Network Working Group T. Keiser
Internet-Draft Sine Nomine
Intended status: Informational April 23, 2012
Expires: October 25, 2012

AFS-3 Directory Object Type Definition
draft-keiser-afs3-directory-object-00

Abstract

Directory lookups in the AFS-3 distributed file system are supported by defining a canonical encoding for a directory object, and transmitting all--or part--of that object from a file server to its clients so that clients may resolve paths into AFS file IDs (FIDs). This memo describes the AFS-3 directory object wire encoding.

Internet Draft Comments

Comments regarding this draft are solicited. Please include the AFS-3 protocol standardization mailing list (afs3-standardization@openafs.org) as a recipient of any comments.

Status of this Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on October 25, 2012.

Table of Contents

1. Introduction

AFS-3 [AFS1] [AFS2] is a distributed file system that has its origins in the VICE project [CMU-ITC-84-020] [VICE1] at the Carnegie Mellon University Information Technology Center [CMU-ITC-83-025], a joint venture between CMU and IBM. VICE later became AFS when CMU moved development to a new commercial venture called Transarc Corporation, which later became IBM Pittsburgh Labs. AFS-3 is a suite of un-standardized network protocols based on a remote procedure call (RPC) suite known as Rx [AFS3-RX]. While de jure standards for AFS-3 fail to exist, the various AFS-3 implementations have agreed upon certain de facto standards, largely helped by the existence of an open source fork called OpenAFS that has served the role of reference implementation. In addition to using OpenAFS as a reference, IBM wrote and donated developer documentation that contains somewhat outdated specifications for the Rx protocol and all AFS-3 remote procedure calls, as well as a detailed description of the AFS-3 system architecture.

Unlike most classical network file systems, AFS-3 explicitly eschews remote procedure calls to facilitate file server-assisted directory lookup operations. This was a conscious decision meant to limit server load by placing lookup operations on the clients. In the common cases, where there is significant locality of reference to directory entries, this results in a substantial reduction in server load (especially given the AFS-3 cache coherence model). It should be noted that 23 years of empirical evidence have borne out this decision as useful for many general-purpose workloads, while disadvantageous for certain very specific workloads (e.g., large directory objects with extremely non-uniform directory entry reference distributions--where the server overhead of a lookup rpc would would inconsequential compared to the directory file transfer overhead of the existing model).

Due to the distributed nature of AFS-3 directory objects, a canonical directory wire-format is an intrinsic part of the AFS-3 protocol. This memo documents the directory object wire format; a future document will document the lookup and modification algorithms, which by the decentralized nature of AFS-3 directories, must be implemented in the to-be-specified manner.

1.1. Abbreviations

AFS - Historically, AFS stood for the Andrew File System; AFS no longer stands for anything

RPC - Remote Procedure Call

Rx - The Remote Procedure Call mechanism utilized by AFS-3

XDR - eXternal Data Representation

2. Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

3. Constants

AFS_PAGESIZE = 2048

the size of each page in an AFS-3 directory object (in octets)

