagtgcttccggtagaga
ggagttattataaagtagatagatagtcatgaaggagagcattattctggcctatactgttttaattgcg
tacggtccaacacgtaccttctggagccggtgtgaccgtaaacgattaagtgtgataatgggagccgctt
tcacaatgccccctgcatccgcgaagac
.
qlt:
JJJJJJFJJSTJPJOJTJJDJ8JSJJJEJIJ9JJJIEJJ@9OJ6JJJJJJIJJJ86JJJJJJJJJPJJFJ
JJSJJQJJJJJJJJJJJJQJJJSJJJJJJJJJJSJNJCBJJJFJJJJJJJJJJJJJJJJJJJJJJJJJJ
;JJJJMJJ>JJJJJJ>JJJJJJJJJJJJJJJ96JJJJ7JJJOJJJJJ6JJJJJJJJJJJE:JJ>JJJJJJ
JJQJJJJJJQJFJJBJIJJ9IJJJJJPJOJJJJDDIJJJ=OJP>JJJJJJJEJJJJJJJJJJJJJJ<@J
GJJJTJKJJJGJJJ5JJJJJEJJ7JJAJJJJJJLJJIJJJJJJJJJJJF@JJJJKJJGJHHJJJ78J85J
JJJJLJJJJDJJJJ4JJJJJJJJ:JKJQJJJ6JJJJJJJJJDJQJAJJ7JFJT;JMJPJJJJJJ<JJJJLJ
JJPJJJJJJJJJ>JJJJJJJJJJ=JJJJ
.
clr:0,448
}
{FRG
act:A
acc:3
typ:E
loc:10000000002
src:
3r
[5515,6135]
B.0.3 (14) [16,281] [0,265]
J.0.1 (2) [0,111] [509,620]
.
etm:0
seq:
atcgagtctcaactccttgagctggaattatcgtccacaacgctagagatgcaccgcggtaacctgtcta
cgtagtcacaccacgcgggccgatgaggtacttgacgaccggcacctgtcaccttcttaatatcgtattg
agagttaatacgcgctttgcgcctgcatcttattcgccgcagagcagcactacaccccgctctaacgtgg
atctacattcaggccggtcgcttgtaataatttgatgcccgactagtccgggtctatccctgtgctggta
acggttgatcccgcaagtaactcgtctcaaagatcaatattacacagagatgccagctcgtttgggctaa
tggacagttacaaaagagagaggtgtcagggtgtggccacgcgctttagatatggctacccgcttcacat
taacctgatcaataggcattagatgcgggtacagtcacaagccgattctaacaatttataccacaatcag
cgcagacacccgttcattttaacgcctagatcgcttctggtcgctcaaccataatccggtcatgaagtgg
```

```
gggaacaagatctgtaagaaatccggagtctggctcgacccatcggcctggagatct
.
qlt:
JFJJJJ@JJ>JJGPJ>JJCJJJJJJAJJJJHJ7JSJMJJJJJJJJJJ<J<JJJJJJJJ95JJJJJJJJJC
JJBJJJ6OJJJJJ?JJJJJJ>JJJ;JJJJ8JJJJJJJJJJQJJJJJJTJJJJJBJDJJEJJRAJ@JJ9JJ
JJJB=JJJJJJJJ=JJJJJJJJJJJJARJJJJJ8JDJJJJKJ5PJJJJ;JJ@JJ?JQJJTJJJJJGJJ=J
JJJJ<J<J=JJJJJRJJ5JJJJJ=<;JDJJJJJJJ<MJGJJJJJJJRJJ5J>EJ<JJJJJJJ8QJJJJJJF
JJJJJBJ>J<JJJIQJ?JJJJJJJNJMJJJJJ:JJJJJPJJJNCJ6JJJJEJ:<JJJJJJIJJJJIJJJJ
J<<J?JBJJJJJJ7JJJJJ8JJCJJJ?@JPJJJJJOJPICJJIJHTJLJAJ?JJJ?FJJJJJJJJJJGJJ
>JJJJJJJJJFJJJJGJJJJJHHJJQLJJJJJAJJJJJ7JJ@JDJJJGJJJJJJJJJPJJ7JJJJJG<JJ
<5JJJJJN?JJJJJJ6TJJJJJJJJNJJJJJJJJJJJJJL>JJJ@JJJTJDLJE;JJJGLJGJJJFJ86J
JMJJJ?JJCJJJJJ6RJSJJJJJJJDJJRJPJJJN4JJ<TJJJ?KJJRCJJJ=OJHJ
.
clr:0,617
}
{LKG
act:A
typ:B
fg1:2
fg2:3
etm:0
dst:1
ori:I
}
```

B. Output Examples

