Chapter 6: Format Overview

This conceptual description of the Bento format is intended as a road map to the detailed specification. As such, it says what but not how. When we get to the detailed level, we will see various caveats and tricks which are passed over in silence at this level. However, the format is very simple, so the details actually concern specific usage conventions more than extensions to the basic structure.

Key Ideas

There are four key ideas in the Bento format:

1) everything in the container is an object,
2) objects have persistent IDs,
3) all the metadata lives in the TOC (Table of Contents),
4) objects consist entirely of values, and
5) each value knows its own property, type, and data location.

Let us discuss these in turn.

Everything is an object

In an Bento container, every accessable byte is part of a value of some object. Even the metadata that defines the structure of the container, and the label of the container, are values of an object. Type descriptions are objects, property descriptions are objects, etc. We will exploit this fact in various ways below.

Objects have persistent IDs

Every Bento object is designated by a persistent ID which is unique within the scope of its container. Objects may have additional IDs and/or names that are unique in larger scopes, but this is not required.

Object IDs provide a compact, convenient way to refer to an object. Clearly, we must provide an efficient mechanism to get from any object ID to information about that object.

All the metadata lives in the TOC

This is a difference between Bento and most other container formats, such as ASN.1, formats derived from IFF, etc. In these other formats, the metadata is associated with the chunks of data that it describes, a design approach that we call **internally tagged**. Jerry Morrison, the designer of IFF, was one of the major influences on the design of Bento; it was partly his experience with IFF that led us to move away from internal tagging.

There are three reasons for this difference from other formats:

a) Bento needs to support very flexible layout, such as multi-media interleaving, and internal tags would be inconvenient and even harmful for this.
b) Applications inspecting an object can make decisions about it more efficiently if all of its metadata is concentrated in one place, rather than being spread out over the container with its values.

c) We want to be able to assimilate existing formats that contain collections of objects without forcing them to change. This implies that we must be able to designate regions within the existing structure as values, without forcing them to somehow retrofit internal tags.

Note that putting all the metadata in the TOC does not increase the amount of storage required. If we have a TOC for random access, it has to contain an object ID and an offset to each value in any case. Moving the type and size of the value to the TOC does not use more storage; it even uses less if the object ID also would have been stored in the object header.

This approach to metadata does impose one significant design constraint. A Bento container can only be read by starting with the TOC. This raises two questions: (1) how do we find the TOC, and (2) how do we access the TOC when we need information?

1) In standard Bento containers the container label points to the TOC.

   Possibly some non-standard containers will exist that require other mechanisms (discussed below). However, these will be exotic cases.

2) Since we need to access the information in the TOC whenever we want to read a value, we have to have it available at all times. This normally means that the container needs to be on a random access device.

   If a container needs to be read on a device that does not support efficient random access (such as a CD-ROM) the TOC can be split up into sub-TOCs that sit in front of the groups of objects they describe, and then the container can be accessed largely in stream order.

<u>Objects consist entirely of values</u>

In Bento, an object has no value as such. Each object has properties, and each property has values. The Bento format provides no information about an object except its ID.

Of course, an object can have a single value; in that case the value of the property "is" the value of the object. Thus we can easily accomodate the "normal" case.

<u>Each value knows its own property, type, and data location</u>

Each value consists of a property ID (or role), a type (or format), and data. For example, a graphic object might have a value that describes its "clip mask"; the property ID would specify what role the value plays, but not what format it is stored in. The type would define how the mask is represented: rectangle, bit mask, path, Mac region, PostScript path, etc. The data would be the representation of the mask itself.

At the level of the container standard itself, there are no restrictions on what values an object can have, how many values it can have, etc. However, individual object formats may dictate rules in this area. In general, applications should be prepared to encounter additional values that they do not understand; these can be ignored. This allows other applications to annotate objects with additional values that may not be generally understood. Typically, these values will be associated with properties that are unknown to the application.

The data of a value is an uninterpreted sequence of bytes which may be from 0 to 2^{64} bytes long. (The current Bento library only supports access to 2^{32} bytes per value, but the format is defined to support up to 2^{64} .) This sequence of bytes has no format requirements or restrictions. Furthermore, the byte sequences representing the data for various values of various objects can be placed anywhere in the container. Thus there are no strong data format requirements for the container as a whole, although it must contain the metadata to define its structure somewhere.

Special Cases

All of the mechanisms above are consistent across all the uses of objects. However, there are two special cases that need to be considered.

Multiple values

The format allows a single object to have multiple values with the same property ID. All the values must have different types. Such multiple values are intended to be used as alternative representations of the same information.

Value segments

The table of contents can contain multiple entries for a single value. These entries mean that the value represented by the entry consists of multiple segments.

This permits values to be broken up into chunks and interleaved, without creating problems for applications that view them as single values. In addition, it allows an application to build TOC entries that "synthesize" a value out of separate parts, as is required in retrofitting some file formats.

Note that these two special cases can be mixed freely. A property can have multiple values, and one or more of the values can be composed of multiple segments.

Other Issues

Globally Unique Names

To fulfill the requirement for locally generated unique names for types and properties, we have chosen to use the identifiers defined in ISO 9070. These are names that begin with a naming authority (assigned to a system vendor or an application vendor), and then continue with a series of more and more specific segments, until they end in a specific type or property name.

