Chapter 7: Format Definition

In this chapter, we will describe the concrete format of the file label and the table of contents. The motivation for the design choices is given in the previous chapter.

The format is effectively completely new, although it still encodes essentially the same information. Changes to this chapter are not indicated by change bars because almost everything would be marked.

The new format accomodates 64 bit value offsets, 32 bit generation numbers, and reference tables for values. At the same time, the format results in much lower overhead for most TOC entries.

Container Label Format

As mentioned in the previous chapter, a standard Bento container must have a label at the end. Exotic containers may have labels in other locations. However, the label will always provide the same information.

The label contains the smallest possible amount of information, because it has to be the most stable part of the standard. It consists of seven fields:

Magic byte sequence

```
unsigned char magicBytes[8];
```

This identifies the container as a Bento container. It must be chosen to be unlikely to occur as the initial or final bytes in any existing file.

Label flags

```
unsigned short flags;
```

These are currently unused.

Buffer size

```
unsigned short blockSize;
```

This defines the size of the TOC blocks in this container, in multiples of 1024 bytes. This used to be the encoding field, which is no longer needed because the TOC now has a fixed encoding.

Container format major version number

```
unsigned short majorVersion;
```

The major format version number changes only on incompatible format changes.

Container format minor version number

```
unsigned short minorVersion;
```

The minor version number changes on upward compatible format changes. When the major version number changes, the minor version number is set to zero.

Offset of TOC

```
unsigned long tocOffset;
```

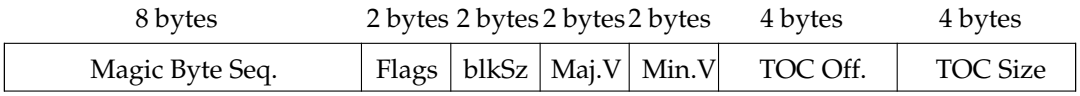

This provides the offset of the top level TOC (Table of Contents) from the beginning of the container. The TOC describes the structure of the rest of the container.

Size of TOC

```
unsigned long tocSize;
```

This provides the size of the TOC in bytes.

A picture of the Bento label format is as follows:



We do not expect additional information to be added to the TOC label. Instead, it can be added as values of the distinguished object #1, which represents the container itself.

Table Of Contents Format

Understanding the TOC is the key to understanding how the Bento format works. We begin with an overview, and follow up with the concrete details.

The TOC consists of a sequence of entries. Each entry corresponds to a single segment of a value of some object.

TOC entries are sorted by object ID, and within a single object they are sorted by property ID. This gives us two things:

- All the entries for a given object are contiguous in the TOC, and all the entries for a given property are contiguous within the object.
- We can find an object within the TOC, or a property within an object, by binary search. If an object ID or a property is not defined, we can quickly determine that it is not defined.

Thus, each object in the container is represented in the TOC by a sequence of entries, one for each segment of a value of the object. (Bento has no way to represent an object without at least one value.)

Since each TOC entry defines a value, we know immediately that it must indicate the object ID, property, type, and data of the value. In addition, it indicates the generation number of the value and it may contain additional bookkeeping information for the value. Let us examine each of these in detail:

Object ID

The ID of the object that this value is part of.

Property

The property is indicated by the object ID of a property description.

Type

The type is indicated by the object ID of a type description.

Data

The value is indicated by the offset and length of the sequence of bytes representing the value. The offset is a 0 origin byte offset from the beginning of the container. The length is a byte count, and may be 0, indicating a 0 length value.

If the data is four bytes long or less, it may be included directly in the TOC as an **immediate value**, rather than being referenced by offset and length.

Generation number

This allows applications to check consistency between different properties.

Bookkeeping information

This field provides some additional information about the entry. It is described in more detail below.

Table Of Contents Low-Level Design

A TOC entry could simply be defined by putting all the information above in a record. Unfortunately, this record would be relatively large and would be very likely to contain redundant and/or unused information.

Specifically, the object ID, property, type, generation number, and bookkeeping information are each 4 bytes long. The data length is also four bytes, and the data offset can be as much as eight bytes. This is a total of 32 bytes, which would impose a significant overhead on small values.

Furthermore, sequential TOC entries are very likely to have the same object ID and generation number. They are may have the same property, and even the same type. (This last case means that the sequential entries are segments of the same value.) Many TOC entries will not need any bookkeeping information. Thus a scheme that requires all these fields for each TOC entry imposes considerable unnecessary overhead.

Instead, we have adopted an approach in which each TOC entry contains only the information that is new or different compared with the previous TOC entry. This results in a TOC that is organized as a stream rather than a table, and is parsed as it is read in. Due to the compression, we have found that reading in the new TOC format is significantly faster than reading in the old tabular format.

