*Bionic Buffalo Tech Note #66*

# File Identification Headers

*last revised Monday 2003.07.20*

## Introduction

Bionic Buffalo has a standard header structure for non-text data files. The purpose is to allow quick determination of the application and user owning the file, and also of the underlying file organization. This Tech Note describes the standard header structure, and how it is used.

The information herein corresponds to the file header structure version A.0.8.2.

## Application, Organization, and Owner

The identification header is concerned primarily with three aspects of the file: *application*, *organization* and *owner*.

> The application is the program which uses the file. Examples of applications are accounting programs, software engineering tools, and process control systems.
>
> The organization is the underlying file structure. Examples include flat files, indexed sequential files, various kinds of databases, and archives containing other files.
>
> The owner is an indication of who controls or is responsible for the file. Examples include users, processes, or customers.

It sometimes is convenient to be able to learn information about all three aspects of the file before beginning processing. For example, an application may want to ascertain that a file in fact belongs to that application, and not simply that it has a certain kind of name. The underlying file access or database routines may want to know that the organization of the file they are processing is appropriate for them. In both cases, an important function is to determine that the version of the software being used to read or write the file is current or compatible with the version of the file itself. A program might want to operate only on files belonging to a certain owner, ignoring any other files.

The identification header separates the application, organization, and owner information for several reasons:

> In some situations, the software may be concerned with one but not the other. For instance, an application may be concerned about locating its own files for backup or uninstallation (regardless of the organization), while a database utility might be concerned about backing up or reorganizing a file without regard to the interpretation of the data records themselves.
>
> Not all applications, nor all organizations, can be known in advance. When a file of unknown organization is encountered (for example), the application owning it should still be able to recognize the file as its own.
>
> An application may allow stoage of the same data using several different organizations. For example, more than one type of indexed file might be possible, depending on the size of the database or the performance requirements. The application should be able to select the appropriate access routine by knowing the organization.

It is also important to be able to recognize an application, organization, or owner without knowing anything about it. In some cases, a program might want to invoke the access routine or application dynamically, without using complex logic to determine which dynamic routine to use. In other cases, a human may find it convenient to know about an unfamiliar file, again without having to understand the details of the file's organization or of the application which created it.

Instead of relying on a central repository of known or registered applications, organizations, and owners, the standard header identifies entities in two ways: UUID and text string. UUIDs, or “universally unique identifiers”, are 16-octet sequences based on pseudo-random numbers, and are highly unlikely to collide. A UUID may be created by using the common `uuidgen` utility, and will serve to identify an application, organization, or user without resorting to a central repository. Since an unknown UUID is practically useless to humans wanting to know something about a file, the header also contains text strings for the file's application, organization, and owner.

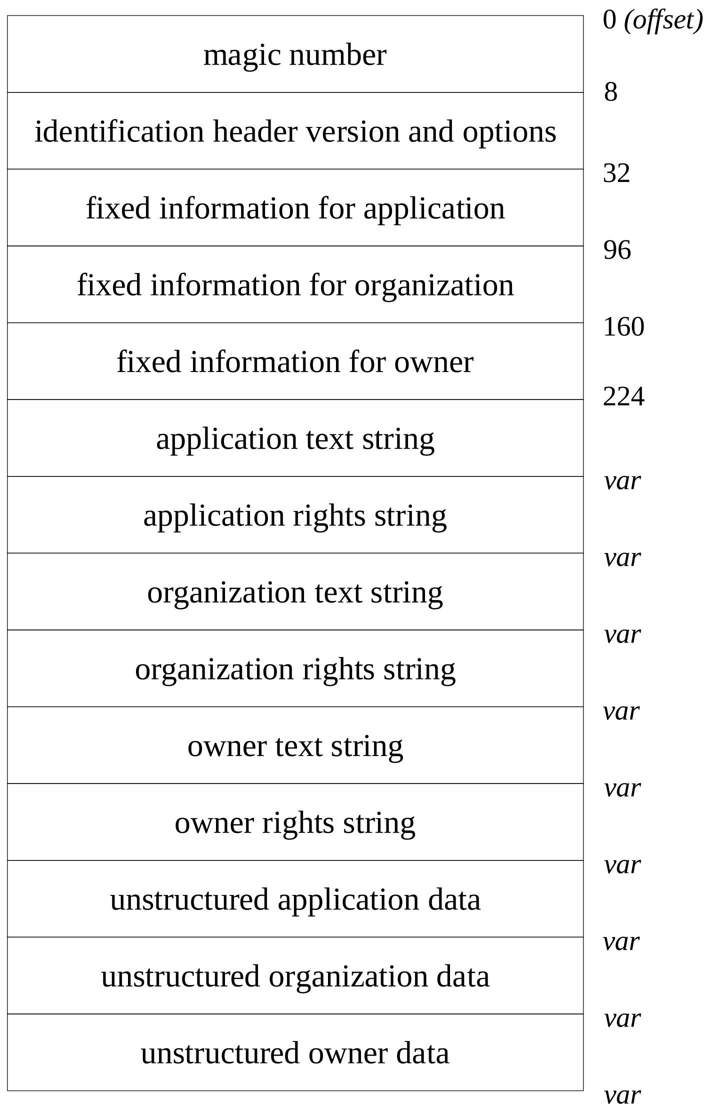
The use of UUIDs and text strings allows the header format to be used by anyone, not only by Bionic Buffalo. Except for UUID and other parameter values, there should be no difference among headers defined by Bionic Buffalo and those defined by other developers.