```
{ADT
{ADL
who:Celsim 1.54 (gaussian)  2000/02/22
ctm:952112525
vsn: $Id: IOSpecDoc_Human.rtf,v 1.141.15 2000/03/24 17:58:14 skravitz20:16:07 karin
com:
.
}
{ADL
who:/home/skravitz/humanSrc/AS/bin/gatekeeper
ctm:953754213
vsn:(blank)
com:



Complete call: /home/skravitz/humanSrc/AS/bin/gatekeeper -Q -P -f a006.Store a006
Started: Wed Mar 22 14:43:33 2000
Working directory: /data/assembly1/skravitz/a006
.
}
.
}
{MDI
ref:(1,1)
mea:1725.8
std:178.37
min:1250
max:2098
buc:15
his:
2
1
4
2
3
9
7
14
14
11
8
8
9
3
2
}
{AFG
acc:(22,21)
scn:
.
mst:N
chi:0
clr:0,654
}
{AFG
acc:(57,56)
scn:
.
mst:U
chi:0
clr:0,487
```

```
}
{AFG
acc:(94,93)
scn:
.
mst:N
chi:0
clr:0,636
}
{AFG
acc:(105,104)
scn:
.
mst:N
chi:0
clr:0,533
}
{AFG
acc:(2,1)
scn:
.
mst:U
chi:0
clr:0,448
}
{AFG
acc:(20,19)
scn:
.
mst:G
chi:0
clr:0,668
}

{UTG
acc:(4711,0)
src:
gen> uu [7631,6842]
.
cov:2.000
sta:N
abp:0
bbp:0
len:809
cns:
GTTGCGAACGTGATTCCCTATACCCGTACCAGAATATCCAGGTGATGTACTCTTTCCGTGGTCGCACTTT
AGTAAAAAATCACTTGATCGGCTGTAGAGCGCTCTGTGCTGCACTACGTACTTACCCCAGGATGAGGTTC
TGGGCTAATCAACTTGCAATGTGCAATCGCCAGGGTAGTACAATCGTTGCATCCTAGTAAAACAACTCTT
GTTCCCTCATGGACGTGTTTTCGATACAGGATGTGTCCAACCGGGTGATGTGACTGTTGTCCGGGGAAGC
GATGCTCTCGCCCCAGATTCCTTCCCTCTGCTGCGGCACCACAGTAGTTTGTCGTGTGCAGTTACACTCA
CTGAAGGCCATTCGTCACATTTAAGCATCGCACCCTATATCCAGGAGATGTGCGCACGGTGTGCGAAGAT
TCTGCGGTGCCGGGGGTTAATACCATTAGACATGAATCATAGCGGTGTAAAGCGTTGTCGAAGCTCAGAA
TAAAGATGTTTAATGCCAGACCACGTGTGGAGTAGTGTTGTGTATTACTAACGCGAGTAACTAGACAGGC
ATCGAGTGCTTCCCGGTAGAGGAGGAGTTATATATAAAGTAGATAGATCAGTCATGAAGGAGAGCATTAT
TCTGGCCTATACTGTTTTAATTGCGGTACGGTCCAACACGTACCTTCTGGAGCCGGTGTGACCGTAAACG
ATTAAGTGTGATAATGGGAGCCGCTTTCACAATGCCCCCTGCATCCGCGAAGACCGGATGTGGCGGGAAT
TGAGAATAGAAAATGGTGCACGCCCCGAACCGATGTCTC
.
qlt:
JJJJ5DAOJJJJJJJGCJFJJJJJJJJJJJJJJJJJNJDJJSJJJJJ?J?JQJJJJJJJJJJJJO@FJJ
LJLJJJJ>OJ??JJJDJJJJJJJS>RJJ=RIJJJJJEJJJJ>J?JJF>JJJJJBJJJJJJJJJJJJJAH
GQCJJJJ4JQJ;JJM<JJFJJJD@J6JJSJJDRLJJJJJJAJJHJJJJJJJJJJLOJJJJ:NJ7@JJJJ8
JJJJJAJRJDJJJ?JJJJJJJOJQRJJJ6JJJJJE8:JRM58JTOJEJJJJJ<JNJRJJJQJ;JJJ9JJJ
JJJ==JJJJJ4JJSDJBJNJGJRJJJ=JJJJJJJ=J5JJEJJ=J8JJ9JJ5J;JJJJKJJJJJ5HJJJJ:
>J9JJJJCJJJJT4J=JJJIJJJJIJJJ4JJJJJJJ:J:JJJMLJJRJJJJJJJJOJILJ9JJLJ@9JJ9
JEJJQJT6JNJJJJJJJJJJJJJ>OJJJJJJJJJJJ5JJ:JJTJJJ8JJJJJJJJ4J=SJJJJJJJJJJJ
J=JJIJJJJJJJJJJJJJ:;9JJJJGQJJ;JKDJQJJQJJRJJJ@JJPJJIJJJJJJJJJJJJFJJJGJJJ
JJ6JJ4MJJJJJJJJJJJJJ5JBFSJ@JJJJ9JJJJJJJJJJJ=JJJJJJJNJJJJ4JJJOJJHJJPJJL
```

