Chapter 8: Format Usage

This chapter serves two purposes. FIrst, it provides several examples of how the container format can be used. These are complete, worked examples with specific values for all the relevant TOC entries.

Second, the more advanced examples show how to use the built-in extensibility of Bento to address additional needs, by defining the required properties and types. Thus these advanced examples provide examples of how to extend the basic format.

Usage Examples

These usage examples are given in a relatively complete and literal form. Even the TOC entries for standard objects are given, although they are not required, to provide a complete picture of what is going on.

The examples are given in the prevous, tablular form of the TOC, which is much more human readable than the new stream format.

Many details of the representation have been invented for these examples, such as the actual ID values, details of the naming conventions used in the property and type names, etc. The discussion in the Format Overview and Format Definition chapters explains the items which are part of the standard as such.

In particular, the Global Unique Names in the examples have not been revised to conform to ISO structured name syntax.

Embedded Stream Files

Suppose we have a container that simply contains two objects, with references from the first stream to the second. For example, the first object could consist of a stream of rich text, and the second object could be an image that is logically embedded in the text. Let us further suppose that the image has two alternate representations in different formats, each of which is a stream. This example exercises much of the basic machinery of Bento.

Below is a picture of the relevant parts of the TOC, and following that is a detailed discussion of the entries in the TOC.

. . .

a	268	18	22	vo.1	vl.1	0	0

. . .

b	400348	18	22	vo.2	vl.2	0	0
c	400563	18	22	vo.3	vl.3	0	0
d	723421	38	268	vo.4	vl.4	6	0
e	723655	38	400348	vo.5	vl.5	8	0
f	723655	38	400563	vo.6	vl.6	11	0

. . .

vo.1: "Bento:ContentStandards:Text:RTXT"

vo.2: "AppleComputer:Imaging:PICT"

vo.3: "Microsoft:Imaging:WindowsMetafile"

vo.4: <the actual RTXT stream...embedded ID 723655...>

vo.5: <the actual PICT stream>

vo.6: <the actual Windows metafile stream>

Each TOC entry is labeled with a lowercase letter to make it easier to refer to it; the letters are not actually part of the TOC contents. For the sake of this example we assume that the rich text type is a standard object that does not need to be in the TOC, but that an entry is provided anyway. We also assume that the types of the image streams are not standard objects, so that they must be provided.

Entry (a) is the only entry in the rich text type description. The local ID of the rich text type is 268. Note that since this is less than 2^{16} we know that it is a standard object. Thus this TOC entry is not really required, but it is provided for illustative purposes. The value of the entry is simply the Globally Unique Name that identifies the type. The text of the Globally Unique Name is listed below the TOC entries, labeled "vo.1" (for "value offset 1"). The value of vl.1 is the length of the string, including the null at the end (I didn't count). The property ID (18) means "TypeName"; the type ID (22) means "GloballyUniqueName". The description object is immutable, so it has generation 0. No flag bits are set (in this example, we are not using any flags).

Entries (b) and (c) are very similar. Each one is a type description object. Because their IDs are greater than 2^{16}, we know that they are not standard objects, and thus they are actually required in the TOC (this is mainly for illustrative reasons). The property and type IDs are the same as in entry (a).

Entry (d) is the single entry for the RTXT object. Its property ID (38) means "Primary Value". Its type ID (268) is the same as the ID of entry (a), indicating that its value is an RTXT value. Its offset (vo.4) points to the actual rich text data, which contains an embedded ID (723655). Its generation number (6) means that its entry and value were last modified in the sixth generation (copy) of this container.

Entries (e) and (f) are also very similar, but in a different way than (b) and (c). Note that both entries have the same ID, indicating that they are both properties of the same object. Furthermore, both entries have the same property ID (38) indicating that they are both primary values of the object. Thus, they are alternative representations of the primary value. Looking at their type IDs, we can see that one value is a PICT (400348) and the other is a Windows metafile (400563). Furthermore, looking at the generation numbers, we can see that both have been updated more recently than the text stream but that the Windows metafile was updated most recently.

The TOC Itself

A somewhat more recursive example is the description of the TOC by itself. Every TOC actually contains such a self-description, so reader code actually has to deal with some of this structure, but it will typically not be visible at the application level.

