APPENDIX E
E

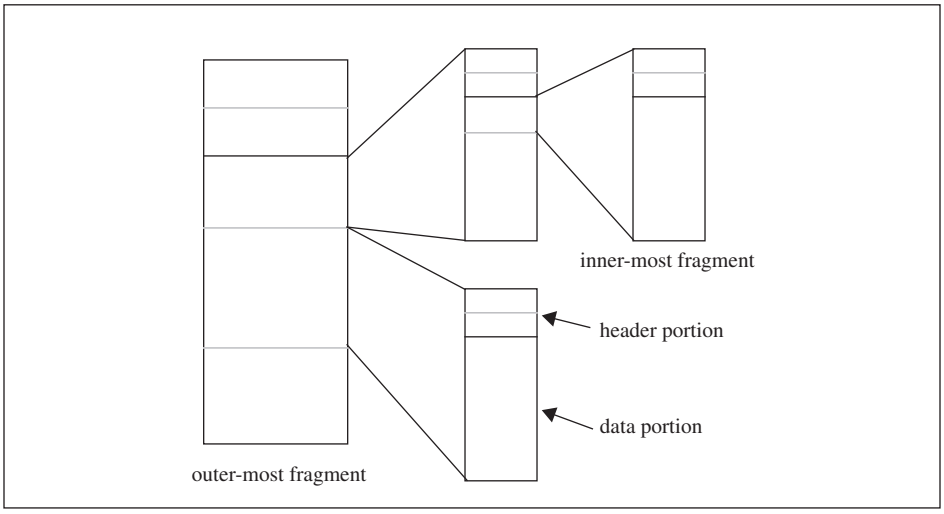
CEBAF Common Event Format

The CEBAF Common Event Format was designed to meet the following requirements:

- a partitioned format in which each fragment contains a header portion and a data portion.
- a recursive format to support complex structures (a tree). That is, fragments may contain other fragments within their data fields.
- a length field in each fragment header.
- a "type" field in each fragment header giving the data type of the data for that fragment. This allows for data conversion from one machine architecture to another, and marks each fragment as a branch or leaf node in the tree.
- a "tag" field in each header to indicate the source or purpose of the data contained in that fragment.
- a small fragment header.

This structure is shown in the following figure:

FIGURE 3 Event with fragment depth of three.



The last requirement above (low overhead) makes it difficult to derive a single header format. In some situations it may be desirable to have a header with a lot of identifying information and a large length field. In other situations, only minimal length and tagging information is needed.

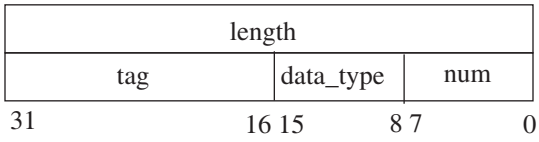
As a result of these conflicting requirements, 3 header formats of differing sizes will be used. The corresponding fragments, in decreasing order of header size and functionality, will be referred to as "banks", "segments", and "packets".

E.1 Event Format

Banks

Following the nomenclature of other high energy physics formats, the first and largest constituent will be referred to as a "bank". The bank header will consist of 2 longwords in the following format:

FIGURE 4 Bank Header Format



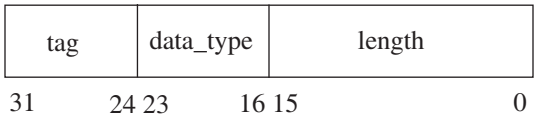
The bank "length" is the length of the fragment in longwords, excluding the length word (i.e. length is the number of words to follow); "tag" is an integer identifier which may be used as an index into an optional dictionary of bank and segment names and titles (described later); "data_type" is an integer giving the type of data in the data portion of this bank, and "num" is an integer which may be used as a bank number or may be used to encode additional information about the bank.

Each event (at its outermost level) will be a single bank in which the bank length is the event length; for this reason, the minimum overhead per event is 8 bytes. The tag and num fields together can be used to specify the type of event. For example, physics events will have a tag of 0xCEBA and a num field equal to the readout code number.