```
JG6JJJJJJJJFJ9JJSJJNJPJJJJOJJ5JJJJJJ7JJJJJJJJJ9JJCJJ9J?JJJJJJJ<@MJOJG
J@JFJ?JJJ7JJJJ>JJJJNHJJJJJJJJJHJJCB6P8JJ@JJRJDO>JJJQJJJJJJJT=JJBJJJMJ
JJGJJJJJJJ>JJJJJJJS5JJJMJJJJJJJJJJJJTJ?
.
for:0
nfr:5
{MPS
typ:E
mid:22
src:

.
pos:0,669
dln:15
del:
192 192 218 235 286 325 357 362 370 377 382 399 406 446 640
}
{MPS
typ:E
mid:57
src:

.
pos:100,603
dln:16
del:
103 175 182 196 212 225 232 256 261 301 349 365 447 458 466 476
}
{MPS
typ:E
mid:94
src:

.
pos:148,798
dln:14
del:
55 89 126 133 138 163 215 223 230 317 413 432 475 493
}
{MPS
typ:E
mid:105
src:

.
pos:260,809
dln:16
del:
16 23 28 53 66 98 112 119 142 191 204 300 309 318 352 394
}
{MPS
typ:E
mid:2
src:

.
pos:291,754
dln:15
del:
25 38 70 75 83 86 89 112 161 174 270 279 288 304 349
}
}
{ULK
ut1:4711
ut2:4723
ori:N
ovt:N
ipc:0
```

```
gui:1
mea:-57.000
std:191.458
num:1
sta:U
jls:
57,56,B
}
{ULK
ut1:4711
ut2:4723
ori:N
ovt:N
ipc:0
gui:1
mea:47.000
std:191.451
num:1
sta:U
jls:
2,3,B
}
{ULK
ut1:4712
ut2:4723
ori:O
ovt:N
ipc:0
gui:1
mea:67.000
std:191.449
num:1
sta:U
jls:
271,272,B
}
{CCO
acc:(4873,0)
pla:U
len:809
cns:
GTTGCGAACGTGATTCCCTATACCCGTACCAGAATATCCAGGTGATGTACTCTTTCCGTGGTCGCACTTT
AGTAAAAAATCACTTGATCGGCTGTAGAGCGCTCTGTGCTGCACTACGTACTTACCCCAGGATGAGGTTC
TGGGCTAATCAACTTGCAATGTGCAATCGCCAGGGTAGTACAATCGTTGCATCCTAGTAAAACAACTCTT
GTTCCCTCATGGACGTGTTTTCGATACAGGATGTGTCCAACCGGGTGATGTGACTGTTGTCCGGGGAAGC
GATGCTCTCGCCCCAGATTCCTTCCCTCTGCTGCGGCACCACAGTAGTTTGTCGTGTGCAGTTACACTCA
CTGAAGGCCATTCGTCACATTTAAGCATCGCACCCTATATCCAGGAGATGTGCGCACGGTGTGCGAAGAT
TCTGCGGTGCCGGGGGTTAATACCATTAGACATGAATCATAGCGGTGTAAAGCGTTGTCGAAGCTCAGAA
TAAAGATGTTTAATGCCAGACCACGTGTGGAGTAGTGTTGTGTATTACTAACGCGAGTAACTAGACAGGC
ATCGAGTGCTTCCCGGTAGAGGAGGAGTTATATATAAAGTAGATAGATCAGTCATGAAGGAGAGCATTAT
TCTGGCCTATACTGTTTTAATTGCGGTACGGTCCAACACGTACCTTCTGGAGCCGGTGTGACCGTAAACG
ATTAAGTGTGATAATGGGAGCCGCTTTCACAATGCCCCCTGCATCCGCGAAGACCGGATGTGGCGGGAAT
TGAGAATAGAAAATGGTGCACGCCCCGAACCGATGTCTC
.
qlt:
JJJJ5DAOJJJJJJJGCJFJJJJJJJJJJJJJJJJJJNJDJJSJJJJJ?J?JQJJJJJJJJJJJJJO@FJJ
LJLJJJJ>OJ??JJJDJJJJJJJJS>RJJ=RIJJJJJEJJJJ>J?JJF>JJJJJBJJJJJJJJJJJJJAH
GQCJJJJ4JQJ;JJM<JJFJJJD@J6JJSJJDRLJJJJJJAJJHJJJJJJJJJJLOJJJJ:NJ7@JJJJ8
JJJJJAJRJDJJJ?JJJJJJJOJQRJJJ6JJJJJE8:JRM58JTOJEJJJJJ<JNJRJJJQJ;JJJ9JJJ
JJJ==JJJJJ4JJSDJBJNJGJRJJJ=JJJJJJJ=J5JJEJJ=J8JJ9JJ5J;JJJJKJJJJJ5HJJJJ:
>J9JJJJCJJJJT4J=JJJIJJJJIJJJ4JJJJJJJ:J:JJJMLJJRJJJJJJJJOJILJ9JJLJ@9JJ9
JEJJQJT6JNJJJJJJJJJJJJJ>OJJJJJJJJJJJ5JJ:JJTJJJ8JJJJJJJJ4J=SJJJJJJJJJJJ
J=JJIJJJJJJJJJJJJJ:;9JJJJGQJJ;JKDJQJJQJJRJJJ@JJPJJIJJJJJJJJJJJFJJJGJJJ
JJ6JJ4MJJJJJJJJJJJJJ5JBFSJ@JJJJ9JJJJJJJJJJJ=JJJJJJJNJJJJ4JJJOJJHJJPJJL
JG6JJJJJJJJJFJ9JJSJJNJPJJJJOJJ5JJJJJJ7JJJJJJJJJ9JJCJJ9J?JJJJJJJ<@MJOJG
J@JFJ?JJJ7JJJJ>JJJJNHJJJJJJJJJHJJCB6P8JJ@JJRJDO>JJJQJJJJJJJJT=JJBJJJMJ
JJGJJJJJJJ>JJJJJJJS5JJJMJJJJJJJJJJJJTJ?
.
```

