Network Working Group
INTERNET-DRAFT
Intended Status: Informational

Glenn Fowler
Google
Landon Curt Noll
Cisco Systems
Kiem-Phong Vo
Google
Donald Eastlake
Futurewei Technologies
Tony Hansen
AT&T Laboratories

Expires: January 9, 2023
July 10,2022

The FNV Non-Cryptographic Hash Algorithm
<draft-eastlake-fnv-18.txt>

Abstract

FNV (Fowler/Noll/Vo) is a fast, non-cryptographic hash algorithm with good dispersion. The purpose of this document is to make information on FNV and open source code performing FNV conveniently available to the Internet community.

Status of This Document

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

Distribution of this document is unlimited. Comments should be sent to the authors.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

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."

The list of current Internet-Drafts can be accessed at https://www.ietf.org/1id-abstracts.html. The list of Internet-Draft Shadow Directories can be accessed at https://www.ietf.org/shadow.html.

Table of Contents

Table of Contents (continued)

1. Introduction

The FNV hash algorithm is based on an idea sent as reviewer comments to the [IEEE] POSIX P1003.2 committee by Glenn Fowler and Phong Vo in 1991. In a subsequent ballot round Landon Curt Noll suggested an improvement on their algorithm. Some people tried this hash and found that it worked rather well. In an EMail message to Landon, they named it the "Fowler/Noll/Vo" or FNV hash. [FNV]

FNV hashes are designed to be fast while maintaining a low collision rate. The high dispersion of the FNV hashes makes them well suited for hashing nearly identical strings such as URLs, hostnames, filenames, text, IP addresses, etc. Their speed allows one to quickly hash lots of data while maintaining a reasonably low collision rate. However, they are generally not suitable for cryptographic use. (See Section 7.1.)

The FNV hash is widely used, for example in DNS servers, the Twitter service, database indexing hashes, major web search / indexing engines, netnews history file Message-ID lookup functions, anti-spam filters, a spellchecker programmed in Ada 95, flatassembler's open source x86 assembler - user-defined symbol hashtree, non-cryptographic file fingerprints, computing Unique IDs in DASM (DTN (Delay Tolerant Networking) Applications for Symbian Mobile-phones), Microsoft's hash_map implementation for VC++ 2005, the realpath cache in PHP 5.x (php-5.2.3/TSRM/tsrm_virtual_cwd.c), and many other uses.

A study has recommended FNV in connection with the IPv6 Flow Label field [IPv6flow]. There is a proposal to use FNV for BFD sequence number generation [BFDseq].

FNV hash algorithms and source code have been released into the public domain. The authors of the FNV algorithm took deliberate steps to disclose the algorithm in a public forum soon after it was invented. More than a year passed after this public disclosure and the authors deliberately took no steps to patent the FNV algorithm. Therefore, it is safe to say that the FNV authors have no patent claims on the FNV algorithm as published.

If you use an FNV function in an application, you are kindly requested to send an EMail about it to: fnv-mail@asthe.com

2. FNV Basics

This document focuses on the FNV-1a function whose pseudo-code is as follows:

```
hash = offset_basis
for each octet_of_data to be hashed
        hash = hash xor octet_of_data
        hash = hash * FNV_Prime
return hash
```

In the pseudo-code above, hash is a power-of-two number of bits (32, 64, ... 1024) and offset_basis and FNV_Prime depend on the size of hash.

The FNV-1 algorithm is the same, including the values of offset_basis and FNV_Prime, except that the order of the two lines with the "xor" and multiply operations are reversed. Operational experience indicates better hash dispersion for small amounts of data with FNV-1a. FNV-0 is the same as FNV-1 but with offset_basis set to zero. FNV-1a is suggested for general use.

2.1 FNV Primes

The theory behind FNV_Prime's is beyond the scope of this document but the basic property to look for is how an FNV_Prime would impact dispersion. Now, consider any n-bit FNV hash where n is >= 32 and also a power of 2, in particular n = 2**s. For each such n-bit FNV hash, an FNV_Prime p is defined as:

When s is an integer and 4 < s < 11, then FNV_Prime is the smallest prime p of the form:

```
256**int((5 + 2**s)/12) + 2**8 + b
```

where b is an integer such that:

```
0 < b < 2**8
The number of one-bits in b is 4 or 5
```

and where (p mod (2**40 - 2**24 - 1)) > (2**24 + 2**8 + 2**7).

Experimentally, FNV_Primes matching the above constraints tend to have better dispersion properties. They improve the polynomial feedback characteristic when an FNV_Prime multiplies an intermediate hash value. As such, the hash values produced are more scattered throughout the n-bit hash space.

The case where s < 5 is not considered because the resulting hash quality is too low. Such small hashes can, if desired, be derived from a 32 bit FNV hash by XOR folding (see Section 3). The case where s > 10 is not considered because of the doubtful utility of such large FNV hashes and because the criteria for such large FNV_Primes is more complex, due to the sparsity of such large primes, and would needlessly clutter the criteria given above.

Per the above constraints, an FNV_Prime should have only 6 or 7 one-bits in it. Therefore, some compilers may seek to improve the performance of a multiplication with an FNV_Prime by replacing the multiplication with shifts and adds. However, note that the performance of this substitution is highly hardware-dependent and should be done with care. FNV_Primes were selected primarily for the quality of resulting hash function, not for compiler optimization.

2.2 FNV offset_basis

The offset_basis values for the n-bit FNV-1a algorithms are computed by applying the n-bit FNV-0 algorithm to the 32 octets representing the following character string in ASCII [RFC20]:

chongo <Landon Curt Noll> /\../\

The \'s in the above string are not C-style escape characters. In C-string notation, these 32 octets are:

"chongo <Landon Curt Noll> /\\../\\"

That string was used because the person testing FNV with non-zero offset_basis values was looking at an email message from Landon and was copying his standard email signature line; however, they couldn't see very well and copied it incorrectly. In fact, he uses

chongo (Landon Curt Noll) /\oo/\

but, since it doesn't matter, no effort has been made to correct this.

In the general case, almost any offset_basis will serve so long as it is non-zero. The choice of a non-standard offset_basis may be beneficial in defending against some attacks that try to induce hash collisions.

2.3 FNV Endianism

For persistent storage or interoperability between different hardware platforms, an FNV hash shall be represented in the little endian format. That is, the FNV hash will be stored in an array hash[N] with N bytes such that its integer value can be retrieved as follows:

```
unsigned char   hash[N];
for ( i = N-1, value = 0; i >= 0; --i )
    value = ( value << 8 ) + hash[i];
```

Of course, when FNV hashes are used in a single process or a group of processes sharing memory on processors with compatible endian-ness, the natural endian-ness of those processors can be used regardless of its type, little, big, or some other exotic form.

The code provided in Section 6 has FNV hash functions that return a little endian byte vector. Because they are slightly more efficient, code returning FNV hashes of 32-bit or 64-bit size as integers, on computers supporting integers of those sides, are also provided. Such integers are compatible with the same size byte vectors on little endian computers but use of the functions returning integers on big endian or other non-little-endian machines will be byte-reversed or otherwise incompatible with the byte vectors.

3. Other Hash Sizes and XOR Folding

Many hash uses require a hash that is not one of the FNV sizes for which constants are provided in Section 5. If a larger hash size is needed, please contact the authors of this document.

Most hash applications make use of a hash that is a fixed size binary field. Assume that k bits of hash are desired and k is less than 1024 but not one of the sizes for which constants are provided in Section 5. The recommended technique is to take the smallest FNV hash of size S, where S is larger than k, and calculate the desired k-bit-hash using xor folding as shown below. The final bit masking operation is logically unnecessary if the size of the variable k-bit-hash is exactly k bits.

```
temp = FNV_S ( data-to-be-hashed )
k-bit-hash = ( temp xor temp>>k ) bitwise-and ( 2**k - 1 )
```

Hash functions are a trade-off between speed and strength. For example, a somewhat stronger hash may be obtained for exact FNV sizes by calculating an FNV twice as long as the desired output (S = 2*k) and performing such data folding using a k equal to the size of the desired output. However, if a much stronger hash, for example one suitable for cryptographic applications, is wanted, algorithms designed for that purpose, such as those in [RFC6234], should be used.

If it is desired to obtain a hash result that is a value between 0 and max, where max+1 is a not a power of two, simply choose an FNV hash size S such that 2**S > max. Then calculate the following:

```
FNV_S mod ( max+1 )
```

The resulting remainder will be in the range desired but will suffer from a bias against large values with the bias being larger if 2**S is only a little bigger than max. If this bias is acceptable, no further processing is needed. If this bias is unacceptable, it can be avoided by retrying for certain high values of hash, as follows, before applying the mod operation above:

```
X = ( int( ( 2**S - 1 ) / ( max+1 ) ) ) * ( max+1 )
while ( hash >= X )
      hash = ( hash * FNV_Prime ) + offset_basis
```

4. Hashing Multiple Values Together

It is common for there to be a few different component values, say three strings X, Y, and Z, where a hash over all of them is desired. The simplest thing to do is to concatenate them and compute the hash of that concatenation, as in

```
hash ( X | Y | Z )
```

where the vertical bar character ("|") represents string concatenation. Note that, for FNV, the same hash results if X, Y, and Z are actually concatenated and the FNV hash applied to the resulting string or if FNV is calculated on an initial substring and the result used as the offset_basis when calculating the FNV hash of the remainder of the string. This can be done several times. Assuming FNVoffset_basis (v, w) is FNV of w using v as the offset_basis, then in the example above, fnvx = FNV (X) could be calculated and then fnvxy = FNVoffset_basis (fnvx, Y), and finally fnvxyz = FNVoffset_basis (fnvxy, Z). The resulting fnvxyz would be the same as FNV (X | Y | Z);

Cases are also common where such a hash needs to be repeatedly calculated where the component values vary but some vary more frequently than others. For example, assume some sort of computer network traffic flow ID, such as the IPv6 flow ID [RFC6437], is to be calculated for network packets based on the source and destination IPv6 address and the Traffic Class [RFC8200]. If the Flow ID is calculated in the originating host, the source IPv6 address would likely always be the same or perhaps assume one of a very small number of values. By placing this quasi-constant IPv6 source address first in the string being FNV hashed, FNV (IPv6source) could be calculated and used as the offset_basis for calculating FNV of the IPv6 destination address and Traffic Class for each packet. As a result, the per packet hash would be over 17 bytes rather than over 33 bytes saving computational resources. The code in this document includes functions facilitating the use of a non-standard offset_basis.

5. FNV Constants

The FNV Primes are as follows:

```
  32 bit FNV_Prime = 2**24 + 2**8 + 0x93 = 16,777,619
                                         = 0x01000193

  64 bit FNV_Prime = 2**40 + 2**8 + 0xB3 = 1,099,511,628,211
                                       = 0x00000100 000001B3

 128 bit FNV_Prime = 2**88 + 2**8 + 0x3B =
                                 309,485,009,821,345,068,724,781,371
                               = 0x00000000 01000000 00000000 0000013B

 256 bit FNV_Prime = 2**168 + 2**8 + 0x63 =
374,144,419,156,711,147,060,143,317,175,368,453,031,918,731,002,211 =
0x0000000000000000 0000010000000000 0000000000000000 0000000000000163

 512 bit FNV_Prime = 2**344 + 2**8 + 0x57 = 35,
835,915,874,844,867,368,919,076,489,095,108,449,946,327,955,754,392,
558,399,825,615,420,669,938,882,575,126,094,039,892,345,713,852,759 =
0x0000000000000000 0000000000000000 0000000001000000 0000000000000000
  0000000000000000 0000000000000000 0000000000000000 0000000000000157

1024 bit FNV_Prime = 2**680 + 2**8 + 0x8D = 5,
016,456,510,113,118,655,434,598,811,035,278,955,030,765,345,404,790,
744,303,017,523,831,112,055,108,147,451,509,157,692,220,295,382,716,
162,651,878,526,895,249,385,292,291,816,524,375,083,746,691,371,804,
094,271,873,160,484,737,966,720,260,389,217,684,476,157,468,082,573 =
0x0000000000000000 0000000000000000 0000000000000000 0000000000000000
  0000000000000000 0000010000000000 0000000000000000 0000000000000000
  0000000000000000 0000000000000000 0000000000000000 0000000000000000
  0000000000000000 0000000000000000 0000000000000000 000000000000018D
```

The FNV offset_basis values are as follows:

```
  32 bit offset_basis = 2,166,136,261 = 0x811C9DC5

  64 bit offset_basis = 14695981039346656037 = 0xCBF29CE4 84222325

 128 bit offset_basis = 144066263297769815596495629667062367629 =
                          0x6C62272E 07BB0142 62B82175 6295C58D

 256 bit offset_basis = 100,029,257,958,052,580,907,070,968,
620,625,704,837,092,796,014,241,193,945,225,284,501,741,471,925,557 =
0xDD268DBCAAC55036 2D98C384C4E576CC C8B1536847B6BBB3 1023B4C8CAEE0535
```

```
 512 bit offset_basis = 9,
659,303,129,496,669,498,009,435,400,716,310,466,090,418,745,672,637,
896,108,374,329,434,462,657,994,582,932,197,716,438,449,813,051,892,
206,539,805,784,495,328,239,340,083,876,191,928,701,583,869,517,785 =
0xB86DB0B1171F4416 DCA1E50F309990AC AC87D059C9000000 0000000000000D21
  E948F68A34C192F6 2EA79BC942DBE7CE 182036415F56E34B AC982AAC4AFE9FD9

1024 bit offset_basis =  14,197,795,064,947,621,068,722,070,641,403,
218,320,880,622,795,441,933,960,878,474,914,617,582,723,252,296,732,
303,717,722,150,864,096,521,202,355,549,365,628,174,669,108,571,814,
760,471,015,076,148,029,755,969,804,077,320,157,692,458,563,003,215,
304,957,150,157,403,644,460,363,550,505,412,711,285,966,361,610,267,
868,082,893,823,963,790,439,336,411,086,884,584,107,735,010,676,915 =
0x0000000000000000 005F7A76758ECC4D 32E56D5A591028B7 4B29FC4223FDADA1
  6C3BF34EDA3674DA 9A21D90000000000 0000000000000000 0000000000000000
  0000000000000000 0000000000000000 0000000000000000 000000000004C6D7
  EB6E73802734510A 555F256CC005AE55 6BDE8CC9C6A93B21 AFF4B16C71EE90B3
```

6. The Source Code

[THIS CODE IS BEING WORKED ON.]

The following sub-sections provide reference C source code and a test driver for FNV-1a.

Alternative source code, including 32 and 64 bit FNV-1 and FNV-1a in x86 assembler, is currently available at [FNV].

Section 6.2 provides a test driver.

6.1 FNV-1a C Code

This section provides the direct FNV-1a function for each of the lengths for which it is specified in this document. The functions provided are listed below. Those whose name is of the form FNVxxxB* output a byte vector that will be compatible between systems of different endian-ness, where xxx is "32", "64", "128", "256", "512", or "1024". Those whose name is of the form FNVxxx* (with no "B" after the xxx) return an integer and are NOT compatible between systems of different endian-ness. These integer based functions exist only for xxx of "32" and, on systems supporting 64-bit integers, "64".

FNVxxxstring, FNVxxxblock:
FNVxxxBstring, FNVxxxBblock: These are simple functions for directly returning the FNV hash of a zero terminated byte string not including the zero and the FNV hash of a counted block of bytes. Note that for applications of FNV-32 where 32-bit integers are supported and FNV-64 where 64-bit integers are supported and an integer data type output is acceptable, the code is sufficiently simple that, to maximize performance, use of open coding or macros may be more appropriate than calling a subroutine.

FNVxxxinit, FNVxxxinitBasis:
FNVxxxBinit, FNVxxxBinitBasis: These functions and the next two sets of functions below provide facilities for incrementally calculating FNV hashes. They all assume a data structure of type FNVxxx(B)context that holds the current state of the hash. FNVxxx(B)init initializes that context to the standard offset_basis. FNVxxx(B)initBasis takes an offset_basis value as a parameter and may be useful for hashing concatenations, as described in Section 4, as well as for simply using a non-standard offset_basis.

FNVxxxblockin, FNVxxxstringin:
FNVxxxBblockin, FNVxxxBstringin: These functions hash a sequence of bytes into an FNVxxx(B)context that was originally initialized by FNVxxx(B)init or FNVxxx(B)initBasis. FNVxxx(B)blockin hashes in a counted block of bytes. FNVxxx(B)stringin hashes in a zero terminated byte string not incuding the final zero.

FNVxxxresult:
FNVxxxBresult: This function extracts the final FNV hash result from an FNVxxx(B)context.

The following code is a private header file used by all the FNV functions further below and which states the terms for use and redistribution of all of this code.

<CODE BEGINS>

```
/************************ fnv-private.h ************************/
/****************** See RFC NNNN for details ******************/
/* Copyright (c) 2016, 2017 IETF Trust and the persons identified as
 * authors of the code.  All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 *
 * *  Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 *
 * *  Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in
 *    the documentation and/or other materials provided with the
 *    distribution.
 *
 * *  Neither the name of Internet Society, IETF or IETF Trust, nor the
 *    names of specific contributors, may be used to endorse or promote
 *    products derived from this software without specific prior
 *    written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
 * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
 * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
 * FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
 * COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
 * INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,
 * BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
 * LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
 * CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN
 * ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
 * POSSIBILITY OF SUCH DAMAGE.
```

```
 */

#ifndef _FNV_PRIVATE_H_
#define _FNV_PRIVATE_H_

/*
 *      Six FNV-1a hashes are defined with these sizes:
 *              FNV32          32 bits, 4 bytes
 *              FNV64          64 bits, 8 bytes
 *              FNV128         128 bits, 16 bytes
 *              FNV256         256 bits, 32 bytes
 *              FNV512         512 bits, 64 bytes
 *              FNV1024        1024 bits, 128 bytes
 */

/* Private stuff used by this implementation of the FNV
 * (Fowler, Noll, Vo) non-cryptographic hash function FNV-1a.
 * External callers don't need to know any of this.  */

enum {  /* State value bases for context->Computed */
    FNVinited = 22,
    FNVcomputed = 76,
    FNVemptied = 220,
    FNVclobber = 122 /* known bad value for testing */
};

/* Deltas to assure distinct state values for different lengths */
enum {
   FNV32state = 1,
   FNV32Bstate = 17,
   FNV64state = 3,
   FNV64Bstate = 19,
   FNV128state = 5,
   FNV256state = 7,
   FNV512state = 11,
   FNV1024state = 13
};

#endif
```

<CODE ENDS>

The following code is a simple header file to include all the specific length FNV header files.

