Chapter 1: Introduction to Bento

Bento is a specification for storage and interchange of compound content. Bento defines two things: a format for containers of compound content, and an API to access these containers. Bento containers are used by application programs to store compound content that consists of multiple content objects. Bento is designed to be platform and content neutral, so that it provides a convenient container for transporting any type of compound content between multiple platforms. The Bento code corresponding to this specification currently runs on Macintosh, MSDOS, Microsoft Windows, OS/2 and several varieties of Unix.

This specification defines the format of data in a Bento container, and an API for writing and reading Bento containers. The Bento design is the result of several years of prototyping work, and extensive discussions between several major vendors.

If you have further questions about Bento or want to give us feedback, please contact Jed Harris via email. His address is jed@apple.com. We welcome your responses to Bento.

How to Read This Specification

This introduction and the following requirements chapter provide motivation for the Bento design. It is important to understand the requirements because Bento is a general container standard that addresses a very broad range of uses. Typically any single user of Bento encounters only a subset of these requirements, and it may be difficult to understand the motivation for some design decisions if you are thinking only of a subset of the requirements.

The API. may be easier to understand than the format, because it has some analogies to APIs of existing services such as file systems. However, it does deal with a number of subtle issues that file systems typically do not handle.

The Bento format is simple, but very flexible and recursive. The usage examples in chapter 8 should be helpful in understanding the concepts in the overview (chapter 6) and the description of the format (chapter 7).

Why We Need Bento

Increasingly, documents are made up of multiple content elements, such as text, tables, images, formatting information, mathematical equations, graphs, etc. Often content is created using one application and then included in documents created by other applications. Later, content elements may be copied out of a document and used in yet other documents. And so on.

Right now, applications typically have no way to exchange multiple content elements, unless they have a "private contract" about the format they will use. Furthermore, one application typically has no way to find the content elements in another application's document, so typically it cannot let a user copy the these content elements and reuse them. Finally, every application developer who wants to store multiple content elements in a document typically has to invent her own object storage mechanism.

These are the problems that Bento addresses. Bento provides a mechanism for storing content elements as objects. It defines a standard format for storing multiple different types of objects, and an API for writing out objects and reading them back in. Bento is designed to be as simple, flexible, and efficient as possible, so that it can be used to solve a wide variety of different content storage problems without requiring changes or extensions to the core specification. In turn, this will allow the widest number of applications to cooperate using Bento as a means for storing and exchanging their content elements.

More About Objects

For the purposes of this specification, we will consider an object as simply one or more hunks of data that "hang together" and that can be referenced by other data. Bento objects can be simple or complex, small (a few bytes) or large (up to 2^{64} bytes). Compared with objects in languages such as C++, Bento objects are typically larger and more complex, because they represent user meaningful content elements, rather than the atoms and molecules used to build this content.

For example, a sequence of bytes of data would not be an object, because we can only understand the bytes if we know how they will be used. A paragraph, an image, etc. can be an object if it contains enough information so that we know how to interpret it. Typically an object contains information about what kind of object it is, and some data, which provides the content of the object. In this specification, the information "about" the object is called metadata, and the content of the object is called its value.

What is a Container?

An object container is just some form of data storage or transmission (such as a file, a piece of RAM, or an inter-application message) that is used to hold one or more objects (both their metadata and their values). Bento containers are defined by a set of rules for storing multiple objects in a such a container, so that software that understands the rules can find the objects, figure out what kind of objects they are, and use them correctly. This is basically a simple idea, and Bento is a simple format. However, it was tricky to design, because it has to accomodate an enormous variety of different kinds of objects, different ways that applications want to use objects, and system considerations about how data can be stored.

Bento is intended to provide a container definition that can conveniently, efficiently, and reliably hold all the different kinds of objects that users and applications want to group together, store, and exchange. Bento does not define how any given object is structured internally, because there are already a very large number of different object formats around today, and we are still inventing new ones. Objects stored in an Bento container can have proprietary or standard formats, they can be designed to use the Bento mechanisms or they can be completely ignorant of the the existance of Bento.