This is a portion of the same TOC as the previous example, so we will see some relationships between the content objects described above, and the properties of the TOC itself.

a	1	38	5	vo.1	vl.1	11	0
b	1	2	6	723689	–	11	1
c	1	3	25	723421	–	6	1
d	2	20	22	vo.2	vl.2	0	0
e	3	20	22	vo.3	vl.3	0	0
f	5	18	22	vo.4	vl.4	0	0
g	6	18	22	vo.5	vl.5	0	0

. . .

vo.1: <the actual TOC itself>

vo.2: “Bento:Basic:TOC:IDSeed”

vo.3: “Bento:Basic:TOC:RootContentObject”

vo.4: “Bento:Basic:TOC:AbsoluteTOCFormat”

vo.5: “Bento:Basic:TOC:Integer4Byte”

Entry (a) is an actual reference to the TOC as a value. The offset field (vo.1) will contain the offset of the TOC in the container(that is, the offset of the beginning of this entry). The length field (vl.1) will contain the length of the TOC in bytes. The property indicates that this is the primary value for the TOC object, and the type indicates that it is the normal top-level TOC format.

Entry (b) is also a property of the TOC object. It contains a value 1 larger than the last ID used. Note in this case that it is somewhat higher than highest ID that appeared in the previous example. This occurs if more objects were created, and then deleted. The seed prevents reuse of those IDs, in case some of them have been remembered (either within this container, or in external references to objects in this container). This prevents accidental aliasing.

Since the value of entry (b) is only four bytes long, it can be stored as an immediate value. Note that the immediate flag is set.

Entry (c) indicates the root object of the content in the container. Note that it contains the ID of the RTXT stream in the previous example, since that is the root content object. The ID is also an immediate value.

Note that the generation numbers of the TOC entries correspond to generation numbers of the relevant content objects. The generation of the TOC value itself is the generation of its most recent object. The generation of the root reference, however, indicates when the root was set to that object. It is the same generation as the object itself, so probably that object has been the root since it was created.

The remaining entries, (d) through (g), would not actually appear in a normal TOC because they are standard objects. Note, however, that they would be legal. They are provided for illustrative purposes. The first two are property descriptions, and the second two are type descriptions. Note that the only property of each description is the Globally Unique Name.

Types and Properties

Our last usage example is the most recursive, and would never occur in a real Bento container, but it documents the format, and it may be interesting as an extreme case. The example shows the top of thetype hierarchy, where the properties TypeName and PropertyName, and the type GloballyUniqueName are defined. Naturally these descriptions are all standard objects, and they are very unlikely to appear in any TOC. Furthermore, realistically, if they did, no reader would be able to use them. However, as an example, this may give some insight into both the more exotic uses of Bento, and the structure of the type and property mechanism.

In addition, to tie the example to some of the concrete types we have already seen, we give a more complete derivation of the RTXT type, including its supertype COBJ, and a dictionary that describes RTXT in terms of COBJ.

. . .

a	18	20	22	vo.1	vl.1	0	0
b	20	20	22	vo.2	vl.2	0	0
c	21	20	22	vo.3	vl.3	0	0
d	22	18	22	vo.4	vl.4	0	0
e	22	21	25	26	–	0	3
f	25	18	22	vo.5	vl.5	0	0
g	26	18	22	vo.6	vl.6	0	0
h	76	20	22	vo.7	vl.7	0	0
i	132	18	22	vo.8	vl.8	0	0
j	136	18	22	vo.9	vl.9	0	0
k	268	18	22	vo.10	vl.10	0	0
l	268	76	136	vo.11	vl.11	0	0
m	268	21	25	132	--	0	3

. . .

vo.1: "Bento:Basic:TypeName"
vo.2: "Bento:Basic:PropertyName"
vo.3: "Bento:Basic:SuperType"
vo.4: "Bento:Basic:GloballyUniqueName"
vo.5: "Bento:Basic:LocalIDReference"
vo.6: "Bento:Formats:Printable7BitAscii"
vo.7: "Bento:Descriptions:DataFormat"
vo.8: "Bento:Formats:COBJ:GenericStream"
vo.9: "Bento:Formats:COBJ:TypeDictionary"
vo.10: "Bento:ContentStandards:Text:RTXT"
vo.11: <the actual RTXT type dictionary>

In entries (a), (b), and (d) we finally see the definition of the ubiquitous property IDs 18, 20, and 22. Aside from the fact that they use each other, and (b) and (d) use themselves, they are fairly normal property and type descriptions.