We will describe the TOC format at two levels: first, the logical structure of the stream, and second the actual physical representation.

Logical Stream Structure

The following grammar describes the set of stream elements and how they can be combined. Terminals are given in **bold**; they are described in detail in the list following the grammar. "*" means repeated 0 or more times. "$^+$" means repeated 1 or more times. Square brackets mean present 0 or 1 time.

```
TOC             ::= object-ID property+

property        ::= property-ID value+
```



```
value              ::= type-ID [gen-num] [ref-obj-ID] value-data

value-data         ::= initial-data continued-data*

initial-data       ::= immediate | short-reference
                       | long-reference

continued-data     ::= ctd-immediate | ctd-short-ref
                       ctd-long-ref
```

Object-ID, Property-ID, Type-ID, Ref-obj-ID

These are all object IDs. An object ID is a 32-bit persistent indentifier. The namespace for object IDs is local to the container. Object IDs are assigned sequentially as objects are allocated, and the last ID allocated in the container is maintain as a property of object # 1.

Gen-num

This is a 32-bit generation number of the value. It can be set by default from the container, or explicitly for a given value. Generation numbers are intended to be assigned sequentially to each consistent update of the container. The generation number of the TOC value in object # 1 is the default generation number of the container.

Immediate, Ctd-immediate

These are 0, 1, 2, 3 or 4 byte data values. They represent the actual data of the value. Except for the 0 byte case, they are all packaged in a 4 byte field. The ctd-immediate is exactly the same as the immediate, except that it is flagged to indicate that it is a value segment other than the first.

Short-ref, Ctd-short-ref

These are references to the data for a value segment using a 32-bit offset and a 32-bit length. The ctd-short-ref is exactly the same as the short-ref, except that it is flagged to indicate that it is a value segment other than the first.

Long-Ref, Ctd-long-ref

These are references to the data for a value segment using a 64-bit offset and a 32-bit length. The ctd-long-ref is exactly the same as the long-ref, except that it is flagged to indicate that it is a value segment other than the first.

Physical Stream Format

To make the TOC readable without starting from the beginning, it is broken up into fixed size blocks. The block size in any given container is specified in its label, and can range from 2^{10} bytes to 2^{26} bytes.

At the beginning of each block, there is assumed to be no previous information, and all information is written to the TOC. Thus a block can be read in and interpreted in isolation.

The stream of elements in the TOC is annotated with format information to make it possible to parse it. This annotation is done by placing one-byte codes at each point where the stream is ambiguous. Each distinct code is followed by a specific sequence of fields.

The specific byte codes used int the TOC, and associated sequence of fields for each code are given in the following list. The fields correspond to the grammar terminals described above. All fields are four bytes long, except for long data offsets, which are eight bytes long.

`NewObject 1U`

Fields: object ID, property ID, type ID

Meaning: Full specification of the identity of a value in the container. Found only at the beginning of an object or the beginning of a block.

`NewProperty 2U`

Fields: property ID, type ID

Meaning: Information required to begin a new property in an existing object.

`NewType 3U`

Fields: type ID

Meaning: Information required to begin a new value in an existing property.

`ExplicitGen 4U`

Fields: generation number

Meaning: Generation number of this value is different from preceeding value, or this is the first value in a block.

`Offset4Len4 5U`

Fields: offset (4 bytes), length

Meaning: Reference to value data; the first data segment for the value.

`ContdOffset4Len4 6U`

Fields: continued offset (4 bytes), length

Meaning: Reference to value data; **not** the first data segment for the value.

`Offset8Len4 7U`

Fields: offset (8 bytes), length

Meaning: Reference to value data; the first data segment for the value.

`ContdOffset8Len4 8U`

Fields: continued offset (8 bytes), length

Meaning: Reference to value data; **not** the first data segment for the value.

`Immediate0 9U`

Fields: none

`Immediate1 10U`

Fields: immediate data (1 byte) stored in a 4 byte field

`Immediate2 11U`

Fields: immediate data (2 bytes) stored in a 4 byte field

`Immediate3 12U`

Fields: immediate data (3 bytes) stored in a 4 byte field

`Immediate4 13U`

Fields: immediate data (4 bytes) stored in a 4 byte field

`ContdImmediate4 14U`

Fields: continued immediate data (4 bytes) stored in a 4 byte field

Meaning: Immediate data but **not** the first data segment for the value.

`ReferenceListID 15U`

Fields: object ID

Meaning: The ID of a bookkeeping object associated with this value. Occurs before any data references. Omitted if the value does not have a bookkeeping object.

