# Gbin File Format Specification

| | |
|---|---|
| prepared by: | Hutton, A. |
| reference: | GAIA-C1-TN-ESAC-AH-004-1 |
| issue: | 1 |
| revision: | 0 |
| date: | 2011-12-16 |
| status: | Issued |

## Abstract

Following feedback on the large amounts of memory needed to read gbin files, some changes to the gbin file format will be made to prevent this. The opportunity is being taken to document the format in this technical note.

## Document History

| Issue | Revision | Date | Author | Comment |
|---|---|---|---|---|
| 1 | 0 | 2011-12-16 | AH | Document issued. |
| D | 2 | 2011-12-13 | AH | Include suggestion from G. Holland for unique byte string to separate compressed blocks. |
| D | 1 | 2011-07-22 | AH | Updated draft, following useful feedback from J. Hernandez and H. Siddiqui |
| D | 0 | 2011-07-19 | AH | First draft |

## 1 Introduction

The gbin file format is the format used by DPAC to exchange Main Database records. It is also used as a general purpose storage format for Java objects defined in the MDB data model. Following feedback regarding the memory consumption that took place when reading large gbin files, the format is going to be modified to address the concerns raised. This technical note will document the latest version of the gbin format [1]

An earlier technical note [4] reviewed different strategies and serialiation techniques for reducing memory, and this document follows on from the conclusions given there.

The main motivation for the current change to the gbin format is memory performance, and the aim is not to drastically change the format. However, we intend to take advantage of the change to include some small practical improvements to make working with the gbin files easier (ideas from [3, 1, 5, 2]:-

- Add some identification bytes at the start of the gbin file, and a version number, to make it easy for code to recognise the files.
- Define a header section, which includes meta-data about the file.

In addition, to make transfers of gbin files between DPCs easier by giving data consumers certain assurances about the data they can expect, we have defined a ‘strict’ gbin format, which is a set of constraints on the basic format. These are described in the following section.

Our aim in this document is to provide a clear description of the format of the gbin files. A reference implementation of code to read and write this format is available in GaiaTools.

A brief note on backward compatability: Two versions of the gbin file format have been used up to the present time. The code in GaiaTools which can read older gbin files will not be removed, and so if the GaiaTools code detects a file in an older gbin format, it will use the appropriate reader for that format.

## 2 Uses Of Gbin Files

Gbin files are used to transport java objects defined in the MDB data model between DPCs, and also as a convenient way of persisting these objects. The objects are written by Java applications, and read by Java applications.

[1] A very brief overview of the old format is given in section 9 for comparison.

These two types of use represent two distinct use cases for gbin files, or in other words, two groups of users who have different perspectives on using the files:

1. Transporting MDB records between DPCs. This is a more formal use of gbin files. The format will be shared between distinct groups of users, where the producers and consumers of the data are not the same. In this context, life is made easier for the consumers by providing more constraints on the contents of the files so they know what to expect to receive.
2. As a storage format used internally by DPCs, for example in individual projects. It is also convenient in test cases to hold test input data. Here the producer and consumer of the data is the same - they can make more assumptions about the data they store in a gbin file and expect to be able to use the file in a flexible manner.

To provide the constraints needed in the first scenario, while allowing the format to be also used in the second, we will define a 'strict' gbin format. This is basically a set of constraints that must be followed when producing gbin files for transfer between DPCs. When used internally in a DPC, these constraints can be ignored.

In brief, the strict format specifies which MDB data model must be used to serialize objects, and that all the objects in the gbin file must be the same type. See section 6 for definition of the strict format.

## 3 General Aims

Gbin files have been used in DPAC for some time, and the most general aim in this format change besides resolving the specific memory issue discussed above, is that we can keep using them for the same purposes as before.

However, we can pick out a few broad aims:

- Reading the gbin files should not require a large amount of memory. See section 5 for more details.
- It must be possible to read the gbin file using serial access only (in Java, we could say it has been readable using just a `java.io.InputStream` as input). This faciliates access to data when direct access to the file may not be possible (e.g. Hadoop).
- As regards CPU requirements, it was noted in [4] that the CPU overhead involved in reading gbin files derives principally from decompressing the data. So the general aim as regards CPU usage is that deserialization should not add much beyond that already needed for decompression.