Names generated in this way are both unique and self-documenting. Individual users can generate unique names using this approach. For example, a user developing educational stackware might want to create properties, or even types, to use in scripts. The stackware development environment could automatically generate a unique prefix for the user, based on the serial number of the development tool, and then append the user generated property or type name.

This ensures that if that user's scripts and data are combined in a container with other information generated by other users, no naming conflicts can occur.

Note that globally unique names are not limited to property and type descriptions. Any object can be given a unique name using exactly the same mechanism, and such object names may be useful in some applications.

Note also that objects can be given short names that are only locally unique, as in the RIFF TOC. These would be a different type than Globally Unique Names.

Type and Property Descriptions

Recall that type and property descriptions are objects as well. What should we put in such descriptions?

Since types and properties need to have globally unique names, so that applications can recognize them, type and property descriptions will typically have a globally unique name value. In many cases, this may be the only contents of a description.

Sometimes, however, we may wish to put more information into a description. Here are some examples of useful information that can be attached to types or properties:

<u>Base types</u>

Base types allow us to inherit semantics from other existing types and compose it into more complex types. See the discussion of types and dynamic values for more details.

Note that the base type information is intended to include uses such as encryption, compression, I/O redirection, etc.

<u>Encoding information</u>

A type definition may indicate the default encoding of its values.

Typically, all of the values with the same basic format in a container will have the same encoding, so this new subtype can be shared by all these values. In this case the encoding can be indicated directly in the type description for the format.

If values with the same basic format but multiple encodings exist in the same container, a more complex solution is required. In this case, a subtype may be created just to record the encoding. Such a subtype will typically not need a globally unique name.

<u>Compression information</u>

In addition to the compression technique, typically recorded via a base type, the type can record compression parameters, the codebook used (if applicable), etc.

As with encoding information, a type that exists just to record compression information typically will not need a globally unique name. It will refer to the underlying format type and the compression technique, both of which will have globally unique names.

<u>A template or grammar for a type</u>

This allows applications that have never seen this type before to parse values of that type and potentially get some useful information out of them. Examples of description mechanisms that could be used in this way are ASN.1 and SGML.

The more general type will be indicated as the super-type. For example, a given SGML DTD as a type will have a specific SGML definition of the DTD. The super-type of this type would be SGML itself, which defines the basic encoding conventions.

Method descriptions for a type

A type could have properties that provide method definitions. Providing methods in the container would allow fully encapsulated use of values.

Consistency

Given these mechanisms, some objects may have a large and varied set of property values. Furthermore, containers may be copied with modifications by applications that do not understand all the property values of each object. Often we would like such applications to retain the values that they do not understand, but this naturally raises a major question of consistency. For example, if two values are alternative representations of the same information, and one of them is edited, but the editing application does not understand the other format, then the old value is out of date, and the object as a whole is inconsistent.

To give applications a handle for managing this problem, we define generation numbers. When a container is created for the first time, it has a generation number of 1. Each time a container is copied with modifications, its generation number is incremented. Each property entry contains the generation number of the last generation in which its value was modified (presumably by an application that understood its format). This allows an application to compare several values that should be consistent, and determine whether they were all modified together, and if not, which ones were modified most recently.

Example 1: A Bento container might hold a PICT and a GDI value representing the same drawing. These could be stored as values of the same drawing object. If this drawing is edited by an application that only understands GDI, the PICT value would be out of date, and an application that wanted a current PICT would need to invoke translation services to get a new version.

Example 2: A container might have extra indexes for the TOC, but these might not be understood by all applications that use the container, and it might be modified by an ignorant application that doesn't update its index. Comparison of the generation number of the TOC itself and the generation number of the index would immediately indicate whether they were in synch.

In addition, if desired, a container could retain information about the date and time each generation was created, and perhaps the individual responsible. This information could be provided as properties of the TOC itself.

Finding the Table of Contents

Obviously, given the importance of the table of contents to the structure of an Bento container, the first thing an application needs to do when accessing a container is find the table of contents. The standard mechanism for locating the TOC makes this easy.

The obvious mechanism would be to have a label at the beginning of the container (typically a file) that contains the offset of the table of contents. However, this turns out not to be quite the right approach.

In many cases, developers want to convert their existing file formats into Bento containers. Ideally, they would like to keep the resulting files readable by their existing applications that do not understand Bento. For example, we may be wrapping Bento around a RIFF file, and we may want the result to still be readable by an application that only understands RIFF. If the file has to have a standard Bento label at the beginning, it will be unreadable, unless the Bento label is specially designed to be compatible with RIFF. In that case, the label will be incompatible with some other format. Etc.

Our solution to this is to define the standard Bento format to have the label at the **end** of the container. This makes backward compatibility easy. In addition, a copy of the label can optionally be placed at the beginning of the container. This allows easy recognition of a Bento container if it is coming in from a stream-oriented I/O mechanism.

In exotic cases, this approach may not work. For example, transmission streams may depend on framing information, so that the stream can be read starting anywhere. Bento supports such exotic cases by allowing the TOC to be found in non-standard ways when necessary. Finding the TOC is the responsibility of the I/O handler, which will depend on the system and the container type. The I/O handler can adopt a non-standard approach if required.