Segments

Applications requiring complex structure with less overhead per fragment can use a smaller fragment type called a "segment". The segment header will consist of 1 longword in the following format:

FIGURE 5 Segment Header Format



The major differences between bank and segment headers are the maximum fragment length allowed, and the number of available tags; there is also no "num" field in a segment header, so tags need to be unique. The segment length is still given in longwords (unsigned, up to 1/4 Mbyte, and excluding the header word), and the tag field (unsigned) takes on the values 0-255 vs. 0-65535.

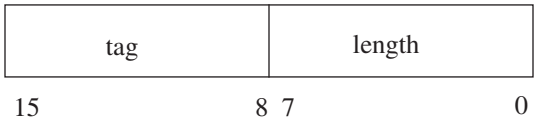
Padding of a bank or segment may be necessary to bring it up to a multiple of 4 bytes; it is up to the application to determine the actual end of the data (it should be obvious from context).

Note that the bank and segment headers must be treated as longwords for the purposes of converting between big and little endian machines.

Packets

For data structures with even less overhead per fragment, a third fragment type, the "packet", may be used. The packet header will consist of a single 16 bit integer as shown in the following format:

FIGURE 6 Packet Header Format



Packets contain data items with widths of no more than 16 bits, and the length field is in units of 16 bit words. Packets cannot be recursive as there is no "data_type" field to indicate when the recursion should stop. They are intended to encode small arrays tagged by a small number (0-255). Unlike banks and segments, packets may start on odd word boundaries.

Data Types

The following is a preliminary list of defined data types (in hex):

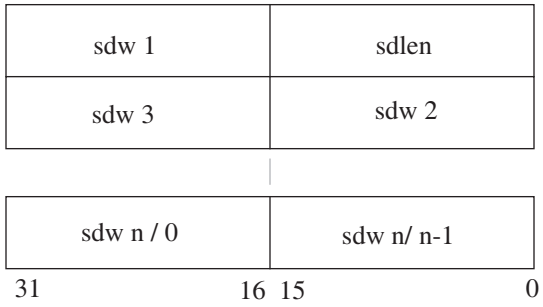
0 = unknown
1 = long (32 bit) integer
2 = IEEE floating point
3 = null terminated ASCII string
4 = 16 bit signed integer
5 = 16 bit unsigned integer
6 = 8 bit signed integer
7 = 8 bit unsigned integer
8 = double precision IEEE floating point
9 = VAX floating point
A = VAX double precision floating point
F = repeating structure
10 = bank
20 = segment
30 = packet with data_type = 0
33 = packet with data_type = 3
34 = packet with data_type = 4
35 = packet with data_type = 5
36 = packet with data_type = 6
37 = packet with data_type = 7

Complex Structures

In applications requiring mixed data types (integer particle ID and real particle energy, for example), banks or segments may contain "repeating structures". In these banks

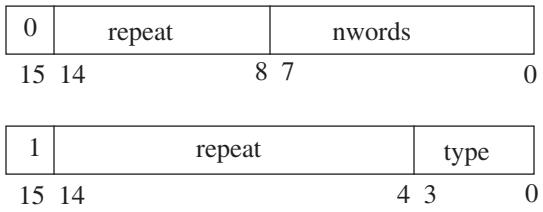
(segments), the first few words of data are the structure description, and the remaining data words are data items of that structure. The structure description has the following form:

FIGURE 7 Structure Description Format



where “sdlen” gives the length of the structure description in longwords, “sdw i” is a 16 bit structure description word in one of the following formats:

FIGURE 8 Structure Description Word Format



The basic idea is to encode a format descriptor like (4I,4F,5(1I,1F),1F). The first form (high bit zero) is used to encode an open parenthesis: “repeat” gives the repeat count of the parenthesized expression, and “nwords” gives the number of following descriptor words until the parenthesis closes. The second form (high bit set) is used to encode a repeated field of data type given by the low 4 bits. Only types 1 through 8 are currently allowed. Since the entire structure description must be a whole number of longwords, it may be necessary to pad the structure with a null descriptor, which is ignored. A repeat count of zero is not allowed; an implicit repeat is performed on the whole structure until the end of the bank or segment is reached.

Another condition to keep in mind when designing structures is to make them multiples of the largest sized data item because some architectures can only access data items on natural boundaries (i.e. longwords on 4 byte boundaries, short words on 2 byte boundaries, doubles on 8 byte boundaries).