<CODE BEGINS>

```
/****************************** FNV.h ******************************/
/****************** See RFC NNNN for details. ********************/
/*
 * Copyright (c) 2016, 2017 IETF Trust and the persons identified as
 * authors of the code.  All rights reserved.
```

```
 * See fnv-private.h for terms of use and redistribution.
 */

#ifndef _FNV_H_
#define _FNV_H_

#include "FNV32.h"
#include "FNV32B.h"
#ifdef FNV_64bitIntegers
# include "FNV64.h"
#endif /* FNV_64bitIntegers */
#include "FNV64B.h"
#include "FNV128.h"
#include "FNV256.h"
#include "FNV512.h"
#include "FNV1024.h"

#endif /* _FNV_H_ */
```

<CODE ENDS>

The following code is a simple header file to control configuration related to big integer and big endian support.

<CODE BEGINS>

```
/************************** FNVconfig.h ***************************/
/******************* See RFC NNNN for details. *********************/
/*
 * Copyright (c) 2016, 2017 IETF Trust and the persons identified as
 * authors of the code.  All rights reserved.
 * See fnv-private.h for terms of use and redistribution.
 */

#ifndef _FNVconfig_H_
#define _FNVconfig_H_

/*
 *  Description:
 *      This file provides configuration ifdefs for the
 *      FNV-1a non-cryptographic hash algorithms.
 *
 * >>>>>>>> IMPORTANT CONFIGURATION ifdefs: <<<<<<<<<< */

/*      FNV_64bitIntegers - Define this if your system supports 64-bit
 *          arithmetic including 32-bit x 32-bit multiplication
 *          producing a 64-bit product. If undefined, it will be
 *          assumed that 32-bit arithmetic is supported including
 *          16-bit x 16-bit multiplication producing a 32-bit result.
 */
// #define FNV_64bitIntegers
```

```
/*
 *      The following allow the FNV test program to override the
 *      above configuration settings.
 */

#ifdef FNV_TEST_PROGRAM
# ifdef TEST_FNV_64bitIntegers
#  ifndef FNV_64bitIntegers
#   define FNV_64bitIntegers
#  endif
# else
#  undef FNV_64bitIntegers
# endif
# ifndef FNV_64bitIntegers /* causes an error if uint64_t is used */
#  define uint64_t foobar /* RFC 3092 */
# endif
#endif

#endif /* _FNVconfig_H_ */
```

<CODE ENDS>

6.1.1 FNV32 Code

The header and C source for 32-bit FNV-1a returning a 32-bit integer.

<CODE BEGINS>

```
/***************************** FNV32.h ******************************/
/******************** See RFC NNNN for details **********************/
/*
 * Copyright (c) 2016 IETF Trust and the persons identified as
 * authors of the code.  All rights reserved.
 * See fnv-private.h for terms of use and redistribution.
*/

#ifndef _FNV32_H_
#define _FNV32_H_

#include "FNVconfig.h"

#include <stdint.h>
/* #define FNV32size (32/8)   not used */

/* If you do not have the ISO standard stdint.h header file, then you
 * must typedef the following types:
 *
 *    type              meaning
 *  uint32_t         unsigned 32 bit integer
 *  uint8_t          unsigned 8 bit integer (i.e., unsigned char)
```

```
 */

#ifndef _FNV_ErrCodes_
#define _FNV_ErrCodes_
/*********************************************************************
 *  All FNV functions provided return as integer as follows:
 *       0 -> success
 *      >0 -> error as listed below
 */
enum {    /* success and errors */
    fnvSuccess = 0,
    fnvNull,            /* Null pointer parameter */
    fnvStateError,      /* called Input after Result, etc. */
    fnvBadParam         /* passed a bad parameter */
};
#endif /* _FNV_ErrCodes_ */

/*
 *  This structure holds context information for an FNV32 hash
 */
typedef struct FNV32context_s {
    int Computed;  /* state */
    uint32_t Hash;
} FNV32context;

/*
 *  Function Prototypes
 *    FNV32string: hash a zero terminated string not including
 *                 the terminating zero
 *    FNV32block: hash a specified length byte vector
 *    FNV32init:  initializes an FNV32 context
 *    FNV32initBasis: initializes an FNV32 context with a
 *                    provided basis
 *    FNV32blockin:  hash in a specified length byte vector
 *    FNV32stringin: hash in a zero terminated string not
 *                   including the zero
 *    FNV32result: returns the hash value
 *
 *    Hash is returned as a 32-bit integer
 */

#ifdef __cplusplus
extern "C" {
#endif

/* FNV32 */
extern int FNV32string ( const char *in,
                         uint32_t * const out );
extern int FNV32block ( const void *in,
                        long int inlength,
```

```
                        uint32_t * const out );
extern int FNV32init ( FNV32context * const );
extern int FNV32initBasis ( FNV32context * const,
                            uint32_t basis );
extern int FNV32blockin ( FNV32context * const,
                          const void *in,
                          long int inlength );
extern int FNV32stringin ( FNV32context * const,
                           const char *in );
extern int FNV32result ( FNV32context * const,
                         uint32_t * const out );

#ifdef __cplusplus
}
#endif

#endif /* _FNV32_H_ */
```

<CODE ENDS>

<CODE BEGINS>

```
/***************************** FNV32.c ****************************/
/****************** See RFC NNNN for details. *********************/
/* Copyright (c) 2016, 2017 IETF Trust and the persons identified
 * as authors of the code.  All rights reserved.
 * See fnv-private.h for terms of use and redistribution.
 */

/* This code implements the FNV (Fowler, Noll, Vo) non-cryptographic
 * hash function FNV-1a for 32-bit hashes returning a 32-bit
 * integer.
 */

#ifndef _FNV32_C_
#define _FNV32_C_

#include "fnv-private.h"
#include "FNV32.h"

/* 32 bit FNV_prime = 2^24 + 2^8 + 0x93 */
#define FNV32prime 0x01000193
#define FNV32basis 0x811C9DC5

/* FNV32 hash a zero terminated string not including the zero
*****************************************************************/
int FNV32string ( const char *in, uint32_t * const out )
{
uint32_t    temp;
uint8_t     ch;

if ( in && out )
```

```
    {
    temp = FNV32basis;
    while ( (ch = *in++) )
        temp = FNV32prime * ( temp ^ ch );
    *out = temp;
    return fnvSuccess;
    }
return fnvNull; /* Null input pointer */
}   /* end FNV32string */

/* FNV32 hash a counted block
 *************************************************************/
int FNV32block ( const void *vin,
                 long int length,
                 uint32_t * const out )
{
const uint8_t *in = (const uint8_t*)vin;
uint32_t    temp;

if ( in && out )
    {
    if ( length < 0 )
        return fnvBadParam;
    for ( temp = FNV32basis; length > 0; length-- )
        temp = FNV32prime * ( temp ^ *in++ );
    *out = temp;
    return fnvSuccess;
    }
return fnvNull; /* Null input pointer */
}   /* end FNV32block */


/*************************************************************
 *       Set of init, input, and output functions below      *
 *       to incrementally compute FNV32                      *
 *************************************************************/

/* initialize context
 *************************************************************/
int FNV32init ( FNV32context * const ctx )
{
return FNV32initBasis ( ctx, FNV32basis );
}   /* end FNV32init */

/* initialize context with a provided basis
 *************************************************************/
int FNV32initBasis ( FNV32context * const ctx, uint32_t basis )
{
if ( ctx )
    {
```

```
    ctx->Hash = basis;
    ctx->Computed = FNVinited+FNV32state;
    return fnvSuccess;
    }
return fnvNull;
}   /* end FNV32initBasis */

/* hash in a counted block
 *************************************************************/
int FNV32blockin ( FNV32context * const ctx,
                   const void *vin,
                   long int length )
{
const uint8_t *in = (const uint8_t*)vin;
uint32_t    temp;

if ( ctx && in )
    {
    if ( length < 0 )
        return fnvBadParam;
    switch ( ctx->Computed )
        {
        case FNVinited+FNV32state:
            ctx->Computed = FNVcomputed+FNV32state;
        case FNVcomputed+FNV32state:
            break;
        default:
            return fnvStateError;
        }
    for ( temp = ctx->Hash; length > 0; length-- )
        temp = FNV32prime * ( temp ^ *in++ );
    ctx->Hash = temp;
    return fnvSuccess;
    }
return fnvNull;
}   /* end FNV32blockin */

/* hash in a zero terminated string not including the zero
 *************************************************************/
int FNV32stringin ( FNV32context * const ctx,
                    const char *in )
{
uint32_t    temp;
uint8_t     ch;

if ( ctx && in )
    {
    switch ( ctx->Computed )
        {
        case FNVinited+FNV32state:
```

```
            ctx->Computed = FNVcomputed+FNV32state;
        case FNVcomputed+FNV32state:
            break;
        default:
            return fnvStateError;
        }
    temp = ctx->Hash;
    while ( (ch = (uint8_t)*in++) )
        temp = FNV32prime * ( temp ^ ch );
    ctx->Hash = temp;
    return fnvSuccess;
    }
return fnvNull;
}   /* end FNV32stringin */

/* return hash
 *************************************************************/
int FNV32result ( FNV32context * const ctx,
                  uint32_t * const out )
{
if ( ctx && out )
    {
    if ( ctx->Computed != FNVcomputed+FNV32state )
        return fnvStateError;
    ctx->Computed = FNVemptied+FNV32state;
    *out = ctx->Hash;
    ctx->Hash = 0;
    return fnvSuccess;
    }
return fnvNull;
}   /* end FNV32result */

#endif    /* _FNV32_C_ */
```

<CODE ENDS>

The header and C source for 32-bit FNV-1a returning a byte vector.

<CODE BEGINS>

```
/***************************** FNV32B.h *****************************/
/******************** See RFC NNNN for details **********************/
/*
 * Copyright (c) 2016, 2017 IETF Trust and the persons identified as
 * authors of the code.  All rights reserved.
 * See fnv-private.h for terms of use and redistribution.
*/

#ifndef _FNV32_H_
#define _FNV32_H_

#include "FNVconfig.h"
```

```
#include <stdint.h>
#define FNV32size (32/8)

/* If you do not have the ISO standard stdint.h header file, then you
 * must typedef the following types:
 *
 *    type              meaning
 *  uint32_t         unsigned 32 bit integer
 *  uint8_t          unsigned 8 bit integer (i.e., unsigned char)
 */

#ifndef _FNV_ErrCodes_
#define _FNV_ErrCodes_
/*********************************************************************
 *  All FNV functions provided return as integer as follows:
 *       0 -> success
 *      >0 -> error as listed below
 */
enum {    /* success and errors */
    fnvSuccess = 0,
    fnvNull,            /* Null pointer parameter */
    fnvStateError,      /* called Input after Result, etc. */
    fnvBadParam         /* passed a bad parameter */
};
#endif /* _FNV_ErrCodes_ */

/*
 *  This structure holds context information for an FNV32 hash
 */
typedef struct FNV32Bcontext_s {
    int Computed;  /* state */
    uint32_t Hash;
} FNV32Bcontext;

/*
 *  Function Prototypes
 *    FNV32Bstring: hash a zero terminated string not including
 *                  the terminating zero
 *    FNV32Bblock: hash a specified length byte vector
 *    FNV32Binit:  initializes an FNV32 context
 *    FNV32BinitBasis: initializes an FNV32 context with a
 *                     provided basis
 *    FNV32Bblockin:  hash in a specified length byte vector
 *    FNV32Bstringin: hash in a zero terminated string not
 *                    including the zero
 *    FNV32Bresult: returns the hash value
 *
 *    Hash is returned as a 32-bit integer
 */
```

```
#ifdef __cplusplus
extern "C" {
#endif

/* FNV32 */
extern int FNV32Bstring ( const char *in,
                          uint8_t * const out[FNV32size] );
extern int FNV32Bblock ( const void *in,
                         long int inlength,
                         uint8_t * const out[FNV32size] );
extern int FNV32Binit ( FNV32Bcontext * const );
extern int FNV32BinitBasis ( FNV32Bcontext * const,
                             uint32_t basis );
extern int FNV32Bblockin ( FNV32Bcontext * const,
                           const void *in,
                           long int inlength );
extern int FNV32Bstringin ( FNV32Bcontext * const,
                            const char *in );
extern int FNV32Bresult ( FNV32Bcontext * const,
                          int8_t * const out[FNV32size] );

#ifdef __cplusplus
}
#endif

#endif /* _FNV32_H_ */
   <CODE ENDS>

   <CODE BEGINS>
/***************************** FNV32B.c ***************************/
/****************** See RFC NNNN for details. ********************/
/* Copyright (c) 2016, 2017 IETF Trust and the persons identified as
 * authors of the code.  All rights reserved.
 * See fnv-private.h for terms of use and redistribution.
 */

/* This code implements the FNV (Fowler, Noll, Vo) non-cryptographic
 * hash function FNV-1a for 32-bit hashes returning a byte vector.
 */

#ifndef _FNV32_C_
#define _FNV32_C_

#include "fnv-private.h"
#include "FNV32B.h"

/* 32 bit FNV_prime = 2^24 + 2^8 + 0x93 */
#define FNV32prime 0x01000193
#define FNV32basis 0x811C9DC5
```

```
/* FNV32 hash a zero terminated string not including the zero
*****************************************************************/
int FNV32Bstring ( const char *in, uint8_t * const out[FNV32size] )
{
uint32_t    temp;
uint8_t     ch;

if ( in && out )
    {
    temp = FNV32basis;
    while ( (ch = *in++) )
        temp = FNV32prime * ( temp ^ ch );
    *out[0] = temp & 0xFF;
    temp =>> 8;
    *out[1] = temp & 0xFF;
    temp =>> 8;
    *out[2] = temp & 0xFF;
    temp =>> 8;
    *out[3] = temp & 0xFF;
    return fnvSuccess;
    }
return fnvNull; /* Null input pointer */
}   /* end FNV32Bstring */

/* FNV32 hash a counted block
 ****************************************************************/
int FNV32Bblock ( const void *vin,
                  long int length,
                  uint8_t * const out[FNV32size] )
{
const uint8_t *in = (const uint8_t*)vin;
uint32_t    temp;

if ( in && out )
    {
    if ( length < 0 )
        return fnvBadParam;
    for ( temp = FNV32basis; length > 0; length-- )
        temp = FNV32prime * ( temp ^ *in++ );
    *out[0] = temp & 0xFF;
    temp =>> 8;
    *out[1] = temp & 0xFF;
    temp =>> 8;
    *out[2] = temp & 0xFF;
    temp =>> 8;
    *out[3] = temp & 0xFF;
    *out = temp;
    return fnvSuccess;
    }
return fnvNull; /* Null input pointer */
```

```
}   /* end FNV32Bblock */


/*************************************************************
 *       Set of init, input, and output functions below      *
 *       to incrementally compute FNV32                      *
 *************************************************************/

/* initialize context
 *************************************************************/
int FNV32Binit ( FNV32Bcontext * const ctx )
{
return FNV32BinitBasis ( ctx, FNV32basis );
}   /* end FNV32Binit */

/* initialize context with a provided basis
 *************************************************************/
int FNV32BinitBasis ( FNV32Bcontext * const ctx, uint32_t basis )
{
if ( ctx )
    {
    ctx->Hash = basis;
    ctx->Computed = FNVinited+FNV32Bstate;
    return fnvSuccess;
    }
return fnvNull;
}   /* end FNV32BinitBasis */

/* hash in a counted block
 *************************************************************/
int FNV32Bblockin ( FNV32Bcontext * const ctx,
                    const void *vin,
                    long int length )
{
const uint8_t *in = (const uint8_t*)vin;
uint32_t    temp;

if ( ctx && in )
    {
    if ( length < 0 )
        return fnvBadParam;
    switch ( ctx->Computed )
        {
        case FNVinited+FNV32Bstate:
            ctx->Computed = FNVcomputed+FNV32Bstate;
        case FNVcomputed+FNV32Bstate:
            break;
        default:
            return fnvStateError;
        }
```

```
    for ( temp = ctx->Hash; length > 0; length-- )
        temp = FNV32prime * ( temp ^ *in++ );
    ctx->Hash = temp;
    return fnvSuccess;
    }
return fnvNull;
}   /* end FNV32Bblockin */

/* hash in a zero terminated string not including the zero
 ************************************************************/
int FNV32Bstringin ( FNV32Bcontext * const ctx,
                     const char *in )
{
uint32_t    temp;
uint8_t     ch;

if ( ctx && in )
    {
    switch ( ctx->Computed )
        {
        case FNVinited+FNV32Bstate:
            ctx->Computed = FNVcomputed+FNV32Bstate;
        case FNVcomputed+FNV32Bstate:
            break;
        default:
            return fnvStateError;
        }
    temp = ctx->Hash;
    while ( (ch = (uint8_t)*in++) )
        temp = FNV32prime * ( temp ^ ch );
    ctx->Hash = temp;
    return fnvSuccess;
    }
return fnvNull;
}   /* end FNV32Bstringin */

/* return hash
 ************************************************************/
int FNV32Bresult ( FNV32Bcontext * const ctx,
                   uint8_t * const out[FNV32size] )
{
if ( ctx && out )
    {
    if ( ctx->Computed != FNVcomputed+FNV32Bstate )
        return fnvStateError;
    ctx->Computed = FNVemptied+FNV32Bstate;
    *out[0] = ctx->Hash & 0xFF;
    ctx->Hash =>> 8;
    *out[1] = ctx->Hash & 0xFF;
    ctx->Hash =>> 8;
```

```
    *out[2] = ctx->Hash & 0xFF;
    ctx->Hash =>> 8;
    *out[3] = ctx->Hash & 0xFF;
    ctx->Hash = 0;
    return fnvSuccess;
    }
return fnvNull;
}   /* end FNV32Bresult */

#endif    /* _FNV32_C_ */
```

<CODE ENDS>

6.1.2 FNV64 C Code

The header and C source for 64-bit FNV-1a returning a 64-bit integer.