How You Can Use Bento

To better understand why Bento is useful, let us look at some scenarios of how it can be used in specific cases:

Object Interchange

Let's say we are building a presentation. We want to take data in a spreadsheet, construct a 3D chart from it, and decorate it using a 3D modeling application. Then, we want to render it, animate it, and give it a sound track, titles, etc. As we go along, we want to keep track of the connection to the original information, so that if the data in the spreadsheet changes, we won't have to throw away all our work.

To do this, we will need to store the information created by our spreadsheet, charting application, modeling application, and animation application. In the process, we will be accumulating more and more different types of content elements, with lots of relationships between them. If the applications all agree to store their content elements as Bento objects, they can keep track of all the objects and their connections, even if (say) the animation application doesn't understand spreadsheet objects.

Document Storage

Continuing with this example, suppose we have all the objects set up in our animation, and we would like to save them in a file for future use. Furthermore, we would like to be able to find some of the objects and unpack them even if the animation application isn't around, so that (for example) we can reuse the spreadsheet data, or some of the 3D decorations we have cooked up. We can do this if the animation application stores the objects in a file formatted as a Bento container; then other applications can look inside the Bento file and find the objects they understand.

Enhanced Editing

Now suppose we come back and want to edit some of the objects in our animation. Applications can use two main strategies to edit a Bento document. First, they can read the document into memory, edit it there, and then write it out as a new Bento file. Some applications want to keep an entire document in memory, so this is a natural approach for them.

However, suppose our animation editor needs to handle documents too big to fit in available memory. In this case it can update the original Bento file, or create a new "delta" file. In either case, incremental changes are saved without modifying the existing data in the original file. This provides data integrity, so that the file won't be corrupted if the system crashes during an update. Only the minimal information required to record the changes is saved; for example, if four bytes in a 256K value are overwritten, only those four bytes plus some bookkeeping information are saved. However, in the current implementation of the Bento library, deleted or overwritten bytes are not recycled. Eventually, the application may wish to copy the file to compress out the unused space.

Compression and Other Data Transformations

Of course, the movie we are generating could be very large. However, Bento allows compression and decompression to be done on values. This mechanism is extensible, so a clever new compression scheme could be used that takes advantage of the characteristics of animation. Furthermore, the mechanism is transparent to the application, so the compression mechanism might be provided by a third party, and even added after the initial animation application has been written.

This same mechanism can be used for other types of data transformations as well. For example, the spreadsheet might contain sensitive financial information, so it might be encrypted, with only the users who need to be able to access it having a key.

References Across Containers

Maybe some of the backgrounds in our animation are scanned pictures that are available on a server in our department. These 24 bit scanned images are several megabytes each, and we don't want them on our local hard disk! So, we don't copy them into the animation; instead we put links to them in the file, and Bento knows that these are references, not the real objects, and transparently follows them to find the images in the server library. If the images are stored in Bento containers too, the references aren't limited just to a file, but can be to specific objects within a file.

If we use the Bento format for interchange between applications, when we use one of these images in one application, and then copy the result and paste it into a new application, all that has to be communicated is the reference, not the multi-megabyte image itself.

Finally, when we take our animation home to work on it in the evening, of course the references to the giant images don't work anymore, since we can't connect to the server. However, we can also save alternate representations in a Bento container. If we have saved low-resolution, 8 bit copies of the images, then, using Bento, the application can transparently use the version we do have when it can't find a complete image.

Multi-Media CD-ROM

Imagine that we want to build a CD-ROM using formatted text, video clips, sound, animation, "live" simulations, you name it. We want our different media related, so that when the user clicks on an animated object, it can play a sound or pop up a text description. Furthermore, the information on the CD-ROM has to be stored in a carefully crafted layout, so that when a low-cost machine is playing the CD-ROM, it can read and use different types of data (such as sound and video) "just in time", without a lot of memory to hold the data until it's time to use it, and without introducing annoying pauses while it searches the CD-ROM. This requires breaking large objects up into smaller chunks and interleaving them on the CD-ROM, while keeping track of which chunks belong to which objects. Bento provides the flexibility required to do this interleaving, while leaving the decisions about the specific formats up to tools that understand the details of the media requirements.