One application of structures is as an alternative to packets. For example, consider the problem of encoding the left and right TDC hits of a wire along with the wire number, where the data is 16 bits. Using packets would require 3 words per wire, (using the tag field to give the wire number) but would be limited to 256 wire numbers. Alternatively, a structure of the form (3I), in which the first word is the wire number and the next 2

words are the data, allows 65536 wire numbers and still only uses 3 words per wire hit (plus 2 words of structure definition).

E.2 Physical Record Format

Events (logical records) will be packed into fixed sized physical records. Each physical record will have an 8 longword header as follows:

BLKSIZ blocksize (in longwords)

BLKNUM block number

HDLEN header length (offset to data)

START offset to first start of logical record in block

END number of valid words (header + data) in block

VER header version number (=1 in CODA 1.0)

(reserved, =0)

(reserved, =0)

All lengths and offsets are in units of longwords, and offsets are relative to the start of the block (i.e. offset=8 points to the start of the data area). Blocksizes must be multiples of 256 longwords, and no greater than 32768 longwords (i.e. 1 to 128 Kbytes). With this convention, the "blocksize" word should only have non-zero data in bits 8-15, and can be used to detect any byte swapping problem (big/little endian).

Note that if a logical record spans 3 physical records, the middle physical record will have START=0.

E.3 Name Dictionary

The name dictionary is a simple ASCII file, with at most one name entry per line. Curly brackets { } will be used to indicate the beginning and end of a set of names of sub-fragments.

Each definition line will contain the following:

- index / tag value in hex
- fragment name (ASCII)
- fragment title (ASCII)

The index value for a bank must be in the range 0000 to FFFF and for a segment it must be in the range 00 to FF. The fragment name must contain only alphanumeric characters, and names are case insensitive. The title may contain any printable character. The three fields are delimited by any amount of white space (space or tab), and the title starts at the first non-space character after the name and continues to the end of line or to a matching close curly bracket.

Comments may be embedded anywhere in the file (including the middle of titles) using C style delimiters:

```
/*comment */
```

If the first non-comment, non-white space character on a line is the left curly bracket, all following definitions up to the close curly bracket are for substructures of the previous name. For example:

```
/* This is a sample name dictionary */
1   aname   now is the time /* I hope */ for all
2   another
  {1  abc   good men
   2  def   to come to
  }
3  lastname (sic)
  {99 abc "handle"}
```

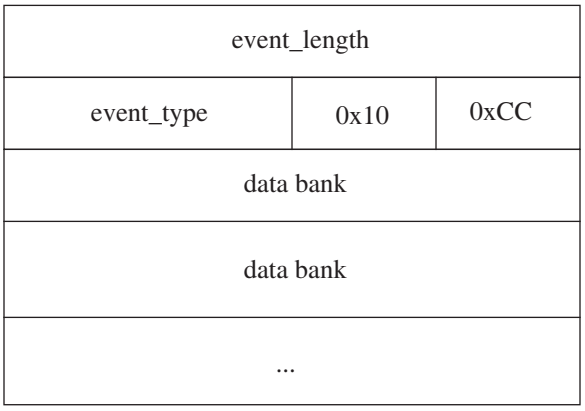
Defines the outermost names *aname*, *another*, and *lastname*, where the fragment *another* has 2 defined sub-fragments named *abc* and *def*, and *lastname* has one defined sub-fragment named *abc*. By convention, sub-fragments may be uniquely referred to by giving all parent names in order separated by periods. In this example, *another.abc* is a fragment with a title of *good men*, and *lastname.abc* has a title of *"handle"*.

APPENDIX F
F CODA Event Bank Definitions

(See Appendix E for a discussion of the bank structure used below.)