# 4 Gbin Format

This section describes the overall format. The format makes use of the Java serialization mechanism, and so the Java language effectively forms part of the format specification. In the following text, unless bytes are explicitly specified, data is written using the Java serialization mechanism.

The general structure of a gbin file is a header, followed by one or more data sections. A gbin file can be appended to multiple times, but only with data sections. The header appears just once at the top. See Figure 1 for a graphical representation.

## 4.1 Gbin File Header

The following information appears just once, at the start of the file.

*Identification Bytes:* appears only at the start of the file (inspired by [5]):

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 0x89 | 0x47 | 0x42 | 0x49 | 0x4e | 0x0d | 0x0a | 0x1a | 0x0a |
| \211 | G | B | I | N | \r | \n | \032 | \n |

*Format version number:* a Java `int` value[2]. The format described in this document is version 4 [3].

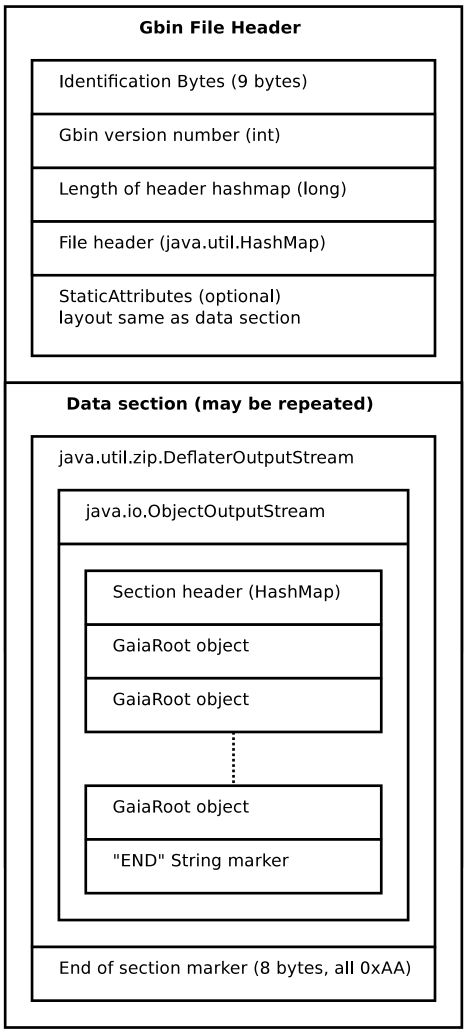
Gbin File Header
Identification Bytes (9 bytes)
Gbin version number (int)
Length of header hashmap (long)
File header (java.util.HashMap)
StaticAttributes (optional)
layout same as data section
Data section (may be repeated)
java.util.zip.DeflaterOutputStream
java.io.ObjectOutputStream
Section header (HashMap)
GaiaRoot object
GaiaRoot object
GaiaRoot object
"END" String marker
End of section marker (8 bytes, all 0xAA)

FIGURE 1: Conceptual diagram of how a gbin is written.

*File header:* the header is basically a map of key value pairs. It is preceded by a Java `long` value[3] giving the length of header. The header itself is a serialized `java.util.HashMap` of `String` vs `Object` entries. The object values should only use types in the Java language itself. Note: the header is uncompressed, unlike the later data sections. See table below for required header values. Additional header values can be added if desired.

[2]Written using `java.io.DataOutputStream`
[3]Version 3 is an earlier draft of the format, identical except for the absence of the end of section marker

*StaticAttributes*: a section holding any static attributes applying to the data in the gbin file. This has the same format as the data section described below, except that in place of `GaiaRoot` objects, there are `GaiaRootTableAttributes` objects, and the section header 'Type' entry is 'StaticAttributes'. Only one such section can exist at the start of the gbin file. If a gbin file is found having more than one such section, or a section which is not the first section, it should be discarded, or, if being stricter, an exception thrown.

| Key | Type | Description |
|---|---|---|
| MdbVersion | `java.lang.String` | The version of the MDB data model used when generating the given data, e.g. '10.1.0'. |
| DpcVersion | `java.lang.String` | The version of the DPC data model used when generating the given data. This should be given only if DPC data is stored in the gbin file, and should *not* be present if IsStrict is true. |
| IsStrict | `java.lang.Boolean` | If this gbin file follows the strict specification. |
| ObjectType | `java.lang.String` | The fully qualified class name of the `GaiaRoot` objects serialized in the gbin file. If IsStrict is true, then all `GaiaRoot` objects *must* be of this type. If IsStrict is false then this entry is informative only. |
| ResetThreshold | `java.lang.Long` | The threshold number of bytes at which reset was called when serializing the data for the `ObjectOutputStream`. This is for information only, and is not required to deserialize the data. It allows for validation code to check a gbin file for compatability with the memory resources available in a processing system. |
| CreationTime | `java.lang.Long` | Date/Time the gbin file was first created. Stored as number of ms between current time and midnight, January 1, 1970 UTC. |