In addition to just playing the CD-ROM, we want to have authoring tools that let users take the video, sound, animation, text, etc. objects on the CD-ROM and re-organize them, so that (for example) a teacher can turn reference material into customized courseware. The tools should be able to look at the objects without having to understand the complicated details of how they are laid out on the CD-ROM, and it should be able to create new objects in its own files that reference the old objects on the CD-ROM, just as though they were ordinary objects in a file. Bento allows these tools to view the interleaved objects as though they were stored normally in contiguous byte sequences.

Limitations of Bento

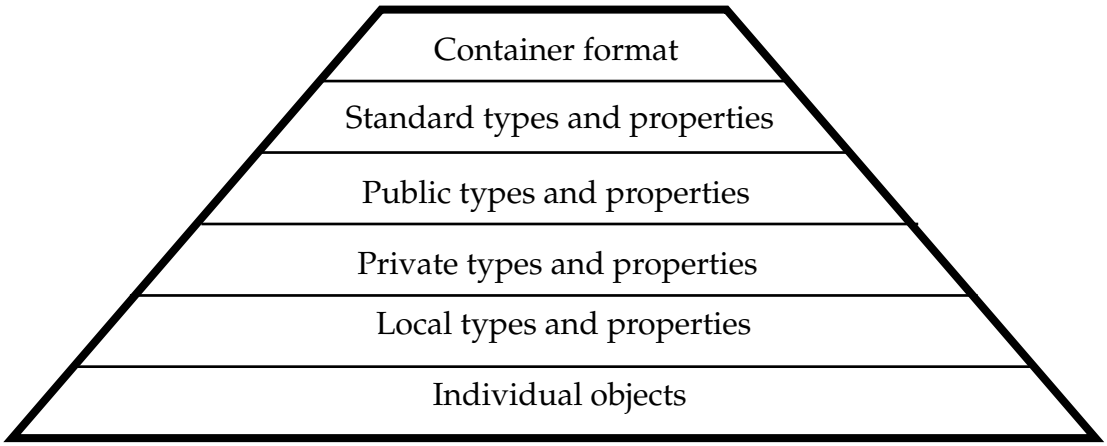
Bento is **not** intended to be used as a general purpose concurrent access database mechanism. Full support for such database style mechanisms would require much more complex libraries, and would conflict with the type of format control required for multimedia support (among other things). Thus it is not appropriate to try to use an Bento container as a general purpose concurrent access persistent object database.

Incremental update is possible using Bento. For example, an application managing a very large document might want to change the text in a paragraph, but avoid copying the entire document to make the change. Instead, it could update the contents of the paragraph, but leave the rest of the document unchanged. The effect of this update would be to make a new copy of the changed parts of the paragraph at the end of the document.

Eventually, the document would need to be copied to get rid of the "dead space" left behind by changes. Bento does not currently reuse such dead space. However, we have done analysis of the additional support required to reuse dead space, and it can be done without modifying the Bento storage format. This functionality is a candidate for Bento Version 2.0.

Scope of This Specification

This document (the current Bento specification) completely defines the format of a Bento container at the byte level. However, their are "higher level" issues that this document does not address. Perhaps we can best understand this approach if we look at any given container as being divided into a number of levels:



This specification only describes the top two levels of this diagram: the container format and the standard objects required by the Bento library.

There are several areas in which continuing work will be required if Bento is to achieve its potential:

First, in order to avoid unnecessary incompatibility, it is very desireable to have a strong basic set of public types. For example, we should have strings in a range of character sets, dates, integers, floating point numbers, etc. Data interchange will be greatly enhanced if we agree up front on these basic values. We are now in the process of listing and prioritizing the initial set of basic public types. We would appreciate feedback on the types you feel should be selected for initial standardization.

Second, addtional "layered" standards are needed in various areas. Such layered standards are being defined for multimedia data interchange (OMFI) and other types of content. Additional layered standards of interest include external reference mechanisms, such as system and file structure independent linking mechanisms.

Finally, to the extent that Bento is successful, many content standards will be defined within it. These standards will use Bento facilities, so they will be tied to Bento, and there will be a continuing need to register their types and properties, resolve potential conflicts between them, etc.