```
for:0
npc:5
nou:1
{MPS
typ:E
mid:22
src:
23
[7631,6982]
J.0.1 (2) [958,1607] [649,0]
lab>TE
.
pos:0,669
dln:15
del:
192 192 218 235 286 325 357 362 370 377 382 399 406 446 640
}
{MPS
typ:E
mid:57
src:
23
[7631,6982]
J.0.1 (2) [958,1607] [649,0]
lab>TE
.
pos:100,603
dln:16
del:
103 175 182 196 212 225 232 256 261 301 349 365 447 458 466 476
}
{MPS
typ:E
mid:94
src:
23
[7631,6982]
J.0.1 (2) [958,1607] [649,0]
lab>TE
.
pos:148,798
dln:14
del:
55 89 126 133 138 163 215 223 230 317 413 432 475 493
}
{MPS
typ:E
mid:105
src:
23
[7631,6982]
J.0.1 (2) [958,1607] [649,0]
lab>TE
.
pos:260,809
dln:16
del:
16 23 28 53 66 98 112 119 142 191 204 300 309 318 352 394
}
{MPS
typ:E
mid:2
src:
23
[7631,6982]
J.0.1 (2) [958,1607] [649,0]
lab>TE
.
```

```
pos:291,754
dln:15
del:
25 38 70 75 83 86 89 112 161 174 270 279 288 304 349
}
{UPS
typ:s
lid:4711
pos:0,809
dln:0
del:
}
}
{CLK
co1:4873
co2:4885
ori:N
ovt:N
ipc:0
gui:1
mea:-57.000
std:191.458
num:1
sta:U
jls:
57,56,B
}
{DSC
acc: 54786
ctg: 4873
}
{SCF
acc:(5034,0)
noc:4
{CTP
ct1:4878
ct2:5033
mea:440.207
std:339.232
ori:O
}
{CTP
ct1:5033
ct2:4887
mea:2233.627
std:378.697
ori:I
}
{CTP
ct1:4887
ct2:4885
mea:1537.473
std:473.362
ori:O
}
{CTP
ct1:4885
ct2:4893
mea:344.179
std:187.479
ori:N
}
}

{ISL
sc1:5
sc2:6
ori:A
```

```
gui:1
mea:992.000
std:1125.097
num:1
jls:
282,281,B
}
```