MAXPAGES = 128

the maximum number of pages in a legacy directory object

BIGMAXPAGES = 1023

the maximum number of pages in a new (circa 1988) directory object

NHASHENT = 128

number of hash buckets in the entry name hash table

RECSIZE = 32

number of octets in a record

LRECSIZE = 5

base-2 logarithm of RECSIZE

EPP = 64

number of records per page

LEPP = 6

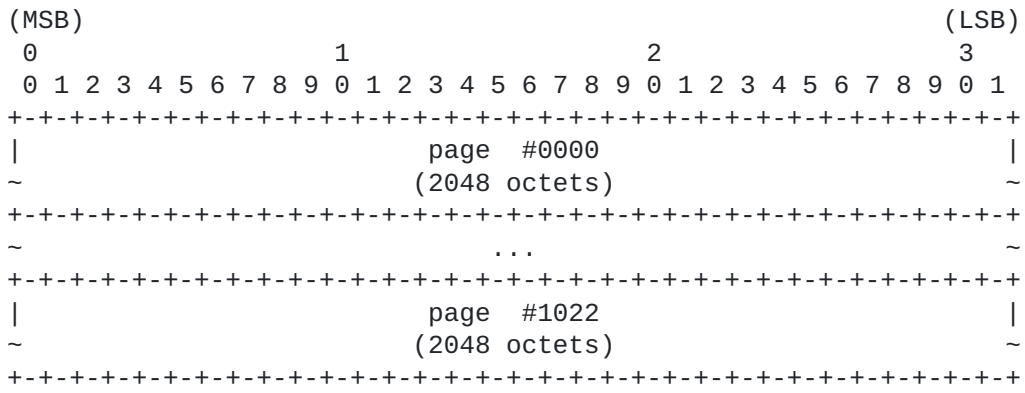
base-2 logarithm of EPP

DHE = 12

number of records taken up in page 0 by the directory header

4. Directory Object Structure

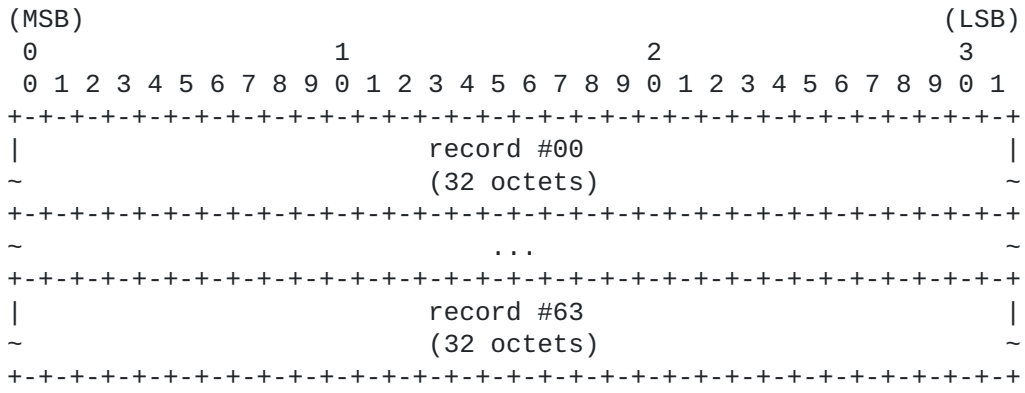
An AFS-3 directory object consists of between 1 and BIGMAXPAGES (1023) "pages" of length AFS_PAGESIZE (2048 octet).



Directory Structure

5. Page Structure

All pages in a directory object are AFS_PAGESIZE (2048 octets) in length. All pages are subdivided into EPP (64) records, each RECSIZE (32 octets) in length.



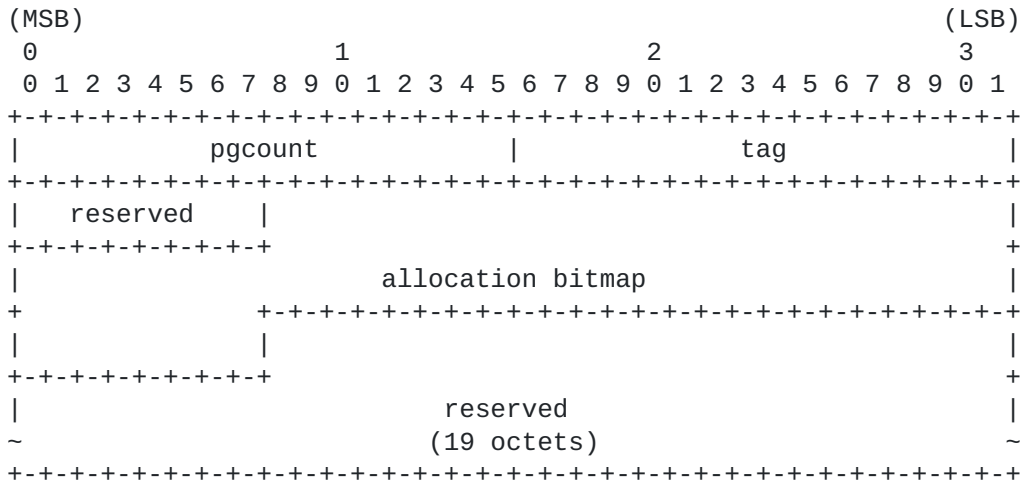
Page Structure

5.1. Record Index

Each record within a directory object is referenced by an index number. This number represents the offset from the start of the file, in units of records, i.e., in multiples of RECSIZE. A record index of 0 would thus point to record 0 in page 0, and a record index of 67 would point to record 3 in page 1. Computing the file offset from an index is simply a matter of left logical shifting the index value by LRECSIZE (5) bits. Conversely, computing the index from a file offset merely involves a right logical shift by LRECSIZE (5) bits.