<CODE BEGINS>

```
/***************************** FNV64.h *****************************/
/****************** See RFC NNNN for details. ********************/
/*
 * Copyright (c) 2016 IETF Trust and the persons identified as
 * authors of the code.  All rights reserved.
 * See fnv-private.h for terms of use and redistribution.
 */

#ifndef _FNV64_H_
#define _FNV64_H_

/*
 *  Description:
 *      This file provides headers for the 64-bit version of the FNV-1a
 *      non-cryptographic hash algorithm.
 */

#include "FNVconfig.h"

#include <stdint.h>
#define FNV64size (64/8)

/* If you do not have the ISO standard stdint.h header file, then you
 * must typedef the following types:
 *
 *    type             meaning
 *  uint64_t        unsigned 64 bit integer (ifdef FNV_64bitIntegers)
 *  uint32_t        unsigned 32 bit integer
 *  uint16_t        unsigned 16 bit integer
 *  uint8_t         unsigned 8 bit integer (i.e., unsigned char)
 */
```

```
#ifndef _FNV_ErrCodes_
#define _FNV_ErrCodes_
/*********************************************************************
 *  All FNV functions provided return as integer as follows:
 *       0 -> success
 *      >0 -> error as listed below
 */
enum {    /* success and errors */
    fnvSuccess = 0,
    fnvNull,            /* Null pointer parameter */
    fnvStateError,      /* called Input after Result, etc. */
    fnvBadParam         /* passed a bad parameter */
};
#endif /* _FNV_ErrCodes_ */

/*
 *  This structure holds context information for an FNV64 hash
 */
#ifdef FNV_64bitIntegers
    /* version if 64 bit integers supported */

typedef struct FNV64context_s {
        int Computed;  /* state */
        uint64_t Hash;
} FNV64context;

#else
    /* version if 64 bit integers NOT supported */

typedef struct FNV64context_s {
        int Computed;  /* state */
        uint16_t Hash[FNV64size/2];
} FNV64context;

#endif /* FNV_64bitIntegers */

/*
 *  Function Prototypes
 *    FNV64string: hash a zero terminated string not including
 *                 the terminating zero
 *    FNV64block: FNV64 hash a specified length byte vector
 *    FNV64init: initializes an FNV64 context
 *    FNV64initBasis: initializes an FNV64 context with a
 *                    provided basis
 *    FNV64blockin: hash in a specified length byte vector
 *    FNV64stringin: hash in a zero terminated string not
 *                   incluing the zero
 *    FNV64result: returns the hash value
 *
 *    Hash is returned as a 64-bit integer if supported, otherwise
```

```
 *         as a vector of 8-bit integers
 */

#ifdef __cplusplus
extern "C" {
#endif

/* FNV64 */
extern int FNV64init ( FNV64context * const );
extern int FNV64blockin ( FNV64context * const,
                          const void * in,
                          long int length );
extern int FNV64stringin ( FNV64context * const,
                           const char * in );

#ifdef FNV_64bitIntegers
  extern int FNV64string ( const char *in,
                           uint64_t * const out );
  extern int FNV64block ( const void *in,
                          long int length,
                          uint64_t * const out );
  extern int FNV64initBasis ( FNV64context * const,
                            uint64_t basis );
  extern int FNV64result ( FNV64context * const,
                           uint64_t * const out );
#else
  extern int FNV64string ( const char *in,
                           uint8_t out[FNV64size] );
  extern int FNV64block ( const void *in,
                          long int length,
                          uint8_t out[FNV64size] );
  extern int FNV64initBasis ( FNV64context * const,
                            const uint8_t basis[FNV64size] );
  extern int FNV64result ( FNV64context * const,
                           uint8_t out[FNV64size] );
#endif /* FNV_64bitIntegers */

#ifdef __cplusplus
}
#endif

#endif /* _FNV64_H_ */
   <CODE ENDS>

   <CODE BEGINS>
/***************************** FNV64.c ******************************/
/******************** See RFC NNNN for details **********************/
/* Copyright (c) 2016 IETF Trust and the persons identified as
 * authors of the code.  All rights reserved.
 * See fnv-private.h for terms of use and redistribution.
```

```
 */

/* This file implements the FNV (Fowler, Noll, Vo) non-cryptographic
 * hash function FNV-1a for 64-bit hashes.
 */

#ifndef _FNV64_C_
#define _FNV64_C_

#include "fnv-private.h"
#include "FNV64.h"

/*****************************************************************
 *        START VERSION FOR WHEN YOU HAVE 64 BIT ARITHMETIC       *
 *****************************************************************/
#ifdef FNV_64bitIntegers

/* 64 bit FNV_prime = 2^40 + 2^8 + 0xb3 */
#define FNV64prime 0x00000100000001B3
#define FNV64basis 0xCBF29CE484222325

/* FNV64 hash a null terminated string  (64 bit)
 *****************************************************************/
int FNV64string ( const char *in, uint64_t * const out )
{
uint64_t    temp;
uint8_t     ch;

if ( in && out )
    {
    temp = FNV64basis;
    while ( (ch = *in++) )
        temp = FNV64prime * ( temp ^ ch );
#ifdef FNV_BigEndian
    FNV64reverse ( out, temp );
#else
    *out = temp;
#endif
    return fnvSuccess;
    }
return fnvNull; /* Null input pointer */
}   /* end FNV64string */

/* FNV64 hash a counted block  (64 bit)
 *****************************************************************/
int FNV64block ( const void *vin,
                 long int length,
                 uint64_t * const out )
{
const uint8_t *in = (const uint8_t*)vin;
```

```
uint64_t    temp;

if ( in && out )
    {
    if ( length < 0 )
        return fnvBadParam;
    for ( temp = FNV64basis; length > 0; length-- )
        temp = FNV64prime * ( temp ^ *in++ );
#ifdef FNV_BigEndian
    FNV64reverse ( out, temp );
#else
    *out = temp;
#endif
    return fnvSuccess;
    }
return fnvNull; /* Null input pointer */
}   /* end FNV64block */

#ifdef FNV_BigEndian

/* Store a Big Endian result back as Little Endian
 ************************************************************/
static void FNV64reverse ( uint64_t *out, uint64_t hash )
{
uint64_t    temp;
int         i;

temp = hash & 0xFF;
for ( i = FNV64size - 1; i > 0; i-- )
    {
    hash >>= 8;
    temp = ( temp << 8 ) + ( hash & 0xFF );
    }
*out = temp;
}   /* end FNV64reverse */

#endif /* FNV_BigEndian */


/*********************************************************************
 *         Set of init, input, and output functions below            *
 *         to incrementally compute FNV64                            *
 *********************************************************************/

/* initialize context  (64 bit)
 ********************************************************************/
int FNV64init ( FNV64context * const ctx )
{
return FNV64initBasis ( ctx, FNV64basis );
}       /* end FNV64init */
```

```
/* initialize context with a provided basis  (64 bit)
 **********************************************************************/
int FNV64initBasis ( FNV64context * const ctx, uint64_t basis )
{
if ( ctx )
    {
    ctx->Hash = basis;
    ctx->Computed = FNVinited+FNV64state;
    return fnvSuccess;
    }
return fnvNull;
}       /* end FNV64initBasis */

/* hash in a counted block  (64 bit)
 **********************************************************************/
int FNV64blockin ( FNV64context * const ctx,
                   const void *vin,
                   long int length )
{
const uint8_t *in = (const uint8_t*)vin;
uint64_t    temp;

if ( ctx && in )
    {
    if ( length < 0 )
        return fnvBadParam;
    switch ( ctx->Computed )
        {
        case FNVinited+FNV64state:
            ctx->Computed = FNVcomputed+FNV64state;
        case FNVcomputed+FNV64state:
            break;
        default:
            return fnvStateError;
        }
    for ( temp = ctx->Hash; length > 0; length-- )
        temp = FNV64prime * ( temp ^ *in++ );
    ctx->Hash = temp;
    return fnvSuccess;
    }
return fnvNull;
}   /* end FNV64input */

/* hash in a zero terminated string not including the zero (64 bit)
 **********************************************************************/
int FNV64stringin ( FNV64context * const ctx,
                    const char *in )
{
uint64_t        temp;
uint8_t         ch;
```

```
if ( ctx && in )
    {
    switch ( ctx->Computed )
        {
        case FNVinited+FNV64state:
            ctx->Computed = FNVcomputed+FNV64state;
        case FNVcomputed+FNV64state:
            break;
        default:
             return fnvStateError;
         }
    temp = ctx->Hash;
    while ( (ch = *in++) )
        temp = FNV64prime * ( temp ^ ch );
    ctx->Hash = temp;
    return fnvSuccess;
    }
return fnvNull;
}   /* end FNV64stringin */

/* return hash  (64 bit)
 ***********************************************************************/
int FNV64result ( FNV64context * const ctx,
                  uint64_t * const out )
{
if ( ctx && out )
    {
    if ( ctx->Computed != FNVcomputed+FNV64state )
        return fnvStateError;
    ctx->Computed = FNVemptied+FNV64state;
#ifdef FNV_BigEndian
    FNV64reverse ( out, ctx->Hash );
#else
    *out = ctx->Hash;
#endif
    ctx->Hash = 0;
    return fnvSuccess;
    }
return fnvNull;
}   /* end FNV64result */

/*****************************************************************
 *        END VERSION FOR WHEN YOU HAVE 64 BIT ARITHMETIC        *
 *****************************************************************/
#else    /*  FNV_64bitIntegers */
/*****************************************************************
 *     START VERSION FOR WHEN YOU ONLY HAVE 32-BIT ARITHMETIC    *
 *****************************************************************/

/* 64 bit FNV_prime = 2^40 + 2^8 + 0xb3 */
```

```
/* #define FNV64prime 0x00000100000001B3 */
#define FNV64primeX 0x01B3
#define FNV64shift 8

/* #define FNV64basis 0xCBF29CE484222325 */
#define FNV64basis0 0xCBF2
#define FNV64basis1 0x9CE4
#define FNV64basis2 0x8422
#define FNV64basis3 0x2325

/* FNV64 hash a null terminated string  (32 bit)
 ****************************************************************/
int FNV64string ( const char *in, uint8_t out[FNV64size] )
{
FNV64context     ctx;
int              err;

 if ( ( err = FNV64init (&ctx) ) != fnvSuccess )
    return err;
 if ( ( err = FNV64stringin (&ctx, in) ) != fnvSuccess )
    return err;
return FNV64result (&ctx, out);
}   /* end FNV64string */

/* FNV64 hash a counted block  (32 bit)
 ****************************************************************/
int FNV64block ( const void *in,
                 long int length,
                 uint8_t out[FNV64size] )
{
FNV64context     ctx;
int              err;

 if ( ( err = FNV64init (&ctx) ) != fnvSuccess )
    return err;
 if ( ( err = FNV64blockin (&ctx, in, length) ) != fnvSuccess )
    return err;
return FNV64result (&ctx, out);
}   /* end FNV64block */


/*****************************************************************
 *         Set of init, input, and output functions below        *
 *         to incrementally compute FNV64                        *
 *****************************************************************/

/* initialize context  (32 bit)
 ****************************************************************/
int FNV64init ( FNV64context * const ctx )
{
```

```
if ( ctx )
    {
    ctx->Hash[0] = FNV64basis0;
    ctx->Hash[1] = FNV64basis1;
    ctx->Hash[2] = FNV64basis2;
    ctx->Hash[3] = FNV64basis3;
    ctx->Computed = FNVinited+FNV64state;
    return fnvSuccess;
    }
return fnvNull;
}   /* end FNV64init */

/* initialize context  (32 bit)
 ****************************************************************/
int FNV64initBasis ( FNV64context * const ctx,
                     const uint8_t basis[FNV64size] )
{
if ( ctx )
    {
#ifdef FNV_BigEndian
    ctx->Hash[0] = basis[1] + ( basis[0]<<8 );
    ctx->Hash[1] = basis[3] + ( basis[2]<<8 );
    ctx->Hash[2] = basis[5] + ( basis[4]<<8 );
    ctx->Hash[3] = basis[7] + ( basis[6]<<8 );
#else
    ctx->Hash[0] = basis[0] + ( basis[1]<<8 );
    ctx->Hash[1] = basis[2] + ( basis[3]<<8 );
    ctx->Hash[2] = basis[4] + ( basis[5]<<8 );
    ctx->Hash[3] = basis[6] + ( basis[7]<<8 );
#endif
    ctx->Computed = FNVinited+FNV64state;
    return fnvSuccess;
    }
return fnvNull;
}   /* end FNV64initBasis */

/* hash in a counted block  (32 bit)
 ****************************************************************/
int FNV64blockin ( FNV64context * const ctx,
                   const void *vin,
                   long int length )
{
const uint8_t *in = (const uint8_t*)vin;
uint32_t   temp[FNV64size/2];
uint32_t   temp2[2];
int        i;

if ( ctx && in )
    {
    if ( length < 0 )
```

```
        return fnvBadParam;
    switch ( ctx->Computed )
        {
        case FNVinited+FNV64state:
            ctx->Computed = FNVcomputed+FNV64state;
        case FNVcomputed+FNV64state:
            break;
        default:
            return fnvStateError;
        }
    for ( i=0; i<FNV64size/2; ++i )
         temp[i] = ctx->Hash[i];
    for ( ; length > 0; length-- )
        {
        /* temp = FNV64prime * ( temp ^ *in++ ); */
        temp2[1] = temp[3] << FNV64shift;
        temp2[0] = temp[2] << FNV64shift;
        temp[3] = FNV64primeX * ( temp[3] ^ *in++ );
        temp[2] *= FNV64primeX;
        temp[1] = temp[1] * FNV64primeX + temp2[1];
        temp[0] = temp[0] * FNV64primeX + temp2[0];
        temp[2] += temp[3] >> 16;
        temp[3] &= 0xFFFF;
        temp[1] += temp[2] >> 16;
        temp[2] &= 0xFFFF;
        temp[0] += temp[1] >> 16;
        temp[1] &= 0xFFFF;
        }
    for ( i=0; i<FNV64size/2; ++i )
        ctx->Hash[i] = temp[i];
    return fnvSuccess;
    }
return fnvNull;
}   /* end FNV64blockin */

/* hash in a string  (32 bit)
 ***************************************************************/
int FNV64stringin ( FNV64context * const ctx,
                    const char *in )
{
uint32_t   temp[FNV64size/2];
uint32_t   temp2[2];
int        i;
uint8_t    ch;

if ( ctx && in )
    {
    switch ( ctx->Computed )
        {
        case FNVinited+FNV64state:
```

```
            ctx->Computed = FNVcomputed+FNV64state;
        case FNVcomputed+FNV64state:
            break;
        default:
             return fnvStateError;
         }
    for ( i=0; i<FNV64size/2; ++i )
         temp[i] = ctx->Hash[i];
    while ( ( ch = (uint8_t)*in++ ) != 0)
        {
        /* temp = FNV64prime * ( temp ^ ch ); */
        temp2[1] = temp[3] << FNV64shift;
        temp2[0] = temp[2] << FNV64shift;
        temp[3] = FNV64primeX * ( temp[3] ^ *in++ );
        temp[2] *= FNV64primeX;
        temp[1] = temp[1] * FNV64primeX + temp2[1];
        temp[0] = temp[0] * FNV64primeX + temp2[0];
        temp[2] += temp[3] >> 16;
        temp[3] &= 0xFFFF;
        temp[1] += temp[2] >> 16;
        temp[2] &= 0xFFFF;
        temp[0] += temp[1] >> 16;
        temp[1] &= 0xFFFF;
        }
    for ( i=0; i<FNV64size/2; ++i )
        ctx->Hash[i] = temp[i];
    return fnvSuccess;
    }
return fnvNull;
}   /* end FNV64stringin */

/* return hash  (32 bit)
 *****************************************************************/
int FNV64result ( FNV64context * const ctx,
                  uint8_t out[FNV64size] )
{
int    i;

if ( ctx && out )
    {
    if ( ctx->Computed != FNVcomputed+FNV64state )
        return fnvStateError;
    for ( i=0; i<FNV64size/2; ++i )
        {
#ifdef FNV_BigEndian
        out[7-2*i] = ctx->Hash[i];
        out[6-2*i] = ctx->Hash[i] >> 8;
#else
        out[2*i] = ctx->Hash[i];
        out[2*i+1] = ctx->Hash[i] >> 8;
```

```
#endif
        ctx -> Hash[i] = 0;
        }
    ctx->Computed = FNVemptied+FNV64state;
    return fnvSuccess;
    }
return fnvNull;
}   /* end FNV64result */

#endif    /*  FNV_64bitIntegers */
/*********************************************************************
 *        END VERSION FOR WHEN YOU ONLY HAVE 32-BIT ARITHMETIC      *
 *********************************************************************/

#endif    /* _FNV64_C_ */
```

<CODE ENDS>

The header and C source for 64-bit FNV-1a returning a byte vector.

<CODE BEGINS>

```
/****************************** FNV64B.h *****************************/
/****************** See RFC NNNN for details. ***********************/
/*
 * Copyright (c) 2016, 2017 IETF Trust and the persons identified as
 * authors of the code.  All rights reserved.
 * See fnv-private.h for terms of use and redistribution.
 */

#ifndef _FNV64_H_
#define _FNV64_H_

/*
 *  Description:
 *      This file provides headers for the 64-bit version of the FNV-1a
 *      non-cryptographic hash algorithm.
 */

#include "FNVconfig.h"

#include <stdint.h>
#define FNV64size (64/8)

/* If you do not have the ISO standard stdint.h header file, then you
 * must typedef the following types:
 *
 *    type             meaning
 *  uint64_t        unsigned 64 bit integer (ifdef FNV_64bitIntegers)
 *  uint32_t        unsigned 32 bit integer
 *  uint16_t        unsigned 16 bit integer
 *  uint8_t         unsigned 8 bit integer (i.e., unsigned char)
```

```
 */

#ifndef _FNV_ErrCodes_
#define _FNV_ErrCodes_
/*********************************************************************
 *  All FNV functions provided return as integer as follows:
 *       0 -> success
 *      >0 -> error as listed below
 */
enum {    /* success and errors */
    fnvSuccess = 0,
    fnvNull,            /* Null pointer parameter */
    fnvStateError,      /* called Input after Result, etc. */
    fnvBadParam         /* passed a bad parameter */
};
#endif /* _FNV_ErrCodes_ */

/*
 *  This structure holds context information for an FNV64 hash
 */
#ifdef FNV_64bitIntegers
    /* version if 64 bit integers supported */

typedef struct FNV64Bcontext_s {
        int Computed;  /* state */
        uint64_t Hash;
} FNV64Bcontext;

#else
    /* version if 64 bit integers NOT supported */

typedef struct FNV64Bcontext_s {
        int Computed;  /* state */
        uint16_t Hash[FNV64size/2];
} FNV64Bcontext;

#endif /* FNV_64bitIntegers */

/*
 *  Function Prototypes
 *    FNV64Bstring: hash a zero terminated string not including
 *                  the terminating zero
 *    FNV64Bblock: FNV64 hash a specified length byte vector
 *    FNV64Binit: initializes an FNV64 context
 *    FNV64BinitBasis: initializes an FNV64 context with a
 *                     provided basis
 *    FNV64Bblockin: hash in a specified length byte vector
 *    FNV64Bstringin: hash in a zero terminated string not
 *                    incluing the zero
 *    FNV64Bresult: returns the hash value
```

```
 *
 *    Hash is returned as a vector of 8-bit integers
 */

#ifdef __cplusplus
extern "C" {
#endif

/* FNV64 */
extern int FNV64Bstring ( const char *in,
                          uint8_t out[FNV64size] );
extern int FNV64Bblock ( const void *in,
                         long int length,
                         uint8_t out[FNV64size] );
extern int FNV64Binit ( FNV64Bcontext * const );
extern int FNV64BinitBasis ( FNV64Bcontext * const,
                             const uint8_t basis[FNV64size] );
extern int FNV64Bblockin ( FNV64Bcontext * const,
                           const void * in,
                           long int length );
extern int FNV64Bstringin ( FNV64Bcontext * const,
                            const char * in );
extern int FNV64Bresult ( FNV64Bcontext * const,
                          uint8_t out[FNV64size] );

#ifdef __cplusplus
}
#endif

#endif /* _FNV64_H_ */
```