## Standard Header Structure

Each of the three aspects of a file is represented by the values of a single data structure, the identification information. There are three parts to this structure: a fixed part with specific fields, a text string part (with two text strings), and an unstructured sequence of octets. The text strings are Unicode, encoded using UTF-8.

The standard identification header has the following components, beginning with the first octet of the file:

In diagrammatic form, the identification header layout is this:


0 *(offset)*
magic number
8
identification header version and options
32
fixed information for application
96
fixed information for organization
160
fixed information for owner
224
application text string
*var*
application rights string
*var*
organization text string
*var*
organization rights string
*var*
owner text string
*var*
owner rights string
*var*
unstructured application data
*var*
unstructured organization data
*var*
unstructured owner data
*var*

The components are:

> a magic number, the octet sequence (hexadecimal) `a4 bc dd a0 a5 e6 44 58` (this is a random number, with no special significance)
>
> some fields defining the version and options of the identification header
>
> the fixed parts of the identification information for the application, organization, and owner
>
> the text strings (including rights strings) from the identification information for the application, organization, and owner

the unstructured parts of the identification information for the application, organization, and owner

The identification magic number, header version, and header options contain the following fields:

| ***offset*** | ***length*** | ***type*** | ***description*** |
|---|---|---|---|
| 0 | 8 | octet sequence | magic number or sequence; must be `a4 bc dd a0 a5 e6 44 58` |
| 8 | 4 | unsigned integer | header format version (currently A.0.8.2, represented as `0x01000802`) |
| 12 | 4 | unsigned integer | required software version to read this header (currently the same as above) |
| 16 | 4 | unsigned integer | total length of identification header (including padding for alignment; see below) |
| 20 | 4 | unsigned integer | header CRC or checksum correction value |
| 24 | 1 | unsigned integer | header CRC or checksum correction type:<br>`0 -` none<br>`1 -` simple checksum (total = `0xffffffff`)<br>`2 -` CRC-32 (total = `0`) |
| 25 | 7 | octet sequence | option flags and values, currently unused and must be set to zeroes |

The total length of the above information is 32 octets. Integers in the header (including in the structures below) are all stored in network (big-endian) byte order.

The fixed information for each aspect (application, organization, and owner) contains the following fields:

| ***offset*** | ***length*** | ***type*** | ***description*** |
|---|---|---|---|
| *x* + 0 | 16 | octet sequence | UUID (created by `uuidgen` or by an equivalent function) |
| *x* + 16 | 16 | octet sequence | serial number (assigned as required when each file must be uniquely or somewhat uniquely identified) |
| *x* + 32 | 4 | unsigned integer | type of file (used to distinguish among several types, when necessary) |
| *x* + 36 | 4 | unsigned integer | number of file (used when several similar files must be counted or labelled) |
| *x* + 40 | 4 | unsigned integer | version number of software used to create this file |

| ***offset*** | ***length*** | ***type*** | ***description*** |
|---|---|---|---|
| *x* + 44 | 4 | unsigned integer | earliest version of software able to read this file |
| *x* + 48 | 2 | unsigned integer | length of unstructured information |
| *x* + 50 | 2 | unsigned integer | length of text string (including terminating NULL) |
| *x* + 52 | 4 | unsigned integer | required alignment of file contents |
| *x* + 56 | 2 | unsigned integer | length of rights string (including terminating NULL) |
| *x* + 58 | 6 | octet sequence | unused, reserved for future use, must be set to zeroes |

The total length of the fixed information, for each aspect, is 64 octets. (The value *x* in the offset column is 32 for application information, 96 for organization information, and 160 for owner information.)