To support all of these areas, we plan to put in place a process for administering the Bento standard, and for registering and standardizing Bento types and properties.

Status of Bento

With the 1.0d5 release, we are approaching a final version of Bento 1.0. All previously planned changes have been completed. Products are currently shipping on the 1.0d5 version of the libraries, and we expect the format supported by the final version of the 1.0 libraries to be compatible with the 1.0d5 version.

Changes in Version 1.0d5

Only substantive changes made since version 1.0a4 have been marked with change bars.

The major changes are as follows:

New Table Of Contents Format

We adopted a new TOC format that will support 64 bit offsets, and that is also more compact and flexible than the old TOC.

A number of users of Bento expressed concern that value offsets in the original TOC were only 32 bits, because they need to support storage in which containers can be more than 2^{32} bytes long. We could clearly see that this would be a very serious issue for many users in a few years. Therefore, we concluded that Bento needed to e able to support 64 bit offsets.

As a result, we have developed a new TOC format that allows optional 64 bit offsets. The new format does not impose any overhead on those who do not use 64 bit offsets. This new TOC format is incompatible with the previous format. However, the 1.0d5 library detects and reads the previous format.

In addition to providing 64 bit offsets, the new format provides significantly reduced overhead. Previously, the minimum overhead for a value was 24 bytes in all cases; with the new format, it can be as low as five bytes in important cases.

Furthermore, the new format provides greater flexibility and considerably reduces the likelihood of further incompatible format changes in the future.

Reference Tracking

Copying or deleting structures of objects that refer to each other requires following references from one object to others. Furthermore, in general the references in a copied object must be adjusted.

In the previous API there was no standard way to find all references from a given value to other objects. This meant that we could not write standard utilities to perform deep copies, deep deletes, or to garbage collect a Bento container.

Furthermore, applications could not copy groups of objects without understanding the format of every value, because of the need to fix up references, and possible ID collisions when the copy was done to a different container.

We have made a pure extension to the API that solves these problems by allowing an application or utility to enumerate and fix up the references from any value, without understanding its format.

Error Reporting

Based on feedback from users, we changed the interface to the error handler to pass error numbers, rather than strings. This allows the error handler to more easily classify errors and determine whether they are recoverable or not.

This change only affects the API of the error handler.

Changes Planned

At this point only one minor change is definitely planned for Bento before it is final.

Add Force Alignment Call

We have been asked by the IMA, which is reviewing Bento for standardization as a the bottom layer of a multimedia interchange format, to provide some way to force alignment of values. While the Bento format currently permits this, the API provides no way to request it. This change only involves adding a single routine to the API and is completely upward compatible. It involves no changes to the format at all.

Changes Being Considered

At this point we have no other changes to Bento that are definitely planned. We are considering three possible changes. These would be the final changes to Bento (aside from any bug fixes) in Version 1.0.

We would very much like to hear from developers who either want these changes or would be significantly inconvenienced by them.

Merge Update List with Table of Contents

We are considering a change to the container format that would only affect updated containers. It would be an incompatible change to updated containers, although code would be provided to read containers in the 1.0d5 format.

This change would reduce the overhead of storing updates, and would increase the likelyhood that we could add storage reuse in Bento 2.0 without an incompatible format change.

Optional Explicit Error Return

A few developers have complained that they must run on some platforms that do not support nonlocal transfers of control (setjump / longjump), and that this makes the Bento error handling model difficult for them to use. In addition, some developers might prefer to have the API return error codes rather than call a handler that has to do a nonlocal control transfer.

We have designed a uniform change to the API that would address this problem. Developers who did not need it could ignore it, as we could provide a backward compatible cover for the modified API.

Error Severity Classification

Developers have asked us to provide a consistent way of determining the severity of an error—in particular, whether it is sensible to attempt to recover and continue, or whether it should be regarded as fatal. We are considering modifying the error handler API to include a severity indicator in addition to the error code itself.

Changes Rejected

In the 1.0d4 spec, we said that we were in the process of designing an accessor API for Bento. We decided not to do this because accessors could be implemented as a layer above the current API with little or no performance penalty and only a subset of Bento users wanted accessors anyway.