<CODE ENDS>

<CODE BEGINS>

```
/***************************** FNV64B.c *****************************/
/******************** See RFC NNNN for details **********************/
/* Copyright (c) 2016, 2017 IETF Trust and the persons identified as
 * authors of the code.  All rights reserved.
 * See fnv-private.h for terms of use and redistribution.
 */

/* This file implements the FNV (Fowler, Noll, Vo) non-cryptographic
 * hash function FNV-1a for 64-bit hashes.
 */

#ifndef _FNV64_C_
#define _FNV64_C_

#include "fnv-private.h"
#include "FNV64.h"
```

```
/*********************************************************************
 *        START VERSION FOR WHEN YOU HAVE 64 BIT ARITHMETIC          *
 *********************************************************************/
#ifdef FNV_64bitIntegers

/* 64 bit FNV_prime = 2^40 + 2^8 + 0xb3 */
#define FNV64prime 0x00000100000001B3
#define FNV64basis 0xCBF29CE484222325

/* FNV64 hash a null terminated string  (64 bit)
 *********************************************************************/
int FNV64Bstring ( const char *in, uint64_t * const out )
{
uint64_t    temp;
uint8_t     ch;

if ( in && out )
    {
    temp = FNV64basis;
    while ( (ch = *in++) )
        temp = FNV64prime * ( temp ^ ch );
#ifdef FNV_BigEndian
    FNV64reverse ( out, temp );
#else
    *out = temp;
#endif
    return fnvSuccess;
    }
return fnvNull; /* Null input pointer */
}   /* end FNV64string */

/* FNV64 hash a counted block  (64 bit)
 *********************************************************************/
int FNV64Bblock ( const void *vin,
                  long int length,
                  uint64_t * const out )
{
const uint8_t *in = (const uint8_t*)vin;
uint64_t    temp;

if ( in && out )
    {
    if ( length < 0 )
        return fnvBadParam;
    for ( temp = FNV64basis; length > 0; length-- )
        temp = FNV64prime * ( temp ^ *in++ );
#ifdef FNV_BigEndian
    FNV64reverse ( out, temp );
#else
    *out = temp;
```

```
#endif
    return fnvSuccess;
    }
return fnvNull; /* Null input pointer */
}   /* end FNV64block */

#ifdef FNV_BigEndian

/* Store a Big Endian result back as Little Endian
 ************************************************************/
static void FNV64Breverse ( uint64_t *out, uint64_t hash )
{
uint64_t    temp;
int         i;

temp = hash & 0xFF;
for ( i = FNV64size - 1; i > 0; i-- )
    {
    hash >>= 8;
    temp = ( temp << 8 ) + ( hash & 0xFF );
    }
*out = temp;
}   /* end FNV64reverse */

#endif /* FNV_BigEndian */


/*********************************************************************
 *         Set of init, input, and output functions below            *
 *         to incrementally compute FNV64                            *
 *********************************************************************/

/* initialize context  (64 bit)
 ********************************************************************/
int FNV64Binit ( FNV64Bcontext * const ctx )
{
return FNV64initBasis ( ctx, FNV64basis );
}       /* end FNV64Binit */

/* initialize context with a provided basis  (64 bit)
 ************************************************************/
int FNV64BinitBasis ( FNV64Bcontext * const ctx, uint64_t basis )
{
if ( ctx )
    {
    ctx->Hash = basis;
    ctx->Computed = FNVinited+FNV64state;
    return fnvSuccess;
    }
return fnvNull;
```

```
}       /* end FNV64BinitBasis */

/* hash in a counted block  (64 bit)
 ********************************************************************/
int FNV64Bblockin ( FNV64Bcontext * const ctx,
                    const void *vin,
                    long int length )
{
const uint8_t *in = (const uint8_t*)vin;
uint64_t    temp;

if ( ctx && in )
    {
    if ( length < 0 )
        return fnvBadParam;
    switch ( ctx->Computed )
        {
        case FNVinited+FNV64state:
            ctx->Computed = FNVcomputed+FNV64state;
        case FNVcomputed+FNV64state:
            break;
        default:
            return fnvStateError;
        }
    for ( temp = ctx->Hash; length > 0; length-- )
        temp = FNV64prime * ( temp ^ *in++ );
    ctx->Hash = temp;
    return fnvSuccess;
    }
return fnvNull;
}   /* end FNV64Binput */

/* hash in a zero terminated string not including the zero (64 bit)
 ********************************************************************/
int FNV64Bstringin ( FNV64Bcontext * const ctx,
                     const char *in )
{
uint64_t        temp;
uint8_t         ch;

if ( ctx && in )
    {
    switch ( ctx->Computed )
        {
        case FNVinited+FNV64state:
            ctx->Computed = FNVcomputed+FNV64state;
        case FNVcomputed+FNV64state:
            break;
        default:
             return fnvStateError;
```

```
         }
    temp = ctx->Hash;
    while ( (ch = *in++) )
        temp = FNV64prime * ( temp ^ ch );
    ctx->Hash = temp;
    return fnvSuccess;
    }
return fnvNull;
}   /* end FNV64Bstringin */

/* return hash  (64 bit)
 *****************************************************************/
int FNV64Bresult ( FNV64Bcontext * const ctx,
                   uint64_t * const out )
{
if ( ctx && out )
    {
    if ( ctx->Computed != FNVcomputed+FNV64state )
        return fnvStateError;
    ctx->Computed = FNVemptied+FNV64state;
#ifdef FNV_BigEndian
    FNV64reverse ( out, ctx->Hash );
#else
    *out = ctx->Hash;
#endif
    ctx->Hash = 0;
    return fnvSuccess;
    }
return fnvNull;
}   /* end FNV64Bresult */

/*****************************************************************
 *        END VERSION FOR WHEN YOU HAVE 64 BIT ARITHMETIC        *
 *****************************************************************/
#else    /*  FNV_64bitIntegers */
/*****************************************************************
 *     START VERSION FOR WHEN YOU ONLY HAVE 32-BIT ARITHMETIC    *
 *****************************************************************/

/* 64 bit FNV_prime = 2^40 + 2^8 + 0xb3 */
/* #define FNV64prime 0x00000100000001B3 */
#define FNV64primeX 0x01B3
#define FNV64shift 8

/* #define FNV64basis 0xCBF29CE484222325 */
#define FNV64basis0 0xCBF2
#define FNV64basis1 0x9CE4
#define FNV64basis2 0x8422
#define FNV64basis3 0x2325
```

```
/* FNV64 hash a null terminated string  (32 bit)
 ***************************************************************/
int FNV64Bstring ( const char *in, uint8_t out[FNV64size] )
{
FNV64Bcontext     ctx;
int               err;

 if ( ( err = FNV64init (&ctx) ) != fnvSuccess )
    return err;
 if ( ( err = FNV64stringin (&ctx, in) ) != fnvSuccess )
    return err;
return FNV64result (&ctx, out);
}   /* end FNV64Bstring */

/* FNV64 hash a counted block  (32 bit)
 ***************************************************************/
int FNV64Bblock ( const void *in,
                  long int length,
                  uint8_t out[FNV64size] )
{
FNV64Bcontext     ctx;
int              err;

 if ( ( err = FNV64init (&ctx) ) != fnvSuccess )
    return err;
 if ( ( err = FNV64blockin (&ctx, in, length) ) != fnvSuccess )
    return err;
return FNV64result (&ctx, out);
}   /* end FNV64Bblock */


/*****************************************************************
 *         Set of init, input, and output functions below        *
 *         to incrementally compute FNV64                        *
 *****************************************************************/

/* initialize context  (32 bit)
 ***************************************************************/
int FNV64Binit ( FNV64Bcontext * const ctx )
{
if ( ctx )
    {
    ctx->Hash[0] = FNV64basis0;
    ctx->Hash[1] = FNV64basis1;
    ctx->Hash[2] = FNV64basis2;
    ctx->Hash[3] = FNV64basis3;
    ctx->Computed = FNVinited+FNV64state;
    return fnvSuccess;
    }
return fnvNull;
```

```
}   /* end FNV64Binit */

/* initialize context  (32 bit)
 ***************************************************************/
int FNV64BinitBasis ( FNV64Bcontext * const ctx,
                      const uint8_t basis[FNV64size] )
{
if ( ctx )
    {
#ifdef FNV_BigEndian
    ctx->Hash[0] = basis[1] + ( basis[0]<<8 );
    ctx->Hash[1] = basis[3] + ( basis[2]<<8 );
    ctx->Hash[2] = basis[5] + ( basis[4]<<8 );
    ctx->Hash[3] = basis[7] + ( basis[6]<<8 );
#else
    ctx->Hash[0] = basis[0] + ( basis[1]<<8 );
    ctx->Hash[1] = basis[2] + ( basis[3]<<8 );
    ctx->Hash[2] = basis[4] + ( basis[5]<<8 );
    ctx->Hash[3] = basis[6] + ( basis[7]<<8 );
#endif
    ctx->Computed = FNVinited+FNV64state;
    return fnvSuccess;
    }
return fnvNull;
}   /* end FNV64BinitBasis */

/* hash in a counted block  (32 bit)
 ***************************************************************/
int FNV64Bblockin ( FNV64Bcontext * const ctx,
                    const void *vin,
                    long int length )
{
const uint8_t *in = (const uint8_t*)vin;
uint32_t   temp[FNV64size/2];
uint32_t   temp2[2];
int        i;

if ( ctx && in )
    {
    if ( length < 0 )
        return fnvBadParam;
    switch ( ctx->Computed )
        {
        case FNVinited+FNV64state:
            ctx->Computed = FNVcomputed+FNV64state;
        case FNVcomputed+FNV64state:
            break;
        default:
            return fnvStateError;
        }
```

```
    for ( i=0; i<FNV64size/2; ++i )
         temp[i] = ctx->Hash[i];
    for ( ; length > 0; length-- )
        {
        /* temp = FNV64prime * ( temp ^ *in++ ); */
        temp2[1] = temp[3] << FNV64shift;
        temp2[0] = temp[2] << FNV64shift;
        temp[3] = FNV64primeX * ( temp[3] ^ *in++ );
        temp[2] *= FNV64primeX;
        temp[1] = temp[1] * FNV64primeX + temp2[1];
        temp[0] = temp[0] * FNV64primeX + temp2[0];
        temp[2] += temp[3] >> 16;
        temp[3] &= 0xFFFF;
        temp[1] += temp[2] >> 16;
        temp[2] &= 0xFFFF;
        temp[0] += temp[1] >> 16;
        temp[1] &= 0xFFFF;
        }
    for ( i=0; i<FNV64size/2; ++i )
        ctx->Hash[i] = temp[i];
    return fnvSuccess;
    }
return fnvNull;
}   /* end FNV64Bblockin */

/* hash in a string  (32 bit)
 ****************************************************************/
int FNV64Bstringin ( FNV64Bcontext * const ctx,
                     const char *in )
{
uint32_t   temp[FNV64size/2];
uint32_t   temp2[2];
int        i;
uint8_t    ch;

if ( ctx && in )
    {
    switch ( ctx->Computed )
        {
        case FNVinited+FNV64state:
            ctx->Computed = FNVcomputed+FNV64state;
        case FNVcomputed+FNV64state:
            break;
        default:
             return fnvStateError;
         }
    for ( i=0; i<FNV64size/2; ++i )
         temp[i] = ctx->Hash[i];
    while ( ( ch = (uint8_t)*in++ ) != 0)
        {
```

```
        /* temp = FNV64prime * ( temp ^ ch ); */
        temp2[1] = temp[3] << FNV64shift;
        temp2[0] = temp[2] << FNV64shift;
        temp[3] = FNV64primeX * ( temp[3] ^ *in++ );
        temp[2] *= FNV64primeX;
        temp[1] = temp[1] * FNV64primeX + temp2[1];
        temp[0] = temp[0] * FNV64primeX + temp2[0];
        temp[2] += temp[3] >> 16;
        temp[3] &= 0xFFFF;
        temp[1] += temp[2] >> 16;
        temp[2] &= 0xFFFF;
        temp[0] += temp[1] >> 16;
        temp[1] &= 0xFFFF;
        }
    for ( i=0; i<FNV64size/2; ++i )
        ctx->Hash[i] = temp[i];
    return fnvSuccess;
    }
return fnvNull;
}   /* end FNV64Bstringin */

/* return hash  (32 bit)
 ***********************************************************************/
int FNV64Bresult ( FNV64Bcontext * const ctx,
                   uint8_t out[FNV64size] )
{
int    i;

if ( ctx && out )
    {
    if ( ctx->Computed != FNVcomputed+FNV64state )
        return fnvStateError;
    for ( i=0; i<FNV64size/2; ++i )
        {
#ifdef FNV_BigEndian
        out[7-2*i] = ctx->Hash[i];
        out[6-2*i] = ctx->Hash[i] >> 8;
#else
        out[2*i] = ctx->Hash[i];
        out[2*i+1] = ctx->Hash[i] >> 8;
#endif
        ctx -> Hash[i] = 0;
        }
    ctx->Computed = FNVemptied+FNV64state;
    return fnvSuccess;
    }
return fnvNull;
}   /* end FNV64Bresult */

#endif    /*  FNV_64bitIntegers */
```

```
/*********************************************************************
 *        END VERSION FOR WHEN YOU ONLY HAVE 32-BIT ARITHMETIC       *
 *********************************************************************/

#endif    /* _FNV64_C_ */
```

<CODE ENDS>

6.1.3 FNV128 C Code

The header and C source for 128-bit FNV-1a returning a byte vector.

<CODE BEGINS>

```
/***************************** FNV128.h *************************/
/***************** See RFC NNNN for details. ********************/
/*
 * Copyright (c) 2016 IETF Trust and the persons identified as
 * authors of the code.  All rights reserved.
 * See fnv-private.h for terms of use and redistribution.
 */

#ifndef _FNV128_H_
#define _FNV128_H_

/*
 *  Description:
 *      This file provides headers for the 128-bit version of the
 *      FNV-1a non-cryptographic hash algorithm.
 */

#include "FNVconfig.h"

#include <stdint.h>
#define FNV128size (128/8)

/* If you do not have the ISO standard stdint.h header file, then you
 * must typedef the following types:
 *
 *    type              meaning
 *  uint64_t         unsigned 64 bit integer (ifdef FNV_64bitIntegers)
 *  uint32_t         unsigned 32 bit integer
 *  uint16_t         unsigned 16 bit integer
 *  uint8_t          unsigned 8 bit integer (i.e., unsigned char)
 */

#ifndef _FNV_ErrCodes_
#define _FNV_ErrCodes_
/*********************************************************************
 *  All FNV functions provided return as integer as follows:
```

```
 *       0 -> success
 *      >0 -> error as listed below
 */
enum {    /* success and errors */
    fnvSuccess = 0,
    fnvNull,            /* Null pointer parameter */
    fnvStateError,      /* called Input after Result or before Init */
    fnvBadParam         /* passed a bad parameter */
};
#endif /* _FNV_ErrCodes_ */

/*
 *  This structure holds context information for an FNV128 hash
 */
#ifdef FNV_64bitIntegers
    /* version if 64 bit integers supported */
typedef struct FNV128context_s {
        int Computed;  /* state */
        uint32_t Hash[FNV128size/4];
} FNV128context;

#else
    /* version if 64 bit integers NOT supported */

typedef struct FNV128context_s {
        int Computed;  /* state */
        uint16_t Hash[FNV128size/2];
} FNV128context;

#endif /* FNV_64bitIntegers */

/*
 *  Function Prototypes
 *    FNV128string: hash a zero terminated string not including
 *                  the terminating zero
 *    FNV128block: FNV128 hash a specified length byte vector
 *    FNV128init: initializes an FNV128 context
 *    FNV128initBasis: initializes an FNV128 context with a
 *                     provided basis
 *    FNV128blockin: hash in a specified length byte vector
 *    FNV128stringin: hash in a zero terminated string not
 *                    including the zero
 *    FNV128result: returns the hash value
 *
 *    Hash is returned as an array of 8-bit integers
 */

#ifdef __cplusplus
extern "C" {
#endif
```

```
/* FNV128 */
extern int FNV128string ( const char *in,
                          uint8_t out[FNV128size] );
extern int FNV128block ( const void *in,
                         long int length,
                         uint8_t out[FNV128size] );
extern int FNV128init ( FNV128context * const );
extern int FNV128initBasis ( FNV128context * const,
                             const uint8_t basis[FNV128size] );
extern int FNV128blockin ( FNV128context * const,
                           const void *in,
                           long int length );
extern int FNV128stringin ( FNV128context * const,
                            const char *in );
extern int FNV128result ( FNV128context * const,
                          uint8_t out[FNV128size] );

#ifdef __cplusplus
}
#endif

#endif /* _FNV128_H_ */
```