5.2. Page header

Each 2048-octet page within an AFS-3 directory object contains a 32-octet (RECSIZE) header at offset 0 in the following form:



Directory Page Header

pgcount: 16 bits (unsigned integer)

This field only holds meaning for page 0. For the modern (post-1988) directory format, this field holds the count of valid directory pages (in network byte order) within the directory object. When this value is zero, it denotes a legacy directory object. Legacy directory objects are considered a deprecated and historical vestige, and thus their format is not described in this memo.

tag: 16 bits (unsigned integer)

This field MUST contain the magic value 1234 in network byte order.

allocation bitmap: 8 octets (bit string)

The allocation bitmap is a bit field which contains one bit per record slot within the page. The bit field is thus 8 (EPP/8) octets in length. A bit value of zero indicates the entry is free; a value of 1 indicates the entry has been allocated. The allocation bitmap will be discussed further in Section 5.2.1.

5.2.1. Allocation Bitmap

The allocation bitmap contains one bit per record. The least significant bit of the first octet within the bitmap references the page header object (which is stored at offset 0). The second least significant bit of the first octet within the bitmap references the record index 1 (offset 32 octets into the page), and so on and so

forth...until the most significant bit of the eighth octet of the allocation bitmap references the 64th--and final--record of the page (record 63 at page offset 2016).

The following invariants hold:

```
page_entry_offset = page_entry_index << LRECSIZE

alloc_bitmap_index = page_entry_index >> 3

alloc_bitmap_bit_num = page_entry_index & 0x7

alloc_bitmap_bit = 1 << alloc_bitmap_bit_num
```

The variable page_entry_index, which is an unsigned integer between 0 and 63, can be derived from the record index (Section 5.1) by bit-wise ANDing it with EPP-1.

The equations above are written in C pseudocode--the variables are all assumed to be unsigned integers, and the operators are assumed to be identical to the ANSI C standard.

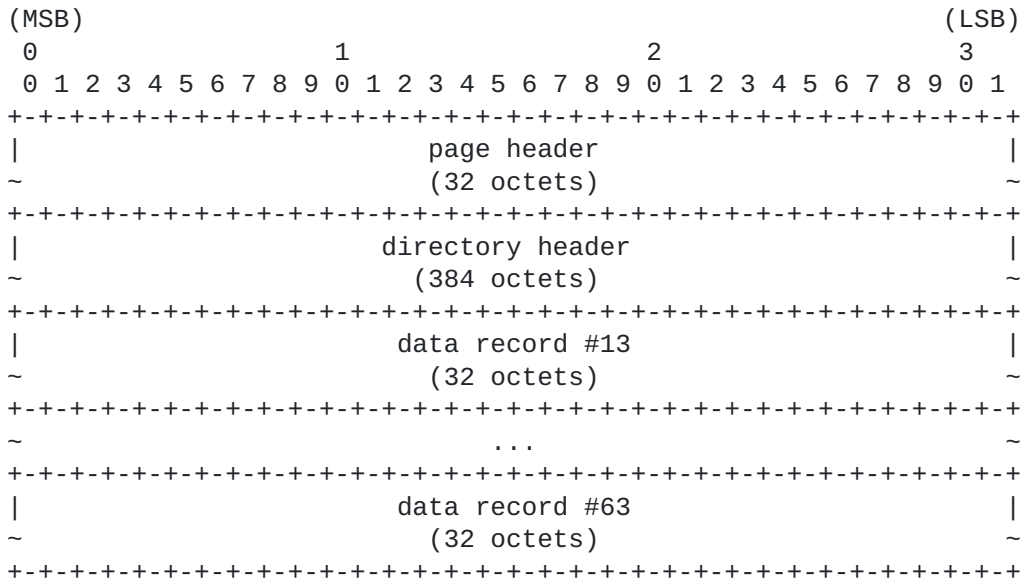
6. Page N=0 structure

Page 0 is special due to the directory header. It is structured as follows:

1. a page header (see Section 5.2)

2. a directory header (see Section 6.1)

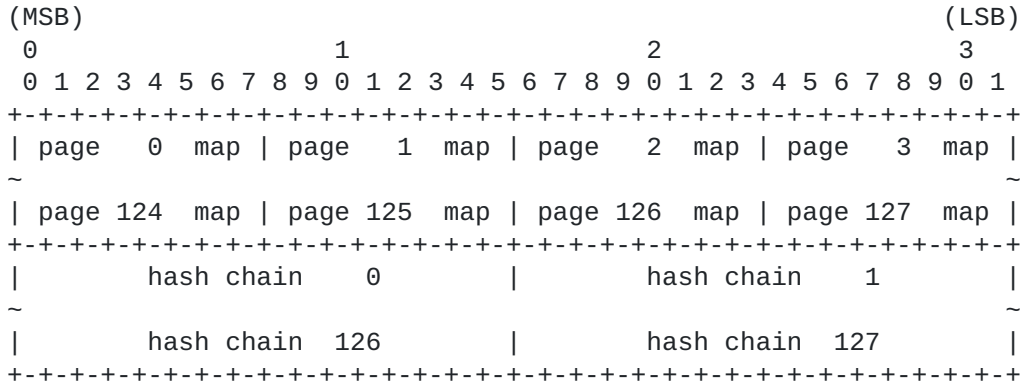
3. 51 data records (EPP-DHE-1=51)



Directory Page N=0 Structure

6.1. Directory Header

The directory header structure is a 384-octet structure that is stored at an offset of 32 octets from the beginning of the directory object (i.e., directly following the page 0 page header--see Section 5.2). The directory header layout is as follows:



Directory Header

page map:

Each of the MAXPAGES page map fields corresponds to one of the MAXPAGES pages in a legacy directory object. Each field contains the count of available entries in this directory page. A value

of EPP indicates that this page has not yet been allocated (i.e., the page header has not been initialized).

hash chain:

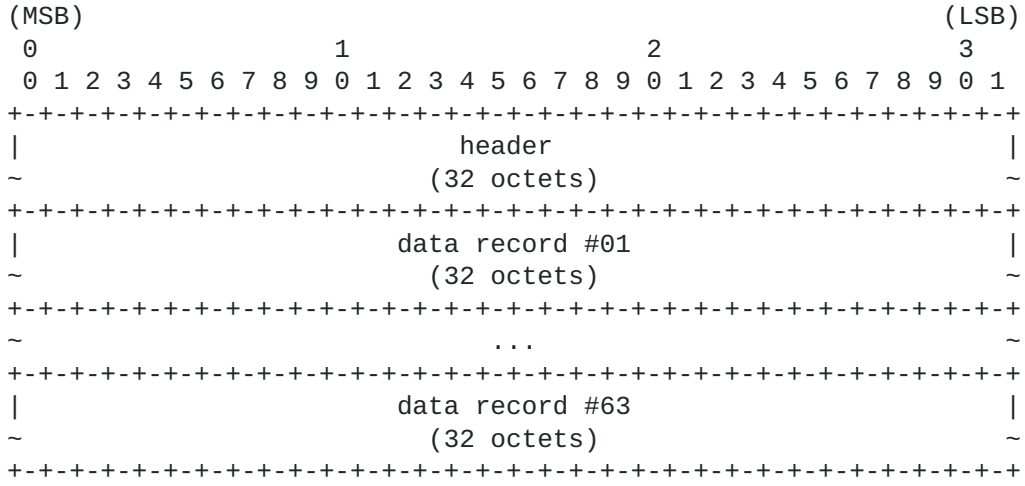
Each of the NHASHENT hash chain fields contains a record index for the first directory entry name that hashes to this bucket (see Section 5.1, and see Section 9 for a description of the hash function)