## 4.2 Data Sections

After the header described above, one or more data sections can be appended to an existing gbin file. Each data section is compressed using the DEFLATE algorithm[4]. The data sections have the following structure:

*Section header:* A serialized `java.util.HashMap` of `String` vs `Object` entries. Only two entries are permitted here, both required. 'Type' must have a String value of either 'Data' or 'StaticAttributes'. 'Count' must equal the number of gbin objects stored in this section.

[4]Written in Java using a `java.util.zip.DeflaterOutputStream`. The compression level is left to the user to decide. Most of the time the default compression will probably be used, though some users may be interested in setting it to `NO_COMPRESSION` to avoid CPU overhead associated with compression/decompression when storing data locally.

*Section data:* One or more serialized `GaiaRoot` objects. The objects are written one after the other. After the final object there a serialized String value “END” is written. Please see section 5 for details regarding the serialization.

*End of section marker:* After the compressed data, a series of 8 bytes, all with the value `0xAA` are written. This marker exists to allow for easy seeking with the file for data blocks. The marker bytes have been chosen to be repetitive so to be unlikely to appear within the compressed data itself[5].

It is the responsability of the producer, when appending data to an existing gbin file, to ensure the appended data is compliant with the existing format.

## 5 Serialization Protocol

As described in the previous section, the `GaiaRoot` objects are stored as serialized objects, compressed using the `DEFLATE` algorithm.

The document [4] discussed the issues surrounding using the Java serialization mechanism to serialize these objects. The conclusion there was that by continuing to use the default `java.io.ObjectOutputStream` and by resetting the stream periodically, the build up of a large handle table by the serialization classes (and consequent use of memory) could be avoided.

In [4] tests were made by resetting the `ObjectOutputStream` every $n$ objects. Since object size can vary, a more general approach would be to reset the stream whenever the number of bytes written (ignoring any compression) exceeds a certain threshold.

The number of serialized bytes will be approximately related to the actual memory used by Java to store the objects in memory. A choice of the specific threshold value will be slightly arbitrary. The number should not be too big, i.e. in gigabytes, as memory use for large gbin files would be unacceptable, nor should it be too small, in kilobytes, as it would be inefficient.

In some scenarios, the user may wish to choose a specific value, for example a very large value, because they are concerned about avoiding duplicates in the object graph. In other cases, for DPC transfers for example, it is important the consumer DPC does not have to set aside large amounts of memory to ingest data.

Therefore, a practical value for the threshold will be chosen, which will be obligatory for the strict gbin format, and which will be the default in general for gbin files. In the rare case that a developer wants to change the threshold, they can do so by explicity using a non-default value.

[5]No guarantee is given that it will *not* appear in the compressed data.

We define the default value here to be $50$Mb. This is a figure based on practical experience and which has been found to be workable.

The Java serialization mechanism is not without its critics, and other serialization protocols may be able to improve the performance slightly, or to reduce redundancy in the data (some redunancy is a consequence of resetting the stream). However, countering these points of view are the following:

- The format is intended to be as simple as possible. Although some performance improvements would be possible by using a custom serialized format, our preference is to avoid this, keeping the code simpler, ensuring the code can be maintained by a broad pool of developers who may not be familiar with serialization techniques.
- Techniques which avoid redundancy (caused by resetting the `ObjectOutputStream` ) may not be so necessary when the data is compressed in a `DeflaterOutputStream` , which will minimise this effect.
- The existing gbin format has been used for some time now, and with it the Java serialization mechanism. Users may have certain expectations regarding Java serialization (e.g. use of transient fields, `serialVersionUID` etc). A change in serialization protocol may have various unexpected side effects that need a lot of existing code to be reviewed.
- The Java serialization format brings with it the cross-platform portability of Java. It avoids having to deal with endianism, floating point value and character encoding etc.