<CODE ENDS>

<CODE BEGINS>

```
/***************************** FNV128.c ****************************/
/******************** See RFC NNNN for details *********************/
/* Copyright (c) 2016 IETF Trust and the persons identified as
 * authors of the code.  All rights
 * See fnv-private.h for terms of use and redistribution.
 */

/* This file implements the FNV (Fowler, Noll, Vo) non-cryptographic
 * hash function FNV-1a for 128-bit hashes.
 */

#ifndef _FNV128_C_
#define _FNV128_C_

#include "fnv-private.h"
#include "FNV128.h"

/* common code for 64 and 32 bit modes */

/* FNV128 hash a null terminated string  (64/32 bit)
 ***************************************************************/
int FNV128string ( const char *in, uint8_t out[FNV128size] )
{
FNV128context    ctx;
int              err;
```

```
err = FNV128init ( &ctx );
if ( err != fnvSuccess ) return err;
err = FNV128stringin ( &ctx, in );
if ( err != fnvSuccess ) return err;
return FNV128result (&ctx, out);
}   /* end FNV128string */

/* FNV128 hash a counted block  (64/32 bit)
 ********************************************************************/
int FNV128block ( const void *in,
                  long int length,
                  uint8_t out[FNV128size] )
{
FNV128context    ctx;
int              err;

err = FNV128init ( &ctx );
if ( err != fnvSuccess ) return err;
err = FNV128blockin ( &ctx, in, length );
if ( err != fnvSuccess ) return err;
return FNV128result ( &ctx, out );
}   /* end FNV128block */


/*********************************************************************
 *        START VERSION FOR WHEN YOU HAVE 64 BIT ARITHMETIC          *
 *********************************************************************/
#ifdef FNV_64bitIntegers

/* 128 bit FNV_prime = 2^88 + 2^8 + 0x3b */
/* 0x00000000 01000000 00000000 0000013B */
#define FNV128primeX 0x013B
#define FNV128shift 24

/* 0x6C62272E 07BB0142 62B82175 6295C58D */
#define FNV128basis0 0x6C62272E
#define FNV128basis1 0x07BB0142
#define FNV128basis2 0x62B82175
#define FNV128basis3 0x6295C58D

/*********************************************************************
 *         Set of init, input, and output functions below            *
 *         to incrementally compute FNV128                           *
 *********************************************************************/

/* initialize context  (64 bit)
 ********************************************************************/
int FNV128init ( FNV128context * const ctx )
{
if ( ctx )
```

```
    {
    ctx->Hash[0] = FNV128basis0;
    ctx->Hash[1] = FNV128basis1;
    ctx->Hash[2] = FNV128basis2;
    ctx->Hash[3] = FNV128basis3;
    ctx->Computed = FNVinited+FNV128state;
    return fnvSuccess;
    }
return fnvNull;
}   /* end FNV128init */

/* initialize context with a provided basis  (64 bit)
 ****************************************************************/
int FNV128initBasis ( FNV128context * const ctx,
                      const uint8_t basis[FNV128size] )
{
int       i;
const uint8_t   *ui8p;
uint32_t  temp;

if ( ctx )
    {
#ifdef FNV_BigEndian
    ui8p = basis;
    for ( i=0; i < FNV128size/4; ++i )
        {
        temp = (*ui8p++)<<8;
        temp = (temp + *ui8p++)<<8;
            temp = (temp + *ui8p++)<<8;
        ctx->Hash[i] = temp +  *ui8p;
        }
#else
    ui8p = basis + ( FNV128size/4 - 1 );
    for ( i=0; i < FNV128size/4; ++i )
        {
            temp = (*ui8p--)<<8;
            temp = (temp + *ui8p--)<<8;
            temp = (temp + *ui8p--)<<8;
        ctx->Hash[i] = temp +  *ui8p;
        }
#endif
    ctx->Computed = FNVinited+FNV128state;
    return fnvSuccess;
    }
return fnvNull;
}   /* end FNV128initBasis */

/* hash in a counted block  (64 bit)
 ****************************************************************/
int FNV128blockin ( FNV128context * const ctx,
```

```
                    const void *vin,
                    long int length )
{
const uint8_t *in = (const uint8_t*)vin;
uint64_t    temp[FNV128size/4];
uint64_t    temp2[2];
int         i;

if ( ctx && in )
    {
    if ( length < 0 )
        return fnvBadParam;
    switch ( ctx->Computed )
        {
        case FNVinited+FNV128state:
            ctx->Computed = FNVcomputed+FNV128state;
        case FNVcomputed+FNV128state:
            break;
        default:
            return fnvStateError;
        }
    for ( i=0; i<FNV128size/4; ++i )
         temp[i] = ctx->Hash[i];
    for ( ; length > 0; length-- )
        {
        /* temp = FNV128prime * ( temp ^ *in++ ); */
        temp2[1] = temp[3] << FNV128shift;
        temp2[0] = temp[2] << FNV128shift;
        temp[3] = FNV128primeX * ( temp[3] ^ *in++ );
        temp[2] *= FNV128primeX;
        temp[1] = temp[1] * FNV128primeX + temp2[1];
        temp[0] = temp[0] * FNV128primeX + temp2[0];
        temp[2] += temp[3] >> 32;
        temp[3] &= 0xFFFFFFFF;
        temp[1] += temp[2] >> 32;
        temp[2] &= 0xFFFFFFFF;
        temp[0] += temp[1] >> 32;
        temp[1] &= 0xFFFFFFFF;
        }
    for ( i=0; i<FNV128size/4; ++i )
        ctx->Hash[i] = temp[i];
    return fnvSuccess;
    }
return fnvNull;
}   /* end FNV128input */

/* hash in a string  (64 bit)
 ***************************************************************/
int FNV128stringin ( FNV128context * const ctx, const char *in )
{
```

```
uint64_t   temp[FNV128size/4];
uint64_t   temp2[2];
int        i;
uint8_t    ch;

if ( ctx && in )
    {
    switch ( ctx->Computed )
        {
        case FNVinited+FNV128state:
            ctx->Computed = FNVcomputed+FNV128state;
        case FNVcomputed+FNV128state:
            break;
        default:
             return fnvStateError;
         }
    for ( i=0; i<FNV128size/4; ++i )
         temp[i] = ctx->Hash[i];
    while ( (ch = (uint8_t)*in++) )
        {
        /* temp = FNV128prime * ( temp ^ ch ); */
        temp2[1] = temp[3] << FNV128shift;
        temp2[0] = temp[2] << FNV128shift;
        temp[3] = FNV128primeX * ( temp[3] ^ *in++ );
        temp[2] *= FNV128primeX;
        temp[1] = temp[1] * FNV128primeX + temp2[1];
        temp[0] = temp[0] * FNV128primeX + temp2[0];
        temp[2] += temp[3] >> 32;
        temp[3] &= 0xFFFFFFFF;
        temp[1] += temp[2] >> 32;
        temp[2] &= 0xFFFFFFFF;
        temp[0] += temp[1] >> 32;
        temp[1] &= 0xFFFFFFFF;
        }
    for ( i=0; i<FNV128size/4; ++i )
        ctx->Hash[i] = temp[i];
    return fnvSuccess;
    }
return fnvNull;
}   /* end FNV128stringin */

/* return hash  (64 bit)
 ***********************************************************************/
int FNV128result ( FNV128context * const ctx, uint8_t out[FNV128size] )
{
int     i;

if ( ctx && out )
    {
    if ( ctx->Computed != FNVcomputed+FNV128state )
```

```
        return fnvStateError;
    for ( i=0; i<FNV128size/4; ++i )
        {
#ifdef FNV_BigEndian
        out[15-4*i] = ctx->Hash[i];
        out[14-4*i] = ctx->Hash[i] >> 8;
        out[13-4*i] = ctx->Hash[i] >> 16;
        out[12-4*i] = ctx->Hash[i] >> 24;
#else
        out[4*i] = ctx->Hash[i];
        out[4*i+1] = ctx->Hash[i] >> 8;
        out[4*i+2] = ctx->Hash[i] >> 16;
        out[4*i+3] = ctx->Hash[i] >> 24;
#endif
        ctx -> Hash[i] = 0;
        }
    ctx->Computed = FNVemptied+FNV128state;
    return fnvSuccess;
    }
return fnvNull;
}   /* end FNV128result */

/*****************************************************************
 *        END VERSION FOR WHEN YOU HAVE 64 BIT ARITHMETIC        *
 *****************************************************************/
#else    /*  FNV_64bitIntegers */
/*****************************************************************
 *     START VERSION FOR WHEN YOU ONLY HAVE 32-BIT ARITHMETIC    *
 *****************************************************************/

/* 128 bit FNV_prime = 2^88 + 2^8 + 0x3b */
/* 0x00000000 01000000 00000000 0000013B */
#define FNV128primeX 0x013B
#define FNV128shift 8

/* 0x6C62272E 07BB0142 62B82175 6295C58D */
uint16_t FNV128basis[FNV128size/2] =
         { 0x6C62, 0x272E, 0x07BB, 0x0142,
           0x62B8, 0x2175, 0x6295, 0xC58D };

/*****************************************************************
 *         Set of init, input, and output functions below        *
 *         to incrementally compute FNV128                       *
 *****************************************************************/

/* initialize context  (32 bit)
 *****************************************************************/
int FNV128init ( FNV128context *ctx )
{
int     i;
```

```
if ( ctx )
    {
    for ( i=0; i<FNV128size/2; ++i )
        ctx->Hash[i] = FNV128basis[i];
    ctx->Computed = FNVinited+FNV128state;
    return fnvSuccess;
    }
return fnvNull;
}   /* end FNV128init */

/* initialize context with a provided basis  (32 bit)
 ****************************************************************/
int FNV128initBasis ( FNV128context *ctx,
                      const uint8_t basis[FNV128size] )
{
int      i;
const uint8_t  *ui8p;
uint32_t temp;

if ( ctx )
    {
#ifdef FNV_BigEndian
    ui8p = basis;
    for ( i=0; i < FNV128size/2; ++i )
        {
        temp = *ui8p++;
        ctx->Hash[i] = ( temp<<8 ) + (*ui8p++);
        }
#else
    ui8p = basis + (FNV128size/2 - 1);
    for ( i=0; i < FNV128size/2; ++i )
        {
        temp = *ui8p--;
        ctx->Hash[i] = ( temp<<8 ) +  (*ui8p--);
        }
#endif
    ctx->Computed = FNVinited+FNV128state;
    return fnvSuccess;
    }
return fnvNull;
}   /* end FNV128initBasis */

/* hash in a counted block  (32 bit)
 ****************************************************************/
int FNV128blockin ( FNV128context *ctx,
                    const void *vin,
                    long int length )
{
const uint8_t *in = (const uint8_t*)vin;
uint32_t   temp[FNV128size/2];
```

```
uint32_t   temp2[3];
int        i;

if ( ctx && in )
    {
    if ( length < 0 )
        return fnvBadParam;
    switch ( ctx->Computed )
        {
        case FNVinited+FNV128state:
            ctx->Computed = FNVcomputed+FNV128state;
        case FNVcomputed+FNV128state:
            break;
        default:
            return fnvStateError;
        }
    for ( i=0; i<FNV128size/2; ++i )
         temp[i] = ctx->Hash[i];
    for ( ; length > 0; length-- )
        {
        /* temp = FNV128prime * ( temp ^ *in++ ); */
        temp[7] ^= *in++;
        temp2[2] = temp[7] << FNV128shift;
        temp2[1] = temp[6] << FNV128shift;
        temp2[0] = temp[5] << FNV128shift;
        for ( i=0; i<8; ++i )
            temp[i] *= FNV128primeX;
        temp[2] += temp2[2];
        temp[1] += temp2[1];
        temp[0] += temp2[0];
        for ( i=7; i>0; --i )
            {
            temp[i-1] += temp[i] >> 16;
            temp[i] &= 0xFFFF;
            }
        }
    for ( i=0; i<FNV128size/2; ++i )
        ctx->Hash[i] = temp[i];
    return fnvSuccess;
    }
return fnvNull;
}   /* end FNV128blockin */

/* hash in a string  (32 bit)
 ****************************************************************/
int FNV128stringin ( FNV128context *ctx,
                     const char *in )
{
uint32_t   temp[FNV128size/2];
uint32_t   temp2[3];
```

```
int        i;
uint8_t    ch;

if ( ctx && in )
    {
    switch ( ctx->Computed )
        {
        case FNVinited+FNV128state:
            ctx->Computed = FNVcomputed+FNV128state;
        case FNVcomputed+FNV128state:
            break;
        default:
             return fnvStateError;
         }
    for ( i=0; i<FNV128size/2; ++i )
         temp[i] = ctx->Hash[i];
    while ( (ch = (uint8_t)*in++) )
        {
        /* temp = FNV128prime * ( temp ^ *in++ ); */
        temp[7] ^= ch;
        temp2[2] = temp[7] << FNV128shift;
        temp2[1] = temp[6] << FNV128shift;
        temp2[0] = temp[5] << FNV128shift;
        for ( i=0; i<8; ++i )
            temp[i] *= FNV128primeX;
        temp[2] += temp2[2];
        temp[1] += temp2[1];
        temp[0] += temp2[0];
        for ( i=7; i>0; --i )
            {
            temp[i-1] += temp[i] >> 16;
            temp[i] &= 0xFFFF;
            }
         }
    for ( i=0; i<FNV128size/2; ++i )
        ctx->Hash[i] = temp[i];
    return fnvSuccess;
    }
return fnvNull;
}   /* end FNV128stringin */

/* return hash  (32 bit)
 ************************************************************************/
int FNV128result ( FNV128context *ctx,
                   uint8_t out[FNV128size] )
{
int    i;

if ( ctx && out )
    {
```

```
    if ( ctx->Computed != FNVcomputed+FNV128state )
        return fnvStateError;
    for ( i=0; i<FNV128size/2; ++i )
        {
#ifdef FNV_BigEndian
        out[15-2*i] = ctx->Hash[i];
        out[14-2*i] = ctx->Hash[i] >> 8;
#else
        out[2*i] = ctx->Hash[i];
        out[2*i+1] = ctx->Hash[i] >> 8;
#endif
        ctx -> Hash[i] = 0;
        }
    ctx->Computed = FNVemptied+FNV128state;
    return fnvSuccess;
    }
return fnvNull;
}   /* end FNV128result */

#endif    /*  Have64bitIntegers */
/*****************************************************************
 *        END VERSION FOR WHEN YOU ONLY HAVE 32-BIT ARITHMETIC      *
 *****************************************************************/

#endif    /* _FNV128_C_ */
```

<CODE ENDS>

6.1.4 FNV256 C Code

The header and C source for 256-bit FNV-1a returning a byte vector.

<CODE BEGINS>

```
/*************************** FNV256.h ************************/
/***************** See RFC NNNN for details. ******************/
/*
 * Copyright (c) 2016 IETF Trust and the persons identified as
 * authors of the code.  All rights reserved.
 * See fnv-private.h for terms of use and redistribution.
 */

#ifndef _FNV256_H_
#define _FNV256_H_

/*
 *  Description:
 *      This file provides headers for the 256-bit version of the
 *      FNV-1a non-cryptographic hash algorithm.
 */
```

```
#include "FNVconfig.h"

#include <stdint.h>
#define FNV256size (256/8)

/* If you do not have the ISO standard stdint.h header file, then you
 * must typedef the following types:
 *
 *    type              meaning
 *  uint64_t         unsigned 64 bit integer (ifdef FNV_64bitIntegers)
 *  uint32_t         unsigned 32 bit integer
 *  uint16_t         unsigned 16 bit integer
 *  uint8_t          unsigned 8 bit integer (i.e., unsigned char)
 */

#ifndef _FNV_ErrCodes_
#define _FNV_ErrCodes_
/*****************************************************************
 *  All FNV functions provided return as integer as follows:
 *       0 -> success
 *      >0 -> error as listed below
 */
enum {    /* success and errors */
    fnvSuccess = 0,
    fnvNull,            /* Null pointer parameter */
    fnvStateError,      /* called Input after Result or before Init */
    fnvBadParam         /* passed a bad parameter */
};
#endif /* _FNV_ErrCodes_ */

/*
 *  This structure holds context information for an FNV256 hash
 */
#ifdef FNV_64bitIntegers
    /* version if 64 bit integers supported */
typedef struct FNV256context_s {
        int Computed;  /* state */
        uint32_t Hash[FNV256size/4];
} FNV256context;

#else
    /* version if 64 bit integers NOT supported */

typedef struct FNV256context_s {
        int Computed;  /* state */
        uint16_t Hash[FNV256size/2];
} FNV256context;

#endif /* FNV_64bitIntegers */
```

```
/*
 *  Function Prototypes
 *    FNV256string: hash a zero terminated string not including
 *                  the zero
 *    FNV256block: FNV256 hash a specified length byte vector
 *    FNV256init: initializes an FNV256 context
 *    FNV256initBasis:  initializes an FNV256 context with a provided
 *                      basis
 *    FNV256blockin: hash in a specified length byte vector
 *    FNV256stringin: hash in a zero terminated string not
 *                    including the zero
 *    FNV256result: returns the hash value
 *
 *    Hash is returned as an array of 8-bit integers
 */

#ifdef __cplusplus
extern "C" {
#endif

/* FNV256 */
extern int FNV256string ( const char *in,
                          uint8_t out[FNV256size] );
extern int FNV256block ( const void *in,
                         long int length,
                         uint8_t out[FNV256size] );
extern int FNV256init ( FNV256context *);
extern int FNV256initBasis ( FNV256context * const,
                             const uint8_t basis[FNV256size] );
extern int FNV256blockin ( FNV256context * const,
                           const void *in,
                           long int length );
extern int FNV256stringin ( FNV256context * const,
                            const char *in );
extern int FNV256result ( FNV256context * const,
                          uint8_t out[FNV256size] );

#ifdef __cplusplus
}
#endif

#endif /* _FNV256_H_ */
```