By including the required earliest version of software able to read the file, when software reads a newer format file, it can determine if it can safely process it. In case the file cannot be read, it is possible to produce an error message indicating which version of the software must be used. In either case, the version information may also be used to remind the user that upgrading the software is possible and may be desirable.

If a program requires that the contents proper of the file be aligned on a certain boundary (for example, on a 4-byte boundary or 512-byte boundary for efficient memory or disk access), then it may set the alignment field to the required value to indicated this preference. Padding may be added after the identification header to cause this alignment to occur.

Although the length of the unstructured information is represented using a 16-bit number, the maximum value supported is 4096. If the value is zero, then there is no unstructured information.

The six strings begin at file offset 224. The Unicode character set is used, encoded UTF-8, and the string is NULL terminated. If any string is empty, that string's NULL terminator still must be present. Although the length of the string is given in the fixed information by a 16-bit number, the maximum value supported is 256 octets, not including the NULL terminator. When multibyte encoding is necessary, this may represent fewer than 256 characters. Only printable characters are permitted in the string; control characters are not allowed.

The first of the two text strings for each aspect should be a short description of that aspect. For example, “ticket posting utility v.5”, “congo A.0.3.1 record-numbered dataset”, or “anonymous visitor no. 138”. The second of the two text strings, the rights string, may contain copyright notices, security classifications, and similar items, if they are present. The text strings are intended for human understanding, and should give the reader a quick notion of what the application, organization, and owner are, as well as notice of any rights asserted on the contents of the file.

The unstructured information is proprietary to the application, organization, or owner. However, it is public in the sense that it is intended to be read (but not written) by other software. Examples of use include:

An accounting program might provide data for use by add-on or supporting applications, such as custom depreciation or report programs. Although only the accounting program itself would be expected to read the rest of the file, the application unstructured information would be available to the supporting software to make choices about what work to perform, or what to request of the accounting application. For instance, the accounting program might put accounting period dates in the unstructured area, so a report program would know whether the rest of the file should be examined.

A database or file access program might put activity and usage statistics in the organization unstructured area, so other applications might know when to select the file for compression, backup, or other activity.

A printing or rendering application might put address or mailbox information in the owner unstructured area, so a centralized printing facility or business can route the finished work back to the user who commissioned it.

Any application might put copyright or trade secret notices in the owner unstructured area.

Bionic Buffale envisions a tagged structure for the unstructured areas, with certain "standard" tags. However, that specification is not yet complete.

After the unstructured areas, there may be padding inserted before the beginning of the file contents proper. The length of the padding will depend on the combined effect of the three alignment values in the fixed areas.

## Using the Identification Headers

The `morocco` library (see *http://www.tatanka.com/prod/info/morocco.html*), available free under GPL, provides several routines for reading and writing these identification headers. They assume a stream (`FILE *`) is already open for the file. They employ a data structure of type `mar_dataset_ident_info_T` to represent the identification header information.

`mar_dataset_stream_identity_get()` returns the identity information for an open stream.

`mar_dataset_stream_identity_set()` writes an identification header to an open stream; if passed a `NULL` stream, it simply calculates the required length of the identification header

`mar_dataset_stream_identity_release()` frees the storage (including secondary memory) associated with a `mar_dataset_ident_info_T` structure.

The routines use separate `mar_dataset_ident_info_T` structures for application, organization, and owner.

A stand-alone program, `dsetid`, is also included with the morocco distribution. `dsetid` will display information from the identification headers of files specified on its command line.

A program may not assume that the header will not change from time to time as long as the application does not have the file opened. Each time the file is opened, a program should examine the header, and determine (among other things) the offset of the beginning of the file contents proper. All i/o access must be offset by a variable amount, depending on the current length of the identification header. This assumption of mutability permits utilities to modify files with limited knowledge of their contents. For instance,

> an ownership transfer utility might change the owner information without regard to the format of the data itself, and without concern for the application or organization information
>
> a database utility might perform backup, compaction, or other activity, possibly altering the content and size of the header organization information

Except for software which knows the organization of the file's contents (such as a keyed access or database routine, and some applications which manage their own file structures), the file contents beyond the header should remain unchanged. If the header is expanded, however, the contents may be moved away from the beginning to accomodate the expansion. Similarly, when a header shrinks, the contents must be moved toward the beginning to take up the gap. Movement of the contents after the header should account for the alignment requirements specified in the fixed portion of the identification header.

---

---