7. Page N>0 structure

Each page N>0 is structured as follows:

1. a page header (see Section 5.2)

2. EPP-1 (63) data records



Directory Page N>0 Structure

8. Directory Entry

Since entry names are of variable length, directory entries are structured as follows:

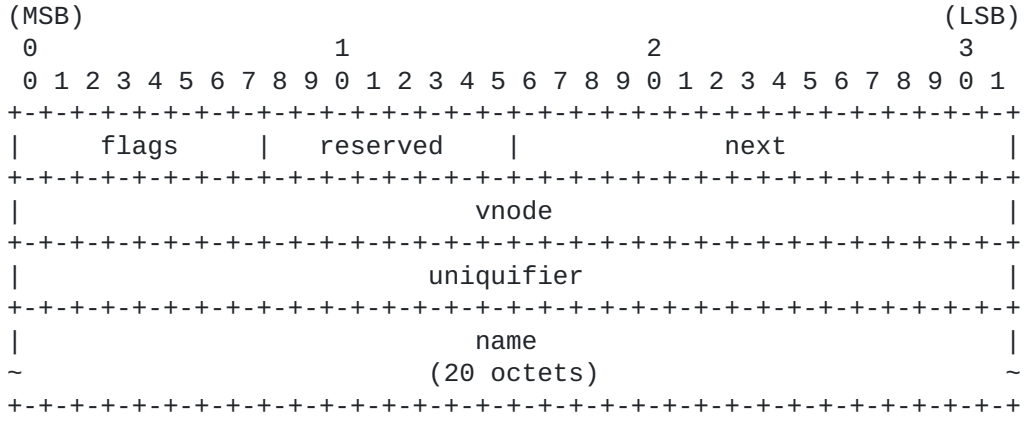
- a directory entry record (see Section 8.1)

- zero or more directory entry extension record(s) (see Section 8.2)

This sequence of records MUST be contiguous, and MUST NOT cross a directory page boundary.

8.1. Directory Entry Record

A directory entry record contains the dirent entry metadata (i.e., the vnode number and uniquifier, the name hash table next pointer, flag bits, and the first twenty octets of the entry name string. Its layout is as follows:



Directory Entry Record

flags: 8 bits

This is an 8-bit wide bit vector. Currently, only one bit is defined: FFIRST (0x1). This flag bit is asserted in every record of this type. There are vestiges in the OpenAFS implementation suggesting that the original intention was to have a second flag bit FNEXT (0x2)--being asserted in the first octet of extension records--which would allow parsers to distinguish between entry records and extension records. However, this idea was never implemented (in fact, the concept of a flags field in an extension record does not exist), and thus this flag bit has no meaning. It should be noted that some tools have attempted to use this bit to distinguish between entry records and entry extension records (under the assumption that 0x1 is not a valid ASCII value for a file name string). This technique is obviously error-prone, and thus SHOULD NOT be relied upon.

next: 16 bits (unsigned integer)

This field contains a record index (see Section 5.1) pointing to the next directory entry record on this name hash chain. A value of zero indicates this is the last record on this chain. This

entry MUST point to a record index which contains a data record of the directory entry type (see Section 8).

vnode: 32 bits (unsigned integer)

This field contains the target vnode number for this directory entry. Together with the uniquifier field, this forms the equivalent of an inode number in the directory entry of a typical on-disk Unix file system.

uniquifier: 32 bits (unsigned integer)

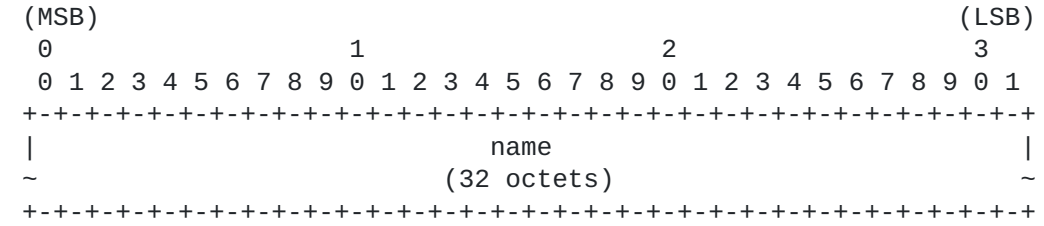
This field contains the target uniquifier for this directory entry. Together with the vnode field, this forms the equivalent of an inode number in the directory entry of a typical on-disk Unix file system.

name: 20 octets (null-terminated string)

The first 20 octets of the directory entry's name string are contained in this field. If the name string (including null terminator) exceeds 20 octets, then this string will continue in a sequence of one or more directory entry extension records (see Section 8.2) that MUST directly proceed this record.