## 6 Strict Gbin Files

The strict gbin format will impose the following additional constraints on the file format:

- The file header field MdbVersion must be present, and must refer to an official MDB release.
- Only MDB data model objects may be in the file - no DPC objects may be present.
- No DpcVersion file header field must be present.
- The file header field IsStrict must be present, and must be true.
- The file header field ResetThreshold value used for resetting the ObjectOuputStream must not be increased above the default value of $50 * 1024 * 1024$ (50Mb), to ensure a consuming user does not need excessive amounts of memory to read the file.

- The file header field ObjectType must be present, and specifies the data model class file of the objects in the file.
- All objects in the gbin file must be of type ObjectType.
- All objects in the gbin file must be serialized using the implementation classes in the MDB jar i.e. no custom implementations.
- A gbin file must contain at least one object.

## 7 Acronyms used in this document

The following table has been generated from the on-line Gaia acronym list:

| Acronym | Description |
|---|---|
| CPU | Central Processing Unit |
| DPAC | Data Processing and Analysis Consortium |
| DPC | Data Processing Centre |
| MDB | Main DataBase |
| UTC | Coordinated Universal Time |

## 8 Reference Documents

[1] A brief look at file format design. `http://decoy.iki.fi/texts/filefd/filefd`.

[2] Designing your own format. `http://www.fileformat.info/mirror/egff/ch08_07.htm`.

[3] File format design. `http://www.magicdb.org/filedesign.html`.

[4] A. Hutton. Note on gbin read performance. `http://gaia.esac.esa.int/dpacsvn/DPAC/CU1/docs/TechNotes/GbinTechNote-AH-002`, March 2011.

[5] A. McFadden. Designing file formats. `http://www.fadden.com/techmisc/file-formats.htm`.

## 9 Appendix: Former Gbin Format

For comparison, the previous version of the gbin format, version 2, is presented in Figure 2.

The main issues regarding the previous version are:

1. There is a complex nesting structure of objects - the embedded byte array and ArrayList make reading the data progressively, and thereby using less memory, very difficult.
2. The inner ObjectOutputStream is written without resetting the handle table. This means that for a large file, the handle table can grow very large. A large amount of memory must then be used to read the file, which may not be available on the consumer system.
3. There are no identification bytes or file header section with file meta data. Some information was embedded in the Zip entry file name, but not in an extensible way.

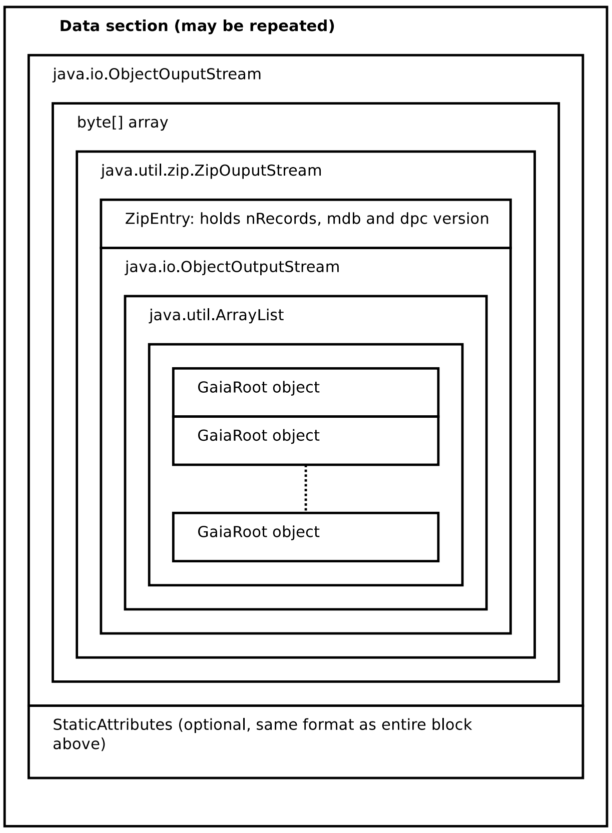
Data section (may be repeated)
java.io.ObjectOuputStream
byte[] array
java.util.zip.ZipOuputStream
ZipEntry: holds nRecords, mdb and dpc version
java.io.ObjectOutputStream
java.util.ArrayList
GaiaRoot object
GaiaRoot object
GaiaRoot object
StaticAttributes (optional, same format as entire block above)

FIGURE 2: Graphical representation of how a v2 gbin file is written.