<CODE ENDS>

<CODE BEGINS>

```
/***************************** FNV256.c ****************************/
/******************** See RFC NNNN for details *********************/
/* Copyright (c) 2016 IETF Trust and the persons identified as
 * authors of the code.  All rights
 * See fnv-private.h for terms of use and redistribution.
```

```
 */

/* This file implements the FNV (Fowler, Noll, Vo) non-cryptographic
 * hash function FNV-1a for 256-bit hashes.
 */

#ifndef _FNV256_C_
#define _FNV256_C_

#include "fnv-private.h"
#include "FNV256.h"

/* common code for 64 and 32 bit modes */

/* FNV256 hash a null terminated string  (64/32 bit)
 ***************************************************************/
int FNV256string ( const char *in, uint8_t out[FNV256size] )
{
FNV256context    ctx;
int              err;

if ( (err = FNV256init ( &ctx )) != fnvSuccess )
    return err;
if ( (err = FNV256stringin ( &ctx, in )) != fnvSuccess )
    return err;
return FNV256result ( &ctx, out );
}   /* end FNV256string */

/* FNV256 hash a counted block  (64/32 bit)
 ***************************************************************/
int FNV256block ( const void *in,
                  long int length,
                  uint8_t out[FNV256size] )
{
FNV256context    ctx;
int              err;

if ( (err = FNV256init ( &ctx )) != fnvSuccess )
    return err;
if ( (err = FNV256blockin ( &ctx, in, length)) != fnvSuccess )
    return err;
return FNV256result ( &ctx, out );
}   /* end FNV256block */


/***************************************************************
 *        START VERSION FOR WHEN YOU HAVE 64 BIT ARITHMETIC         *
 ***************************************************************/
#ifdef FNV_64bitIntegers
```

```
/* 256 bit FNV_prime = 2^168 + 2^8 + 0x63 */
/* 0x0000000000000000 0000010000000000
     0000000000000000 0000000000000163 */
#define FNV256primeX 0x0163
#define FNV256shift 8

/* 0xDD268DBCAAC55036 2D98C384C4E576CC
     C8B1536847B6BBB3 1023B4C8CAEE0535 */
uint32_t FNV256basis[FNV256size/4] = {
         0xDD268DBC, 0xAAC55036, 0x2D98C384, 0xC4E576CC,
         0xC8B15368, 0x47B6BBB3, 0x1023B4C8, 0xCAEE0535 };

/*************************************************************
 *         Set of init, input, and output functions below    *
 *         to incrementally compute FNV256                   *
 *************************************************************/

/* initialize context  (64 bit)
 *************************************************************/
int FNV256init ( FNV256context *ctx )
{
int      i;

if ( ctx )
    {
    for ( i=0; i<FNV256size/4; ++i )
        ctx->Hash[i] = FNV256basis[i];
    ctx->Computed = FNVinited+FNV256state;
    return fnvSuccess;
    }
return fnvNull;
}   /* end FNV256init */

/* initialize context with a provided basis  (64 bit)
 *************************************************************/
int FNV256initBasis ( FNV256context* const ctx,
                      const uint8_t basis[FNV256size] )
{
int      i;
const uint8_t  *ui8p;
uint32_t  temp;

if ( ctx )
    {
#ifdef FNV_BigEndian
    ui8p = basis;
    for ( i=0; i < FNV256size/4; ++i )
        {
            temp = (*ui8p++)<<8;
            temp = (temp + *ui8p++)<<8;
```

```
            temp = (temp + *ui8p++)<<8;
        ctx->Hash[i] = temp +  *ui8p;
        }
#else
    ui8p = basis + (FNV256size/4 - 1);
    for ( i=0; i < FNV256size/4; ++i )
        {
            temp = (*ui8p--)<<8;
            temp = (temp + *ui8p--)<<8;
            temp = (temp + *ui8p--)<<8;
        ctx->Hash[i] = temp +  *ui8p;
        }
#endif
    ctx->Computed = FNVinited+FNV256state;
    return fnvSuccess;
    }
return fnvNull;
}   /* end FNV256initBasis */

/* hash in a counted block  (64 bit)
 ********************************************************************/
int FNV256blockin ( FNV256context *ctx,
                    const void *vin,
                    long int length )
{
const uint8_t *in = (const uint8_t*)vin;
uint64_t    temp[FNV256size/4];
uint64_t    temp2[3];
int         i;

if ( ctx && in )
    {
    if ( length < 0 )
        return fnvBadParam;
    switch ( ctx->Computed )
        {
        case FNVinited+FNV256state:
            ctx->Computed = FNVcomputed+FNV256state;
        case FNVcomputed+FNV256state:
            break;
        default:
            return fnvStateError;
        }
    for ( i=0; i<FNV256size/4; ++i )
         temp[i] = ctx->Hash[i];
    for ( ; length > 0; length-- )
        {
        /* temp = FNV256prime * ( temp ^ *in++ ); */
        temp[7] ^= *in++;
        temp2[2] = temp[7] << FNV256shift;
```

```
        temp2[1] = temp[6] << FNV256shift;
        temp2[0] = temp[5] << FNV256shift;
        for ( i=0; i<FNV256size/4; ++i )
            temp[i] *= FNV256primeX;
        temp[2] += temp2[2];
        temp[1] += temp2[1];
        temp[0] += temp2[0];
        for ( i=FNV256size/4-1; i>0; --i )
            {
            temp[i-1] += temp[i] >> 16;
            temp[i] &= 0xFFFF;
            }
       }
    for ( i=0; i<FNV256size/4; ++i )
        ctx->Hash[i] = temp[i];
    return fnvSuccess;
    }
return fnvNull;
}   /* end FNV256input */

/* hash in a string  (64 bit)
 ****************************************************************/
int FNV256stringin ( FNV256context *ctx, const char *in )
{
uint64_t   temp[FNV256size/4];
uint64_t   temp2[3];
int        i;
uint8_t    ch;

if ( ctx && in )
    {
    switch ( ctx->Computed )
        {
        case FNVinited+FNV256state:
            ctx->Computed = FNVcomputed+FNV256state;
        case FNVcomputed+FNV256state:
            break;
        default:
             return fnvStateError;
         }
    for ( i=0; i<FNV256size/4; ++i )
         temp[i] = ctx->Hash[i];
    while ( (ch = (uint8_t)*in++) )
        {
        /* temp = FNV256prime * ( temp ^ ch ); */
        temp[7] ^= ch;
        temp2[2] = temp[7] << FNV256shift;
        temp2[1] = temp[6] << FNV256shift;
        temp2[0] = temp[5] << FNV256shift;
        for ( i=0; i<FNV256size/4; ++i )
```

```
            temp[i] *= FNV256primeX;
        temp[2] += temp2[2];
        temp[1] += temp2[1];
        temp[0] += temp2[0];
        for ( i=FNV256size/4-1; i>0; --i )
            {
            temp[i-1] += temp[i] >> 16;
            temp[i] &= 0xFFFF;
            }
       }
    for ( i=0; i<FNV256size/4; ++i )
        ctx->Hash[i] = temp[i];
    return fnvSuccess;
    }
return fnvNull;
}   /* end FNV256stringin */

/* return hash  (64 bit)
 ************************************************************************/
int FNV256result ( FNV256context *ctx, uint8_t out[FNV256size] )
{
int     i;

if ( ctx && out )
    {
    if ( ctx->Computed != FNVcomputed+FNV256state )
        return fnvStateError;
    for ( i=0; i<FNV256size/4; ++i )
        {
#ifdef FNV_BigEndian
        out[31-4*i] = ctx->Hash[i];
        out[31-4*i] = ctx->Hash[i] >> 8;
        out[31-4*i] = ctx->Hash[i] >> 16;
        out[31-4*i] = ctx->Hash[i] >> 24;
#else
        out[4*i] = ctx->Hash[i];
        out[4*i+1] = ctx->Hash[i] >> 8;
        out[4*i+2] = ctx->Hash[i] >> 16;
        out[4*i+3] = ctx->Hash[i] >> 24;
#endif
        ctx -> Hash[i] = 0;
        }
    ctx->Computed = FNVemptied+FNV256state;
    return fnvSuccess;
    }
return fnvNull;
}   /* end FNV256result */

/*****************************************************************
 *        END VERSION FOR WHEN YOU HAVE 64 BIT ARITHMETIC         *
```

```
 ************************************************************************/
#else    /*  FNV_64bitIntegers */
/************************************************************************
 *      START VERSION FOR WHEN YOU ONLY HAVE 32-BIT ARITHMETIC    *
 ************************************************************************/

/* version for when you only have 32-bit arithmetic
 ************************************************************************/

/* 256 bit FNV_prime = 2^168 + 2^8 + 0x63 */
/* 0x00000000 00000000 00000100 00000000
     00000000 00000000 00000000 00000163 */
#define FNV256primeX 0x0163
#define FNV256shift 8

/* 0xDD268DBCAAC55036 2D98C384C4E576CC
     C8B1536847B6BBB3 1023B4C8CAEE0535 */
uint16_t FNV256basis[FNV256size/2] = {
         0xDD26, 0x8DBC, 0xAAC5, 0x5036,
         0x2D98, 0xC384, 0xC4E5, 0x76CC,
         0xC8B1, 0x5368, 0x47B6, 0xBBB3,
         0x1023, 0xB4C8, 0xCAEE, 0x0535 };

/************************************************************************
 *         Set of init, input, and output functions below           *
 *         to incrementally compute FNV256                          *
 ************************************************************************/

/* initialize context  (32 bit)
 ************************************************************************/
int FNV256init ( FNV256context *ctx )
{
int     i;

if ( ctx )
    {
    for ( i=0; i<FNV256size/2; ++i )
        ctx->Hash[i] = FNV256basis[i];
    ctx->Computed = FNVinited+FNV256state;
    return fnvSuccess;
    }
return fnvNull;
}   /* end FNV256init */

/* initialize context with a provided basis  (32 bit)
 ************************************************************************/
int FNV256initBasis ( FNV256context *ctx,
                      const uint8_t basis[FNV256size] )
{
int      i;
```

```
const uint8_t  *ui8p;
uint32_t temp;

if ( ctx )
    {
#ifdef FNV_BigEndian
    ui8p = basis;
    for ( i=0; i < FNV256size/2; ++i )
        {
        temp = *ui8p++;
        ctx->Hash[i] = ( temp<<8 ) + (*ui8p++);
        }
#else
    ui8p = basis + FNV256size/2 -1;
    for ( i=0; i < FNV256size/2; ++i )
        {
        temp = *ui8p--;
        ctx->Hash[i] = ( temp<<8 ) +  (*ui8p--);
        }
#endif
    ctx->Computed = FNVinited+FNV256state;
    return fnvSuccess;
    }
return fnvNull;
}   /* end FNV256initBasis */

/* hash in a counted block  (32 bit)
 ****************************************************************/
int FNV256blockin ( FNV256context *ctx,
                    const void *vin,
                    long int length )
{
const uint8_t *in = (const uint8_t*)vin;
uint32_t   temp[FNV256size/2];
uint32_t   temp2[6];
int        i;

if ( ctx && in )
    {
    if ( length < 0 )
        return fnvBadParam;
    switch ( ctx->Computed )
        {
        case FNVinited+FNV256state:
            ctx->Computed = FNVcomputed+FNV256state;
        case FNVcomputed+FNV256state:
            break;
        default:
            return fnvStateError;
        }
```

```
    for ( i=0; i<FNV256size/2; ++i )
        temp[i] = ctx->Hash[i];
    for ( ; length > 0; length-- )
        {
        /* temp = FNV256prime * ( temp ^ *in++ ); */
        temp[15] ^= *in++;
        for ( i=0; i<6; ++i )
            temp2[i] = temp[10+i] << FNV256shift;
        for ( i=0; i<FNV256size/2; ++i )
            temp[i] *= FNV256primeX;
        for ( i=0; i<6; ++i )
            temp[10+i] += temp2[i];
        for ( i=15; i>0; --i )
            {
            temp[i-1] += temp[i] >> 16;
            temp[i] &= 0xFFFF;
            }
        }
    for ( i=0; i<FNV256size/2; ++i )
        ctx->Hash[i] = temp[i];
    return fnvSuccess;
    }
return fnvNull;
}   /* end FNV256blockin */

/* hash in a string  (32 bit)
 ***************************************************************/
int FNV256stringin ( FNV256context *ctx, const char *in )
{
uint32_t   temp[FNV256size/2];
uint32_t   temp2[6];
int        i;
uint8_t    ch;

if ( ctx && in )
    {
    switch ( ctx->Computed )
        {
        case FNVinited+FNV256state:
            ctx->Computed = FNVcomputed+FNV256state;
        case FNVcomputed+FNV256state:
            break;
        default:
             return fnvStateError;
         }
    for ( i=0; i<FNV256size/2; ++i )
         temp[i] = ctx->Hash[i];
    while ( ( ch = (uint8_t)*in++ ) != 0)
        {
        /* temp = FNV256prime * ( temp ^ *in++ ); */
```

```
        temp[15] ^= ch;
        for ( i=0; i<6; ++i )
            temp2[i] = temp[10+i] << FNV256shift;
        for ( i=0; i<FNV256size/2; ++i )
            temp[i] *= FNV256primeX;
        for ( i=0; i<6; ++i )
            temp[10+i] += temp2[i];
        for ( i=15; i>0; --i )
            {
            temp[i-1] += temp[i] >> 16;
            temp[i] &= 0xFFFF;
            }
         }
    for ( i=0; i<FNV256size/2; ++i )
        ctx->Hash[i] = temp[i];
    return fnvSuccess;
    }
return fnvNull;
}   /* end FNV256stringin */

/* return hash  (32 bit)
 ********************************************************************/
int FNV256result ( FNV256context *ctx, uint8_t out[FNV256size] )
{
int    i;

if ( ctx && out )
    {
    if ( ctx->Computed != FNVcomputed+FNV256state )
        return fnvStateError;
    for ( i=0; i<FNV256size/2; ++i )
        {
#ifdef FNV_BigEndian
        out[31-2*i] = ctx->Hash[i];
        out[30-2*i] = ctx->Hash[i] >> 8;
#else
        out[2*i] = ctx->Hash[i];
        out[2*i+1] = ctx->Hash[i] >> 8;
#endif
        ctx->Hash[i] = 0;
        }
    ctx->Computed = FNVemptied+FNV256state;
    return fnvSuccess;
    }
return fnvNull;
}   /* end FNV256result */

#endif    /*  Have64bitIntegers */
/*****************************************************************
 *        END VERSION FOR WHEN YOU ONLY HAVE 32-BIT ARITHMETIC      *
```

```
 ****************************************************************/

#endif    /* _FNV256_C_ */
   <CODE ENDS>
```

6.1.5 FNV512 C Code

The header and C source for 512-bit FNV-1a returning a byte vector.

```
   <CODE BEGINS>
/***************************** FNV512.h *************************/
/***************** See RFC NNNN for details. ********************/
/*
 * Copyright (c) 2016 IETF Trust and the persons identified as
 * authors of the code.  All rights reserved.
 * See fnv-private.h for terms of use and redistribution.
 */

#ifndef _FNV512_H_
#define _FNV512_H_

/*
 *  Description:
 *      This file provides headers for the 512-bit version of the
 *      FNV-1a non-cryptographic hash algorithm.
 */

#include "FNVconfig.h"

#include <stdint.h>
#define FNV512size (512/8)

/* If you do not have the ISO standard stdint.h header file, then you
 * must typedef the following types:
 *
 *    type              meaning
 *  uint64_t         unsigned 64 bit integer (ifdef FNV_64bitIntegers)
 *  uint32_t         unsigned 32 bit integer
 *  uint16_t         unsigned 16 bit integer
 *  uint8_t          unsigned 8 bit integer (i.e., unsigned char)
 */

#ifndef _FNV_ErrCodes_
#define _FNV_ErrCodes_
/*****************************************************************
 *  All FNV functions provided return as integer as follows:
 *       0 -> success
 *      >0 -> error as listed below
```

```
 */
enum {    /* success and errors */
    fnvSuccess = 0,
    fnvNull,            /* Null pointer parameter */
    fnvStateError,      /* called Input after Result or before Init */
    fnvBadParam         /* passed a bad parameter */
};
#endif /* _FNV_ErrCodes_ */

/*
 *  This structure holds context information for an FNV512 hash
 */
#ifdef FNV_64bitIntegers
    /* version if 64 bit integers supported */
typedef struct FNV512context_s {
        int Computed;  /* state */
        uint32_t Hash[FNV512size/4];
} FNV512context;

#else
    /* version if 64 bit integers NOT supported */

typedef struct FNV512context_s {
        int Computed;  /* state */
        uint16_t Hash[FNV512size/2];
} FNV512context;

#endif /* FNV_64bitIntegers */

/*
 *  Function Prototypes
 *    FNV512string: hash a zero terminated string not including
 *                  the terminating zero
 *    FNV512block: FNV512 hash a specified length byte vector
 *    FNV512init: initializes an FNV512 context
 *    FNV512initBasis: initializes an FNV512 context with a
 *                     provided basis
 *    FNV512blockin: hash in a specified length byte vector
 *    FNV512stringin: hash in a zero terminated string not
 *                    including the zero
 *    FNV512result: returns the hash value
 *
 *    Hash is returned as an array of 8-bit integers
 */

#ifdef __cplusplus
extern "C" {
#endif

/* FNV512 */
```

```
extern int FNV512string ( const char *in,
                          uint8_t out[FNV512size] );
extern int FNV512block ( const void *in,
                         long int length,
                         uint8_t out[FNV512size] );
extern int FNV512init ( FNV512context *);
extern int FNV512initBasis ( FNV512context * const,
                             const uint8_t basis[FNV512size] );
extern int FNV512blockin ( FNV512context *,
                           const void *in,
                           long int length );
extern int FNV512stringin ( FNV512context *,
                            const char *in );
extern int FNV512result ( FNV512context *,
                          uint8_t out[FNV512size] );

#ifdef __cplusplus
}
#endif

#endif /* _FNV512_H_ */
```