8.2. Directory Entry Extension Record

When a file name string exceeds the 20 octets set aside in an entry record, one or more extension records MUST be allocated contiguously following the base entry record in order to contain the rest of the name string. The layout of an extension record is as follows:



Directory Entry Extension Record

name: 32 octets (string)

This field contains part of an extended directory entry name string. If this is the last directory entry extension record, then this field MUST contain a null terminator character (octet value zero) within its 32 octets.

9. Name Hash Function

The hash function is a loop over each of the octets within the name string. The hash is computed using integer arithmetic on an unsigned 32-bit integer. The hash MUST be initialized to zero before commencing iteration over the characters in the name string. For each character, the hash value is multiplied by the constant 173, and then the value of the current character is added to the hash. When the null terminator is encountered, the loop is terminated before the hash is multiplied by 173.

For reasons unknown to the author, the resultant unsigned hash value is then compared against the value 2^31. If the hash value is less than 2^31 (i.e., what would be the sign bit--if the hash value were signed-- is not asserted), then the resultant hash value will be the value computed in the above loop bitwise ANDed with the constant NHASHENT-1 (127). However, if the hash value is greater than or equal to 2^31, then the resultant hash value will first be bitwise anded with the constant NHASHENT-1 (127), and then the value will be subtracted from the constant NHASHENT (128) to yield the final hash value.

10. IANA Considerations

This memo includes no request to IANA.

11. AFS Assign Numbers Registrar Considerations

This memo includes no request to the AFS Assigned Numbers Registrar.

12. Security Considerations

Directory metadata can contain sensistive information. This memo merely specifies the wire format encoding. Any implementation which may be utilized to store and retrieve directories containing entries whose name strings might reveal sensitive information should take precautions to ensure that they are never transmitted in the clear, and should take steps to ensure that those entries are not cached on machines lacking appropriate physical and network security.

13. References

13.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

13.2. Informative References

[AFS1] Howard, J., "An Overview of the Andrew File System"", Proc. 1988 USENIX Winter Tech. Conf. pp. 23-26, February 1988.

[AFS2] Howard, J., Kazar, M., Menees, S., Nichols, D., Satyanarayanan, M., Sidebotham, R., and M. West, "Scale and Performance in a Distributed File System", ACM Trans. Comp. Sys. Vol. 6, No. 1, pp. 51-81, February 1988.

[AFS3-RX] Zayas, E., "AFS-3 Programmer's Reference: Specification for the Rx Remote Procedure Call Facility", Transarc Corp. Tech. Rep. FS-00-D164, August 1991.