`EndOfBufr 24U`

Fields: none

Meaning: end of current block, go to next

`NOP 0xFFU`

Fields: none

Meaning: NOP or filler; skipped.

Additional Table Of Contents Issues

The description above covers the byte level format of the TOC. However, there are a number of fine points that are necessary to understand how the TOC is used.

Standard Object IDs

Object ID 0 is never used, to avoid confusion with `NULL`, and for error checking.

The first 2^{16} object IDs (out of 2^{32}) are reserved for standard objects specified in the Bento specification. Thus, any ID with the two high order bytes 0 is the ID of a standard object. Most standard objects are type or property description objects, but some are specific objects required by the implementation. A list of the currently defined standard objects is given in Appendix D.

Standard objects are intended for use in defining the format of containers and supporting the API, and are not intended to be used for application data, types, or properties. Standard objects are required to "bootstrap" the API, since even finding the global name in a type description requires knowing a particular property and type.

TOC entries for standard objects are not required, but they are permitted. Thus, the IDs for standard objects may be used without a corresponding TOC entry.

Any given container can contain TOC entries for any subset of the defined properties for a given standard object, from none to all. If the TOC **does** contain entries for defined properties of standard objects, the entries must conform to the specification. Thus, a reader need not look in the TOC when it encounters a standard object ID.

Allowing entries for standard objects in the TOC gives us a mechanism for providing some backward compatibility. If we want to introduce a new standard object, we can put its description in the TOC of backward compatible containers for a while. Then older applications can still understand that object, assuming they can understand the description. At a minimum, the object description can be used to explain to the user what's missing and what she can do about it.

Additional "non-standard" properties can be given for standard objects, as long as they do not conflict with the standard properties.

TOC Self Reference

Every TOC contains a standard object that is used to describe the TOC itself. In particular, it is object ID 1, so the TOC entries for the TOC itself always come at the beginning of the TOC.

Additional TOC properties can be useful. For example, an index to speed access to the entries by ID could be attached to the TOC through another property. Potentially several such indexes, using different formats, could be attached.

Applications are free to attach additional properties to object 1. For example, it is sometimes useful to have an array of "root" objects in a container, from which all other objects can be found. Information about when the last update was performed, by whom, etc. could be maintained using properties of object 1. Etc.

Generation of Object IDs

Object IDs other than IDs of standard objects are generated by sequentially incrementing a counter from `0x00010000`. Object IDs are never reused in later generations of a container if an object is deleted. The last ID number generated is kept as a property of object # 1 to allow generating further IDs without reuse.

Generation Number

The current generation number of a Bento container is the generation number of its TOC, as recorded in the TOC value entry. Any change at all in a container requires a change to the TOC (since at least one entry has to change), so this is logically consistent.

Immediate values

Eight bytes to twelve bytes of an entry are required to "point to" the value data for an entry (four or eight bytes of offset and four bytes of length). When the property value is large, such as a video clip, PICT, or rich text object, this is reasonable, but when it is just an object ID or an integer, it is excessive.

An entry can be an immediate value. In this case it contains data instead of a offset and length referring to the data. All of the other fields of the entry are interpreted exactly as before. In particular, the type of the entry still describes the format of the value.

The library automatically creates immediate values when a four or less bytes of data are written to a new value segment. If more data is written to the value later, the API automatically converts it to a normal TOC entry.

Note that it is perfectly legal for an entry to "point to" a single byte of data, or even zero bytes of data, somewhere in the file. Such an entry will never be created by the library, but will be read correctly if it is encountered. Immediate values are only an optimization and are never required.

Spelling of Globally Unique Names

A given globally unique name is the value of a property of a type, property, or other description. The type of that property entry defines the format of the name. Currently, all Globally Unique Names must be in ISO 9070 name syntax, and must be derived from a name provided by an ISO naming authority.

Updating and Bookkeeping Information

The information used by the library to record updates and to perform reference tracking is stored as normal values in objects. Of course, these values are marked with standard properties known to the library.

The format of this information is not described in this version of the Bento specification. It is fully documented in the library source code.

Format Usage Issues

Dealing With Tagged Streams

The actual value data corresponding to a TOC entry in general will be completely "naked," with no required bytes specified by the format around it at all. This is in contrast to "internally tagged" file formats, in which each chunk of data is required to begin with (at least) a size and type, and possibly other information.

To accomodate existing internally tagged data formats, in Bento a given format of value can be defined to contain its own size, type or any other metadata. Furthermore, a TOC can be created that provides different TOC entries for data and tags in a stream of tagged chunks. In this case the tags are still in the stream, but they have been "logically separated" from the data when viewed through the Bento API. The tags can even be omitted from the TOC altogether, in which case they will be invisible if the data is viewed through the API.