<CODE ENDS>

<CODE BEGINS>

```
/***************************** FNV512.c ****************************/
/******************** See RFC NNNN for details *********************/
/* Copyright (c) 2016, 2017 IETF Trust and the persons identified as
 * authors of the code.  All rights
 * See fnv-private.h for terms of use and redistribution.
 */

/* This file implements the FNV (Fowler, Noll, Vo) non-cryptographic
 * hash function FNV-1a for 512-bit hashes.
 */

#ifndef _FNV512_C_
#define _FNV512_C_

#include "fnv-private.h"
#include "FNV512.h"

/* common code for 64 and 32 bit modes */

/* FNV512 hash a null terminated string  (64/32 bit)
 ***************************************************************/
int FNV512string ( const char *in, uint8_t out[FNV512size] )
{
FNV512context    ctx;
int              err;
```

```
if ( (err = FNV512init ( &ctx )) != fnvSuccess )
    return err;
if ( (err = FNV512stringin ( &ctx, in )) != fnvSuccess )
    return err;
return FNV512result ( &ctx, out );
}   /* end FNV512string */

/* FNV512 hash a counted block  (64/32 bit)
 ****************************************************************/
int FNV512block ( const void *in,
                  long int length,
                  uint8_t out[FNV512size] )
{
FNV512context    ctx;
int              err;

if ( (err = FNV512init ( &ctx )) != fnvSuccess )
    return err;
if ( (err = FNV512blockin ( &ctx, in, length)) != fnvSuccess )
    return err;
return FNV512result ( &ctx, out );
}   /* end FNV512block */


/*****************************************************************
 *        START VERSION FOR WHEN YOU HAVE 64 BIT ARITHMETIC         *
 *****************************************************************/
#ifdef FNV_64bitIntegers

/*
  512 bit FNV_prime = 2^344 + 2^8 + 0x57 =
   0x0000000000000000 0000000000000000
     0000000001000000 0000000000000000
     0000000000000000 0000000000000000
     0000000000000000 0000000000000157 */
#define FNV512primeX 0x0157
#define FNV512shift 24

/* 0xB86DB0B1171F4416 DCA1E50F309990AC
     AC87D059C9000000 0000000000000D21
     E948F68A34C192F6 2EA79BC942DBE7CE
     182036415F56E34B AC982AAC4AFE9FD9 */

uint32_t FNV512basis[FNV512size/4] = {
         0xB86DB0B1, 0x171F4416, 0xDCA1E50F, 0x209990AC,
         0xAC87D059, 0x9C000000, 0x00000000, 0x00000D21,
         0xE948F68A, 0x34C192F6, 0x2EA79BC9, 0x42DBE7CE,
         0x18203641, 0x5F56E34B, 0xAC982AAC, 0x4AFE9FD9 };

/*****************************************************************
```

```
 *         Set of init, input, and output functions below           *
 *         to incrementally compute FNV512                          *
 ************************************************************************/

/* initialize context  (64 bit)
 ************************************************************************/
int FNV512init ( FNV512context *ctx )
{
if ( ctx )
    {
    for ( i=0; i<FNV512size/4; ++i )
        ctx->Hash[i] = FNV512basis[i];
    ctx->Computed = FNVinited+FNV512state;
    return fnvSuccess;
    }
return fnvNull;
}   /* end FNV512init */

/* initialize context with a provided basis  (64 bit)
 ************************************************************************/
int FNV512initBasis ( FNV512context* const ctx,
                      const uint8_t basis[FNV512size] )
{
int      i;
const uint8_t  *ui8p;
uint32_t  temp;

if ( ctx )
    {
#ifdef FNV_BigEndian
    ui8p = basis;
    for ( i=0; i < FNV512size/4; ++i )
        {
            temp = (*ui8p++)<<8;
            temp = (temp + *ui8p++)<<8;
            temp = (temp + *ui8p++)<<8;
        ctx->Hash[i] = temp +  *ui8p;
        }
#else
    ui8p = basis + (FNV512size/4 - 1);
    for ( i=0; i < FNV512size/4; ++i )
        {
            temp = (*ui8p--)<<8;
            temp = (temp + *ui8p--)<<8;
            temp = (temp + *ui8p--)<<8;
        ctx->Hash[i] = temp +  *ui8p;
        }
#endif
    ctx->Computed = FNVinited+FNV512state;
    return fnvSuccess;
```

```
    }
return fnvNull;
}   /* end FNV512initBasis */

/* hash in a counted block  (64 bit)
 ********************************************************************/
int FNV512blockin ( FNV512context *ctx,
                    const void *vin,
                    long int length )
{
const uint8_t *in = (const uint8_t*)vin;
uint64_t    temp[FNV512size/4];
uint64_t    temp2[3];

if ( ctx && in )
    {
    switch ( ctx->Computed )
        {
        case FNVinited+FNV512state:
            ctx->Computed = FNVcomputed+FNV128state;
        case FNVcomputed+FNV512state:
            break;
        default:
            return fnvStateError;
        }
    if ( length < 0 )
        return fnvBadParam;
    for ( i=0; i<FNV512size/4; ++i )
         temp[i] = ctx->Hash[i];
    for ( ; length > 0; length-- )
        {
        /* temp = FNV512prime * ( temp ^ *in++ ); */
        temp[7] ^= *in++;
        temp2[2] = temp[7] << FNV512shift;
        temp2[1] = temp[6] << FNV512shift;
        temp2[0] = temp[5] << FNV512shift;
        for ( i=0; i<FNV512size/4; ++i )
            temp[i] *= FNV512primeX;
        temp[2] += temp2[2];
        temp[1] += temp2[1];
        temp[0] += temp2[0];
        for ( i=FNV512size/4-1; i>0; --i )
            {
            temp[i-1] += temp[i] >> 16;
            temp[i] &= 0xFFFF;
            }
       }
    for ( i=0; i<FNV512size/4; ++i )
        ctx->Hash[i] = temp[i];
    return fnvSuccess;
```

```
    }
return fnvNull;
}   /* end FNV512input */

/* hash in a string  (64 bit)
 ****************************************************************/
inf FNV512stringin ( FNV512context *ctx, const char *in )
{
uint64_t   temp[FNV512size/4];
uint64_t   temp2[2];
int        i;
uint8_t    ch;

if ( ctx && in )
    {
    switch ( ctx->Computed )
        {
        case FNVinited+FNV512state:
            ctx->Computed = FNVcomputed+FNV512state;
        case FNVcomputed+FNV512state:
            break;
        default:
             return fnvStateError;
         }
    for ( i=0; i<FNV512size/4; ++i )
         temp[i] = ctx->Hash[i];
    while ( ch = (uint8_t)*in++ )
        {
        /* temp = FNV512prime * ( temp ^ ch ); */
        temp[7] ^= ch;
        temp2[2] = temp[7] << FNV128shift;
        temp2[1] = temp[6] << FNV128shift;
        temp2[0] = temp[5] << FNV128shift;
        for ( i=0; i<FNV512size/4; ++i )
            temp[i] *= FNV512prime;
        temp[2] += temp2[2];
        temp[1] += temp2[1];
        temp[0] += temp2[0];
        for ( i=FNVsize512/4-1; i>0; --i )
            {
            temp[i-1] += temp[i] >> 16;
            temp[i] &= 0xFFFF;
            }
       }
    for ( i=0; i<FNV512size/4; ++i )
        ctx->Hash[i] = temp[i];
    return fnvSuccess;
    }
return fnvNull;
}   /* end FNV512stringin */
```

```
/* return hash  (64 bit)
 ************************************************************************/
int FNV512result ( FNV512context *ctx, uint8_t out[FNV512size] )
{
if ( ctx && out )
    {
    if ( ctx->Computed != FNVcomputed+FNV512state )
        return fnvStateError;
    for ( i=0; i<FNV512size/4; ++i )
        {
#ifdef FNV_BigEndian
        out[15-2*i] = ctx->Hash[i];
        out[14-2*i] = ctx->Hash[i] >> 8;
#else
        out[2*i] = ctx->Hash[i];
        out[2*i+1] = ctx->Hash[i] >> 8;
#endif
        ctx -> Hash[i] = 0;
        }
    ctx->Computed = FNVemptied+FNV512state;
    return fnvSuccess;
    }
return fnvNull;
}   /* end FNV512result */

/*****************************************************************
 *        END VERSION FOR WHEN YOU HAVE 64 BIT ARITHMETIC         *
 *****************************************************************/
#else    /*  FNV_64bitIntegers */
/*****************************************************************
 *      START VERSION FOR WHEN YOU ONLY HAVE 32-BIT ARITHMETIC    *
 *****************************************************************/

/* version for when you only have 32-bit arithmetic
 ************************************************************************/

/*
  512 bit FNV_prime = 2^344 + 2^8 + 0x57 =
   0x0000000000000000 0000000000000000
     0000000001000000 0000000000000000
     0000000000000000 0000000000000000
     0000000000000000 0000000000000157 */
#define FNV512primeX 0x0157
#define FNV512shift 8

/* 0xB86DB0B1171F4416 DCA1E50F309990AC
     AC87D059C9000000 0000000000000D21
     E948F68A34C192F6 2EA79BC942DBE7CE
     182036415F56E34B AC982AAC4AFE9FD9 */
```

```
uint16_t FNV512basis[FNV512size/2] = {
     0xB86D, 0xB0B1, 0x171F, 0x4416, 0xDCA1, 0xE50F, 0x3099, 0x90AC,
     0xAC87, 0xD059, 0xC900, 0x0000, 0x0000, 0x0000, 0x0000, 0x0D21,
     0xE948, 0xF68A, 0x34C1, 0x92F6, 0x2EA7, 0x9BC9, 0x42DB, 0xE7CE,
     0x1820, 0x3641, 0x5F56, 0xE34B, 0xAC98, 0x2AAC, 0x4AFE, 0x9FD9 };


/*************************************************************
 *         Set of init, input, and output functions below    *
 *         to incrementally compute FNV512                   *
 *************************************************************/

/* initialize context  (32 bit)
 *************************************************************/
int FNV512init ( FNV512context *ctx )
{
int     i;

if ( ctx )
    {
    for ( i=0; i<FNV512size/2; ++i )
        ctx->Hash[i] = FNV512basis[i];
    ctx->Computed = FNVinited+FNV512state;
    return fnvSuccess;
    }
return fnvNull;
}   /* end FNV512init */

/* initialize context with a provided basis  (32 bit)
 *************************************************************/
int FNV512initBasis ( FNV512context *ctx,
                      const uint8_t basis[FNV512size] )
{
int      i;
const uint8_t  *ui8p;
uint32_t temp;

if ( ctx )
    {
#ifdef FNV_BigEndian
    ui8p = basis;
    for ( i=0; i < FNV512size/2; ++i )
        {
        temp = *ui8p++;
        ctx->Hash[i] = ( temp<<8 ) + (*ui8p++);
        }
#else
    ui8p = basis + ( FNV512size/2 - 1 );
    for ( i=0; i < FNV512size/2; ++i )
        {
```

```
        temp = *ui8p--;
        ctx->Hash[i] = ( temp<<8 ) + (*ui8p--);
        }
#endif
    ctx->Computed = FNVinited+FNV512state;
    return fnvSuccess;
    }
return fnvNull;
}   /* end FNV512initBasis */

/* hash in a counted block  (32 bit)
 ******************************************************************/
int FNV512blockin ( FNV512context *ctx,
                    const void *vin,
                    long int length )
{
const uint8_t *in = (const uint8_t*)vin;
uint32_t   temp[FNV512size/2];
uint32_t   temp2[6];
int        i;

if ( ctx && in )
    {
    switch ( ctx->Computed )
        {
        case FNVinited+FNV512state:
            ctx->Computed = FNVcomputed+FNV512state;
        case FNVcomputed+FNV512state:
            break;
        default:
            return fnvStateError;
        }
    if ( length < 0 )
        return fnvBadParam;
    for ( i=0; i<FNV512size/2; ++i )
        temp[i] = ctx->Hash[i];
    for ( ; length > 0; length-- )
        {
        /* temp = FNV512prime * ( temp ^ *in++ ); */
        temp[15] ^= *in++;
        for ( i=0; i<6; ++i )
            temp2[i] = temp[10+i] << FNV512shift;
        for ( i=0; i<FNV512size/2; ++i )
            temp[i] *= FNV512primeX;
        for ( i=0; i<6; ++i )
            temp[10+i] += temp2[i];
        for ( i=15; i>0; --i )
            {
            temp[i-1] += temp[i] >> 16;
            temp[i] &= 0xFFFF;
```

```
            }
        }
    for ( i=0; i<FNV512size/2; ++i )
        ctx->Hash[i] = temp[i];
    return fnvSuccess;
    }
return fnvNull;
}   /* end FNV512blockin */

/* hash in a string  (32 bit)
 ***************************************************************/
int FNV512stringin ( FNV512context *ctx, const char *in )
{
uint32_t   temp[FNV512size/2];
uint32_t   temp2[6];
int        i;
uint8_t    ch;

if ( ctx && in )
    {
    switch ( ctx->Computed )
        {
        case FNVinited+FNV512state:
            ctx->Computed = FNVcomputed+FNV512state;
        case FNVcomputed+FNV512state:
            break;
        default:
             return fnvStateError;
         }
    for ( i=0; i<FNV512size/2; ++i )
         temp[i] = ctx->Hash[i];
    while ( (ch = (uint8_t)*in++) )
        {
        /* temp = FNV512prime * ( temp ^ *in++ ); */
        temp[15] ^= ch;
        for ( i=0; i<6; ++i )
            temp2[i] = temp[10+i] << FNV512shift;
        for ( i=0; i<FNV512size/2; ++i )
            temp[i] *= FNV512primeX;
        for ( i=0; i<6; ++i )
            temp[10+i] += temp2[i];
        for ( i=15; i>0; --i )
            {
            temp[i-1] += temp[i] >> 16;
            temp[i] &= 0xFFFF;
            }
         }
    for ( i=0; i<FNV512size/2; ++i )
        ctx->Hash[i] = temp[i];
    return fnvSuccess;
```

```
    }
return fnvNull;
}   /* end FNV512stringin */

/* return hash  (32 bit)
 ****************************************************************/
int FNV512result ( FNV512context *ctx, unsigned char out[16] )
{
int    i;

if ( ctx && out )
    {
    if ( ctx->Computed != FNVcomputed+FNV512state )
        return fnvStateError;
    for ( i=0; i<FNV512size/2; ++i )
        {
#ifdef FNV_BigEndian
        out[31-2*i] = ctx->Hash[i];
        out[30-2*i] = ctx->Hash[i] >> 8;
#else
        out[2*i] = ctx->Hash[i];
        out[2*i+1] = ctx->Hash[i] >> 8;
#endif
        ctx->Hash[i] = 0;
        }
    ctx->Computed = FNVemptied+FNV512state;
    return fnvSuccess;
    }
return fnvNull;
}   /* end FNV512result */

#endif    /*  FNV_64bitIntegers */
/*****************************************************************
 *        END VERSION FOR WHEN YOU ONLY HAVE 32-BIT ARITHMETIC      *
 *****************************************************************/

#endif    /* _FNV512_C_ */
```

<CODE ENDS>

6.1.6 FNV1024 C Code

The header and C source for 1024-bit FNV-1a returning a byte vector.