F.1 Standard Physics Event

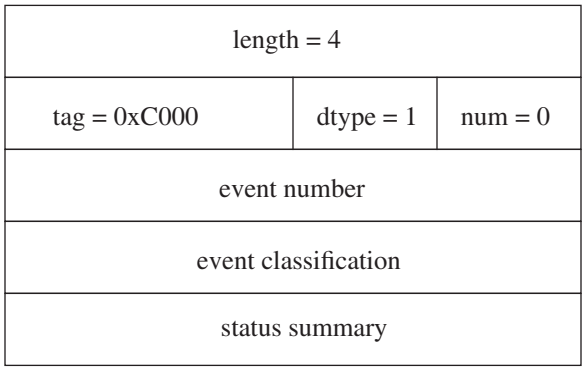
A standard physics event will consist of an outermost bank with a data type of “bank”, a tag equal to the event type, and the “num” field containing the hex constant 0xCC (used to verify that this is an event header word):



The first data bank is created by the event builder (see below), and other banks come from the readout controllers. Event types produced by CODA data acquisition front ends range from 0-15, corresponding to the event readout code.

F.2 Event ID Bank

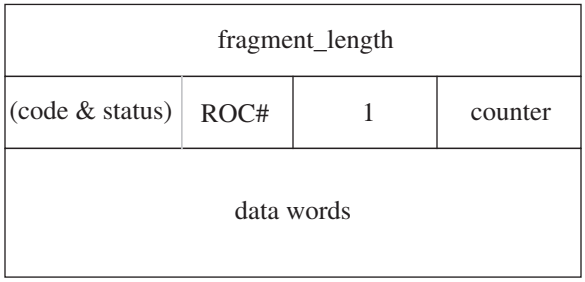
The event builder creates an event ID bank in the following format:



The event number is a counter of the events within a run, starting at 1 with each new run. The event classification word initially holds the 4 bit trigger code; analysis programs would use other bits to indicate the presence of application specific features in the event. The status summary initially holds readout status, one bit per readout controller, and is later replaced with analysis status information.

F.3 Readout Controller Data Banks

Data from each readout controller will be contained in separate banks, with the tag equal to the ROC number (0-31) and the datatype = 1 (longwords). The num field for these banks will contain the low 8 bits of the event counter on the corresponding ROC, and will be used by the event builder to verify that the fragments are properly synchronized.



Prior to event building, the high 11 bits of the tag field will be used to pass the readout code number and status information from the ROC's to the event builder. These bits are cleared by the event builder leaving a tag from 0-31.

F.4 Run Control and Sync Events

For each of the state transitions *prestart, go, pause,* and *end,* and for each synchronization event, each ROC generates an event fragment containing information about the transition. The event builder waits until it receives the events from all participating

ROC's and then forwards only one of the events up the analysis chain. The format for these events is as follows:

event_length		
event_type	1	0xCC
data words		

The first data word in each of these events is a 32 bit integer containing the time in seconds since Jan 1, 1970 GMT (this follows the Unix convention for time formats). Additional data words specific to the event type follow (see below).

The following event types are defined:

Event Type	Definition
16	Sync event
17	PreStart event
18	Go event
19	Pause event
20	End event

F.5 Sync Event

Sync events are generated automatically by the trigger supervisor, or upon operator command. Sync events have the following format:

event_length = 5		
event_type = 16	1	0xCC
time		
number of events since last sync		
number of events in run		
status		

Time is the number of seconds since Jan 1, 1970 GMT. Status is a bit mask with one bit set for each ROC which encountered an error in checking the readout synchronization.

F.6 PreStart Event

PreStart events are generated during the transition from the downloaded state to the pre-started state. PreStart events have the following format:

event_length = 4		
event_type = 17	1	0xCC
time		
run number		
run type		

Time is the number of seconds since Jan 1, 1970 GMT. The run type is a 32 bit number from the rcRunTypes file (Appendix A).

F.7 Go Event

Go events are generated for each Start or Resume command. Go events have the following format:

event_length = 4		
event_type = 18	1	0xCC
time		
(reserved)		
number of events in run thus far		

Time is the number of seconds since Jan 1, 1970 GMT. The number of events in the run will be 0 for the first “go” (i.e. the initial Start command), and will be non-zero for each resume command following a pause.

F.8 Pause Event

Pause events are generated for each transition from the active state to the paused state. Pause events have the following format:

event_length = 4		
event_type = 19	1	0xCC
time		
(reserved)		
number of events in run thus far		

Time is the number of seconds since Jan 1, 1970 GMT.

F.9 End Event

End events are generated each time a run is ended. End events have the following format:

event_length = 4		
event_type = 20	1	0xCC
time		
(reserved)		
number of events in run		

Time is the number of seconds since Jan 1, 1970 GMT.