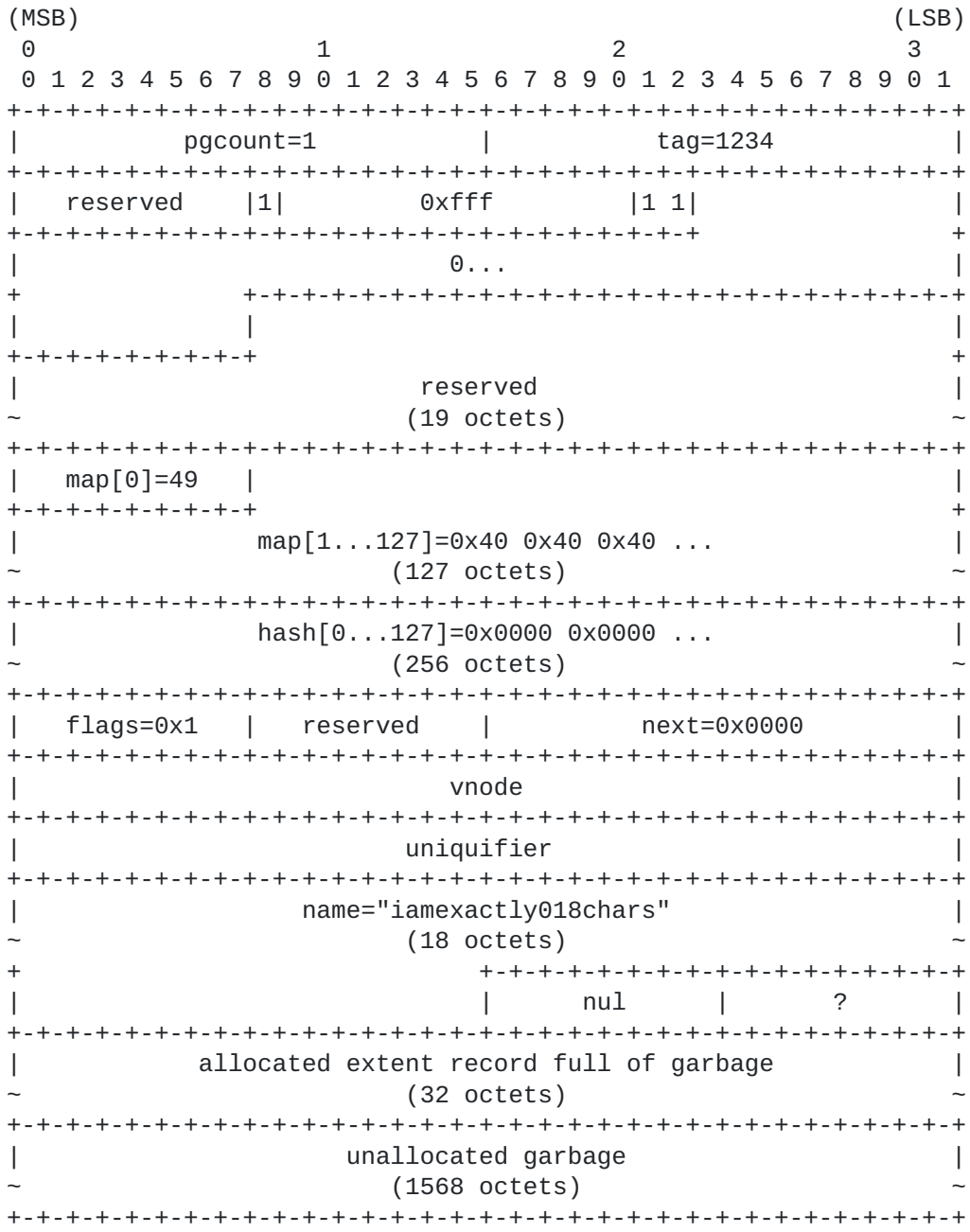
[CMU-ITC-83-025] Morris, J., Van Houweling, D., and K. Slack, "The Information Technology Center", CMU ITC Tech. Rep. CMU-ITC-83-025, 1983.

[CMU-ITC-84-020] West, M., "VICE File System Services", CMU ITC Tech. Rep. CMU-ITC-84-020, August 1984.

[VICE1] Satyanarayanan, M., Howard, J., Nichols, D., Sidebotham, R., Spector, A., and M. West, "The ITC Distributed File System: Principles and Design", Proc. 10th ACM Symp. Operating Sys. Princ. Vol. 19, No. 5, December 1985.

Appendix A. Example Directory Object

A.1. 18-character name string



18-Character Name String

Appendix B. C Implementation of Directory Hash Function

```
#define NHASHENT 128

unsigned int
dir_name_hash(const char * name)
{
    unsigned int hash = 0;

    while(*name++) {
        hash *= 173;
        hash += *name;
    }

    if (hash & 0x80000000) {
        return NHASHENT - (hash & (NHASHENT-1));
    } else {
        return hash & (NHASHENT-1);
    }
}
```

Figure 1

Author's Address

Thomas Keiser
Sine Nomine Associates
43596 Blacksmith Square
Ashburn, VA 20147
USA

Phone: +1 703 723 6673
Email: tkeiser@sinenomine.net