Entry (c) introduces the SuperType property, and it is used in entry (e). Entry (e) effectively says that a GloballyUniqueName is spelled in printable 7 bit ASCII. Note that it uses an immediate reference to the supertype. Entry (f) defines the local ID type we have been using, and entry (g) defines printable 7 bit ASCII.

Entry (h) introduces a very general property that allows us to attach data format descriptions to objects. It is intended primarily for use in type descriptions. The interpretation of the data format description will depend on its type.

Entries (i) and (j) are parts of the description of COBJ, a stream oriented data definition standard under development at Apple. (i) describes the type of actual COBJ values, while (j) describes the type of COBJ type dictionaries, which define the format of particular streams.

Finally, entries (k) and (l) and (m) describe RTXT (ID 268, as we saw in the first example). Here in addition to the Globally Unique Name of RTXT, we have the COBJ type dictionary, and the reference to COBJ as the supertype of RTXT.

Multi-Media Issues

Multi-media formats require two types of support beyond what is defined in the basic format::

1) They need to be able to interleave values and then reconstitute them in two ways:

a) by playing them linearly, without holding a large table of contents in memory or seeking off to look at the table of contents, and

b) by asking for them as "normal" values, and getting them either as a single hunk of data in a buffer or as a non-interleaved stream.

2) They need to have a stream of data containing **local** tables of contents, which will (typically) describe what is coming up in the stream, allowing a"player" to get ready, select the right stuff as it comes along, etc.

Using interleaved values

Requirement (1) is largely handled by value segments, which permit arbitrary interleaving of values. This only addresses the format flexibility and the ability to retrieve the complete value as a "normal" value.

Note that the actual interleaving is determined by the application that writes out the values into the container. Thus, the container does not "understand" interleaving, and has no built in mechanism to define the particular interleaving chosen. This allows total control by the application.

To play interleaved values efficiently, without frequent references to a remote table of contents, the player application and the authoring tools need an additional contract. Each interleaved object needs to "know" the rules by which its values are broken up, how far apart the pieces are, etc. This information can be recorded in an additional property attached to each interleaved object.

Using this additional information, the player application can read the values directly from a stream, without having to remember the TOC entry for each partial value. The player will only need to reference a table of contents to select objects to play, and to retrieve the interleaving specifications.

The definition of particular interleaving is beyond the scope of this specification If and when an industry standard interleaving specification emerges, we can certainly make sure Bento meshes with this standard.

Thus, all that is required to address requirement (1) is an interleaving specification attached to the appropriate objects.

Local tables of contents

Problem (2) can be solved by allowing partial tables of contents in a container. This is required for other reasons, such as breaking a Bento container across multiple volumes. This question is currently under investigation.

Other Usage Issues

Property Index

Sometimes we want to find all the objects in a container with a particular (set of) value(s) of a particular property. With a few objects that have the property in a sea of irrelevant ones that do not have the property, such a search may become very expensive.

To facilitate finding all the objects with a given property, we can keep an index of objects according to which properties they support.

External Reference Table

When a container is moved into a new environment, often its links to other containers, files, etc. need to be fixed up. This fixup can be done by a general utility, or even by the finder (in some future version) if the links are sufficiently inspectable.

The format of the TOC as described provides some of this inspectability. All references to external entities are indirect. Therefore, external references can be found by a scan of the TOC. However this remains somewhat inconvenient. It forces utilities to distinguish between external references and other uses of indirect values by inspecting their types. Furthermore, it does not specify any particular scheme for sharing the potentially common information in external references, such as directory path names.

To simplify the task of fixing up the links of a container, we can provide a list of all the external references in the TOC, as a property of the TOC itself. In its simplest form the external reference list only needs to contain the object IDs of all objects with external references. However, we will probably want some information about the shared content of these indirect values as well, to avoid redundant binding.

Encoding

We need a way to specify encoding information for the value of each entry. A single object may have different property values created on different systems (for example, a PICT and a Windows metafile), and the encoding information needs to be able to reflect this.

The encoding information is logically independent of the type of the value, but in many cases, the encoding may be derivable from the type. In some cases, however, the type and encoding are essentially independent.

To deal with this set of design constraints, we will specify encoding by the type. However, when we have a type that is essentially independent of encoding, we will need to specify the encoding explicitly as a property of the type description. Note that in some cases we could have two or more types with different encoding properties, but the same underlying "abstract" format type.