<CODE BEGINS>

```
/************************** FNV1024.h ************************/
/***************** See RFC NNNN for details. ******************/
/*
 * Copyright (c) 2016 IETF Trust and the persons identified as
```

```
 * authors of the code.  All rights reserved.
 * See fnv-private.h for terms of use and redistribution.
 */

#ifndef _FNV1024_H_
#define _FNV1024_H_

/*
 *  Description:
 *      This file provides headers for the 1024-bit version of the
 *      FNV-1a non-cryptographic hash algorithm.
 */

#include "FNVconfig.h"

#include <stdint.h>
#define FNV1024size (1024/8)

/* If you do not have the ISO standard stdint.h header file, then you
 * must typedef the following types:
 *
 *    type              meaning
 *  uint64_t         unsigned 64 bit integer (ifdef FNV_64bitIntegers)
 *  uint32_t         unsigned 32 bit integer
 *  uint16_t         unsigned 16 bit integer
 *  uint8_t          unsigned 8 bit integer (i.e., unsigned char)
 */

#ifndef _FNV_ErrCodes_
#define _FNV_ErrCodes_
/*****************************************************************
 *  All FNV functions provided return as integer as follows:
 *       0 -> success
 *      >0 -> error as listed below
 */
enum {    /* success and errors */
    fnvSuccess = 0,
    fnvNull,            /* Null pointer parameter */
    fnvStateError,      /* called Input after Result or before Init */
    fnvBadParam         /* passed a bad parameter */
};
#endif /* _FNV_ErrCodes_ */

/*
 *  This structure holds context information for an FNV1024 hash
 */
#ifdef FNV_64bitIntegers
    /* version if 64 bit integers supported */
typedef struct FNV1024context_s {
        int Computed;  /* state */
```

```
        uint32_t Hash[FNV1024size/4];
} FNV1024context;

#else
    /* version if 64 bit integers NOT supported */

typedef struct FNV1024context_s {
        int Computed;  /* state */
        uint16_t Hash[FNV1024size/2];
} FNV1024context;

#endif /* FNV_64bitIntegers */

/*
 *  Function Prototypes
 *    FNV1024string: hash a zero terminated string not including
 *                   the terminating zero
 *    FNV1024block: FNV1024 hash a specified length byte vector
 *    FNV1024init: initializes an FNV1024 context
 *    FNV1024initBasis: initializes an FNV1024 context with a
 *                      provided basis
 *    FNV1024blockin: hash in a specified length byte vector
 *    FNV1024stringin: hash in a zero terminated string not
 *                     including the zero
 *    FNV1024result: returns the hash value
 *
 *    Hash is returned as an array of 8-bit integers
 */

#ifdef __cplusplus
extern "C" {
#endif

/* FNV1024 */
extern int FNV1024string ( const char *in,
                           unsigned char out[FNV1024size] );
extern int FNV1024block ( const void *in,
                          long int length,
                          unsigned char out[FNV1024size] );
extern int FNV1024init ( FNV1024context *);
extern int FNV1024initBasis ( FNV1024context * const,
                              const uint8_t basis[FNV1024size] );
extern int FNV1024blockin ( FNV1024context *,
                            const void *in,
                            long int length );
extern int FNV1024stringin ( FNV1024context *,
                             const char *in );
extern int FNV1024result ( FNV1024context *,
                           unsigned char out[FNV1024size] );
```

```
#ifdef __cplusplus
}
#endif

#endif /* _FNV1024_H_ */
   <CODE ENDS>

   <CODE BEGINS>
/***************************** FNV1024.c ***************************/
/******************* See RFC NNNN for details ********************/
/* Copyright (c) 2016, 2017 IETF Trust and the persons identified as
 * authors of the code.  All rights
 * See fnv-private.h for terms of use and redistribution.
 */

/* This file implements the FNV (Fowler, Noll, Vo) non-cryptographic
 * hash function FNV-1a for 1024-bit hashes.
 */

#ifndef _FNV1024_C_
#define _FNV1024_C_

#include "fnv-private.h"
#include "FNV1024.h"

/* common code for 64 and 32 bit modes */

/* FNV1024 hash a null terminated string  (64/32 bit)
 ***************************************************************/
int FNV1024string ( const char *in, uint8_t out[FNV1024size] )
{
FNV1024context    ctx;
int              err;

if ( (err = FNV1024init ( &ctx )) != fnvSuccess)
    return err;
if ( (err = FNV1024stringin ( &ctx, in )) != fnvSuccess)
    return err;
return FNV1024result ( &ctx, out );
}   /* end FNV1024string */

/* FNV1024 hash a counted block  (64/32 bit)
 ***************************************************************/
int FNV1024block ( const void *in,
                   long int length,
                   uint8_t out[FNV1024size] )
{
FNV1024context    ctx;
int              err;
```

```
if ( (err = FNV1024init ( &ctx )) != fnvSuccess)
    return err;
if ( (err = FNV1024blockin ( &ctx, in, length)) != fnvSuccess)
    return err;
return FNV1024result ( &ctx, out );
}   /* end FNV1024block */


/*****************************************************************
 *        START VERSION FOR WHEN YOU HAVE 64 BIT ARITHMETIC      *
 *****************************************************************/
#ifdef FNV_64bitIntegers

/*
 1024 bit FNV_prime = 2^680 + 2^8 + 0x8d =
   0x0000000000000000 0000010000000000
     0000000000000000 0000000000000000
     0000000000000000 0000000000000000
     0000000000000000 0000000000000000
     0000000000000000 0000000000000000
     0000000000000000 000000000000018D
#define FNV1024primeX 0x018D
#define FNV1024shift 24

/* 0x0000000000000000 005F7A76758ECC4D
     32E56D5A591028B7 4B29FC4223FDADA1
     6C3BF34EDA3674DA 9A21D90000000000
     0000000000000000 0000000000000000
     0000000000000000 0000000000000000
     0000000000000000 000000000004C6D7
     EB6E73802734510A 555F256CC005AE55
     6BDE8CC9C6A93B21 AFF4B16C71EE90B3 */

uint32_t FNV1024basis[FNV1024size/4] = {
  0x00000000, 0x00000000, 0x005F7A76, 0x758ECC4D,
  0x32E56D5A, 0x591028B7, 0x4B29FC42, 0x23FDADA1,
  0x6C3BF34E, 0xDA3674DA, 0x9A21D900, 0x00000000,
  0x00000000, 0x00000000, 0x00000000, 0x00000000,
  0x00000000, 0x00000000, 0x00000000, 0x00000000,
  0x00000000, 0x00000000, 0x00000000, 0x0004C6D7,
  0xEB6E7380, 0x2734510A, 0x555F256C, 0xC005AE55,
  0x6BDE8CC9, 0xC6A93B21, 0xAFF4B16C, 0x71EE90B3
 };

/*****************************************************************
 *         Set of init, input, and output functions below        *
 *         to incrementally compute FNV1024                       *
 *****************************************************************/

/* initialize context  (64 bit)
```

```
 ************************************************************************/
int FNV1024init ( FNV1024context *ctx )
{
if ( ctx )
    {
    for ( i=0; i<FNV1024size/4; ++i )
        ctx->Hash[i] = FNV1024basis[i];
    ctx->Computed = FNVinited+FNV1024state;
    return fnvSuccess;
    }
return fnvNull;
}   /* end FNV1024init */

/* initialize context with a provided basis  (64 bit)
 ************************************************************************/
int FNV1024initBasis ( FNV1024context* const ctx,
                       const uint8_t basis[FNV1024size] )
{
int      i;
uint8_t  *ui8p;
uint32_t  temp;

if ( ctx )
    {
#ifdef FNV_BigEndian
    ui8p = basis;
    for ( i=0; i < FNV1024size/4; ++i )
        {
            temp = (*ui8p++)<<8;
            temp = (temp + *ui8p++)<<8;
            temp = (temp + *ui8p++)<<8;
        ctx->Hash[i] = temp +  *ui8p;
        }
#else
    ui8p = basis + (FNV1024size/4 - 1);
    for ( i=0; i < FNV1024size/4; ++i )
        {
            temp = (*ui8p--)<<8;
            temp = (temp + *ui8p--)<<8;
            temp = (temp + *ui8p--)<<8;
        ctx->Hash[i] = temp +  *ui8p;
        }
#endif
    ctx->Computed = FNVinited+FNV1024state;
    return fnvSuccess;
    }
return fnvNull;
}   /* end FNV1024initBasis */

/* hash in a counted block  (64 bit)
```

```
 ****************************************************************/
int FNV1024blockin ( FNV1024context *ctx,
                     const void *vin,
                     long int length )
{
const uint8_t *in = (const uint8_t*)vin;
uint64_t    temp[FNV1024size/4];
uint64_t    temp2[3];

if ( ctx && in )
    {
    switch ( ctx->Computed )
        {
        case FNVinited+FNV1024state:
            ctx->Computed = FNVcomputed+FNV128state;
        case FNVcomputed+FNV1024state:
            break;
        default:
            return fnvStateError;
        }
    if ( length < 0 )
        return fnvBadParam;
    for ( i=0; i<FNV1024size/4; ++i )
         temp[i] = ctx->Hash[i];
    for ( ; length > 0; length-- )
        {
        /* temp = FNV1024prime * ( temp ^ *in++ ); */
        temp[7] ^= *in++;
        temp2[2] = temp[7] << FNV1024shift;
        temp2[1] = temp[6] << FNV1024shift;
        temp2[0] = temp[5] << FNV1024shift;
        for ( i=0; i<FNV1024size/4; ++i )
            temp[i] *= FNV1024primeX;
        temp[2] += temp2[2];
        temp[1] += temp2[1];
        temp[0] += temp2[0];
        for ( i=FNV1024size/4-1; i>0; --i )
            {
            temp[i-1] += temp[i] >> 16;
            temp[i] &= 0xFFFF;
            }
       }
    for ( i=0; i<FNV1024size/4; ++i )
        ctx->Hash[i] = temp[i];
    return fnvSuccess;
    }
return fnvNull;
}   /* end FNV1024input */

/* hash in a string  (64 bit)
```

```
 ***************************************************************/
inf FNV1024stringin ( FNV1024context *ctx, const char *in )
{
uint64_t   temp[FNV1024size/4];
uint64_t   temp2[2];
int        i;
uint8_t    ch;

if ( ctx && in )
    {
    switch ( ctx->Computed )
        {
        case FNVinited+FNV1024state:
            ctx->Computed = FNVcomputed+FNV1024state;
        case FNVcomputed+FNV1024state:
            break;
        default:
             return fnvStateError;
         }
    for ( i=0; i<FNV1024size/4; ++i )
         temp[i] = ctx->Hash[i];
    while ( ch = (uint8_t)*in++ )
        {
        /* temp = FNV1024prime * ( temp ^ ch ); */
        temp[7] ^= ch;
        temp2[2] = temp[7] << FNV128shift;
        temp2[1] = temp[6] << FNV128shift;
        temp2[0] = temp[5] << FNV128shift;
        for ( i=0; i<FNV1024size/4; ++i )
            temp[i] *= FNV1024prime;
        temp[2] += temp2[2];
        temp[1] += temp2[1];
        temp[0] += temp2[0];
        for ( i=FNVsize1024/4-1; i>0; --i )
            {
            temp[i-1] += temp[i] >> 16;
            temp[i] &= 0xFFFF;
            }
       }
    for ( i=0; i<FNV1024size/4; ++i )
        ctx->Hash[i] = temp[i];
    return fnvSuccess;
    }
return fnvNull;
}   /* end FNV1024stringin */

/* return hash  (64 bit)
 ***************************************************************/
int FNV1024result ( FNV1024context *ctx, uint8_t out[FNV1024size] )
{
```

```
if ( ctx && out )
    {
    if ( ctx->Computed != FNVcomputed+FNV1024state )
        return fnvStateError;
    for ( i=0; i<FNV1024size/4; ++i )
        {
#ifdef FNV_BigEndian
        out[15-2*i] = ctx->Hash[i];
        out[14-2*i] = ctx->Hash[i] >> 8;
#else
        out[2*i] = ctx->Hash[i];
        out[2*i+1] = ctx->Hash[i] >> 8;
#endif
        ctx -> Hash[i] = 0;
        }
    ctx->Computed = FNVemptied+FNV1024state;
    return fnvSuccess;
    }
return fnvNull;
}   /* end FNV1024result */

/*****************************************************************
 *        END VERSION FOR WHEN YOU HAVE 64 BIT ARITHMETIC        *
 *****************************************************************/
#else    /*  FNV_64bitIntegers */
/*****************************************************************
 *      START VERSION FOR WHEN YOU ONLY HAVE 32-BIT ARITHMETIC   *
 *****************************************************************/

/* version for when you only have 32-bit arithmetic
 *****************************************************************/

/*
 1024 bit FNV_prime = 2^680 + 2^8 + 0x8d =
   0x0000000000000000 0000010000000000
     0000000000000000 0000000000000000
     0000000000000000 0000000000000000
     0000000000000000 0000000000000000
     0000000000000000 0000000000000000
     0000000000000000 000000000000018D */
#define FNV1024primeX 0x018D
#define FNV1024shift 8

/* 0x0000000000000000 005F7A76758ECC4D
     32E56D5A591028B7 4B29FC4223FDADA1
     6C3BF34EDA3674DA 9A21D90000000000
     0000000000000000 0000000000000000
     0000000000000000 0000000000000000
     0000000000000000 000000000004C6D7
     EB6E73802734510A 555F256CC005AE55
```

```
     6BDE8CC9C6A93B21 AFF4B16C71EE90B3 */

uint16_t FNV1024basis[FNV1024size/2] = {
  0x0000, 0x0000, 0x0000, 0x0000, 0x005F, 0x7A76, 0x758E, 0xCC4D,
  0x32E5, 0x6D5A, 0x5910, 0x28B7, 0x4B29, 0xFC42, 0x23FD, 0xADA1,
  0x6C3B, 0xF34E, 0xDA36, 0x74DA, 0x9A21, 0xD900, 0x0000, 0x0000,
  0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
  0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
  0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0004, 0xC6D7,
  0xEB6E, 0x7380, 0x2734, 0x510A, 0x555F, 0x256C, 0xC005, 0xAE55,
  0x6BDE, 0x8CC9, 0xC6A9, 0x3B21, 0xAFF4, 0xB16C, 0x71EE, 0x90B3
};


/*********************************************************************
 *         Set of init, input, and output functions below            *
 *         to incrementally compute FNV1024                          *
 *********************************************************************/

/* initialize context  (32 bit)
 *********************************************************************/
int FNV1024init ( FNV1024context *ctx )
{
int     i;

if ( ctx )
    {
    for ( i=0; i<FNV1024size/2; ++i )
        ctx->Hash[i] = FNV1024basis[i];
    ctx->Computed = FNVinited+FNV1024state;
    return fnvSuccess;
    }
return fnvNull;
}   /* end FNV1024init */

/* initialize context with a provided basis  (32 bit)
 *********************************************************************/
int FNV1024initBasis ( FNV1024context *ctx,
                       const uint8_t basis[FNV1024size] )
{
int      i;
const uint8_t  *ui8p;
uint32_t temp;

if ( ctx )
    {
#ifdef FNV_BigEndian
    ui8p = basis;
    for ( i=0; i < FNV1024size/2; ++i )
        {
```

```
        temp = *ui8p++;
        ctx->Hash[i] = ( temp<<8 ) + (*ui8p++);
        }
#else
    ui8p = basis + ( FNV1024size/2 - 1 );
    for ( i=0; i < FNV1024size/2; ++i )
        {
        temp = *ui8p--;
        ctx->Hash[i] = ( temp<<8 ) + (*ui8p--);
        }
#endif
    ctx->Computed = FNVinited+FNV1024state;
    return fnvSuccess;
    }
return fnvNull;
}   /* end FNV1024initBasis */

/* hash in a counted block  (32 bit)
 ****************************************************************/
int FNV1024blockin ( FNV1024context *ctx,
                     const void *vin,
                     long int length )
{
const uint8_t *in = (const uint8_t*)vin;
uint32_t   temp[FNV1024size/2];
uint32_t   temp2[6];
int        i;

if ( ctx && in )
    {
    switch ( ctx->Computed )
        {
        case FNVinited+FNV1024state:
            ctx->Computed = FNVcomputed+FNV1024state;
        case FNVcomputed+FNV1024state:
            break;
        default:
            return fnvStateError;
        }
    if ( length < 0 )
        return fnvBadParam;
    for ( i=0; i<FNV1024size/2; ++i )
        temp[i] = ctx->Hash[i];
    for ( ; length > 0; length-- )
        {
        /* temp = FNV1024prime * ( temp ^ *in++ ); */
        temp[15] ^= *in++;
        for ( i=0; i<6; ++i )
            temp2[i] = temp[10+i] << FNV1024shift;
        for ( i=0; i<FNV1024size/2; ++i )
```

```
            temp[i] *= FNV1024primeX;
        for ( i=0; i<6; ++i )
            temp[10+i] += temp2[i];
        for ( i=15; i>0; --i )
            {
            temp[i-1] += temp[i] >> 16;
            temp[i] &= 0xFFFF;
            }
        }
    for ( i=0; i<FNV1024size/2; ++i )
        ctx->Hash[i] = temp[i];
    return fnvSuccess;
    }
return fnvNull;
}   /* end FNV1024blockin */

/* hash in a string  (32 bit)
 ***************************************************************/
int FNV1024stringin ( FNV1024context *ctx, const char *in )
{
uint32_t   temp[FNV1024size/2];
uint32_t   temp2[6];
int        i;
uint8_t    ch;

if ( ctx && in )
    {
    switch ( ctx->Computed )
        {
        case FNVinited+FNV1024state:
            ctx->Computed = FNVcomputed+FNV1024state;
        case FNVcomputed+FNV1024state:
            break;
        default:
             return fnvStateError;
         }
    for ( i=0; i<FNV1024size/2; ++i )
         temp[i] = ctx->Hash[i];
    while ( (ch = (uint8_t)*in++) )
        {
        /* temp = FNV1024prime * ( temp ^ *in++ ); */
        temp[15] ^= ch;
        for ( i=0; i<6; ++i )
            temp2[i] = temp[10+i] << FNV1024shift;
        for ( i=0; i<FNV1024size/2; ++i )
            temp[i] *= FNV1024primeX;
        for ( i=0; i<6; ++i )
            temp[10+i] += temp2[i];
        for ( i=15; i>0; --i )
            {
```

```
            temp[i-1] += temp[i] >> 16;
            temp[i] &= 0xFFFF;
            }
         }
    for ( i=0; i<FNV1024size/2; ++i )
        ctx->Hash[i] = temp[i];
    return fnvSuccess;
    }
return fnvNull;
}   /* end FNV1024stringin */

/* return hash  (32 bit)
 ***************************************************************/
int FNV1024result ( FNV1024context *ctx, unsigned char out[16] )
{
int    i;

if ( ctx && out )
    {
    if ( ctx->Computed != FNVcomputed+FNV1024state )
        return fnvStateError;
    for ( i=0; i<FNV1024size/2; ++i )
        {
#ifdef FNV_BigEndian
        out[31-2*i] = ctx->Hash[i];
        out[30-2*i] = ctx->Hash[i] >> 8;
#else
        out[2*i] = ctx->Hash[i];
        out[2*i+1] = ctx->Hash[i] >> 8;
#endif
        ctx->Hash[i] = 0;
        }
    ctx->Computed = FNVemptied+FNV1024state;
    return fnvSuccess;
    }
return fnvNull;
}   /* end FNV1024result */

#endif    /*  FNV_64bitIntegers */
/*****************************************************************
 *        END VERSION FOR WHEN YOU ONLY HAVE 32-BIT ARITHMETIC      *
 *****************************************************************/

#endif    /* _FNV1024_C_ */
   <CODE ENDS>
```

6.2 FNV Test Code

Here is a test driver:

<CODE BEGINS>

```
/***************************** MAIN.c ****************************/
/****************** See RFC NNNN for details. ********************/
/*
 * Copyright (c) 2016 IETF Trust and the persons identified as
 * authors of the code.  All rights reserved.
 * See fnv-private.h for terms of use and redistribution.
 */
/* to do a thorough test you need to run will all four
 combinations of the following defined/undefined */

// #define FNV_64bitIntegers
// #define FNV_BigEndian

#include <stdio.h>
#include <string.h>

#include "fnv-private.h"
#include "FNV.h"

/* global variables */
char    *funcName;
char    *errteststring = "foo";
int     Terr; /* Total errors */
#define NTestBytes 3
uint8_t errtestbytes[NTestBytes] = { (uint8_t)1,
    (uint8_t)2, (uint8_t)3 };

#define NTstrings 3
char    *teststring[NTstrings] = { "", "a", "foobar" };

/*****************************************************************
 * local prototypes
 *****************************************************************/
int TestR ( char *, int should, int was );
void TestNValue ( char *subfunc,
                  char *string,
                  int N,
                  uint8_t *should,
                  uint8_t *was );
void HexPrint ( int i, unsigned char *p );
void Test32 ();
void Test32Value ( char *subfunc, char *string,
                   uint32_t was, uint32_t should );
void Test64 ();
#ifdef FNV_64bitIntegers
```

```
void Test64Value ( char *subfunc, char *string,
                   uint64_t should, uint64_t was );
#else
#define uint64_t foobar
#endif /* FNB_64bitIntegers */
void Test128 ();
void Test256 ();
void Test512 ();
void Test1024 ();

void TestNValue ( char *subfunc,
                  char *string,
                  int N,
                  uint8_t was[N],
                  uint8_t should[N] );

/*****************************************************************
 * main
 *****************************************************************/
int main (int argc, const char * argv[])
{
#ifdef FNV_64bitIntegers
    printf ("Have 64-bit Integers. ");
#else
    printf ("Do not have 64-bit integers. ");
#endif
#ifdef FNV_BigEndian
    printf ("Calculating for Big Endian.0);
#else
    printf ("Not calculating for Big Endian.0);
#endif
    funcName = "Testing TestR ";
         /* test the Test Return function */
    TestR ( "should fail", 1, 2 );
    TestR ( "should not have failed", 0, 0 );

    Test32();
    Test64();
    Test128();
    Test256();
    Test512();
    Test1024();

    printf ("Type return to exit.0);
    (void)getchar();
    printf ("Goodbye!0);

    return 0;
}     /* end main */
```

```
/* Test status returns
 ***************************************************************/
int TestR ( char *name, int expect, int actual )
{
if ( expect != actual )
    {
    printf ( "%s%s returned %i instead of %i.0,
              funcName, name, actual, expect );
    ++Terr;
    }
return actual;
}    /* end TestR */

/* Return true if the bytes are in reverse order from each other */
int revcmp(uint8_t *buf1, uint8_t *buf2, int N) {
  int i;
  uint8_t *bufc = buf2 + N;
  for ( i = 0; i < N / 2; i++ )
    if (*buf1++ != *--bufc)
      return 0;
  return 1;
}

/* General byte vector return error test
 ***************************************************************/
void TestNValue ( char *subfunc,
                  char *string,
                  int N,
                  uint8_t *was,
                  uint8_t *should )
{
#ifdef FNV_BigEndian
if ( revcmp ( was, should, N) == 0)
#else
if ( memcmp ( was, should, N) != 0)
#endif
    {
    printf ( "%s %s of '%s' computed ", funcName, subfunc, string );
    HexPrint ( N, was );
    printf ( ", should have been " );
    HexPrint ( N, should );
    printf ( ".0 );
    ++Terr;
    }
}   /* end TestNValue */

/* print some hex
 ***************************************************************/
 void HexPrint ( int count, unsigned char *ptr )
{
```

```
int   i;

for ( i = 0; i < count; ++i )
  printf ( "%02X", ptr[i] );
}   /* end HexPrint */


/*****************************************************************
 * FNV32 test
 *****************************************************************/
void Test32 ()
{
int            i, err;
long int       iLen;
uint32_t       eUint32;
FNV32context   eContext;
uint32_t    FNV32svalues[NTstrings] = {
    0x811c9dc5, 0xe40c292c, 0xbf9cf968 };
uint32_t    FNV32bvalues[NTstrings] = {
    0x050c5d1f, 0x2b24d044, 0x0c1c9eb8 };

/* test Test32Value */
funcName = "Test32Value";
Test32Value ( "should fail", "test", FNV32svalues[1], FNV32svalues[2] );

funcName = "FNV32";

/* test error checks */
Terr = 0;
TestR ( "init", fnvSuccess, FNV32init (&eContext) );
    TestR ( "string", fnvNull,
            FNV32string ( (char *)0, &eUint32 ) );
    TestR ( "string", fnvNull,
            FNV32string ( errteststring, (uint32_t *)0 ) );
    TestR ( "block", fnvNull,
            FNV32block ( (uint8_t *)0, 1, &eUint32 ) );
    TestR ( "block", fnvBadParam,
            FNV32block ( errtestbytes, -1, &eUint32 ) );
    TestR ( "block", fnvNull,
            FNV32block ( errtestbytes, 1, (uint32_t *)0 ) );
    TestR ( "init", fnvNull,
            FNV32init ( (FNV32context *)0 ) );
    TestR ( "initBasis", fnvNull,
            FNV32initBasis ( (FNV32context *)0, 42 ) );
    TestR ( "blockin", fnvNull,
            FNV32blockin ( (FNV32context *)0,
                           errtestbytes, NTestBytes ) );
    TestR ( "blockin", fnvNull,
            FNV32blockin ( &eContext, (uint8_t *)0,
                           NTestBytes ) );
```

```
    TestR ( "blockin", fnvBadParam,
            FNV32blockin ( &eContext, errtestbytes, -1 ) );
    eContext.Computed = FNVclobber+FNV32state;
    TestR ( "blockin", fnvStateError,
            FNV32blockin ( &eContext, errtestbytes,
                           NTestBytes ) );
    TestR ( "stringin", fnvNull,
            FNV32stringin ( (FNV32context *)0, errteststring ) );
    TestR ( "stringin", fnvNull,
            FNV32stringin ( &eContext, (char *)0 ) );
    TestR ( "stringin", fnvStateError,
            FNV32stringin ( &eContext, errteststring ) );
    TestR ( "result", fnvNull,
            FNV32result ( (FNV32context *)0, &eUint32 ) );
    TestR ( "result", fnvNull,
            FNV32result ( &eContext, (uint32_t *)0  ) );
    TestR ( "result", fnvStateError,
            FNV32result ( &eContext, &eUint32  ) );
if ( Terr )
    printf ( "%s test of error checks failed %i times.0,
             funcName, Terr );
else
    printf ( "%s test of error checks passed0, funcName );

/* test actual results */
Terr = 0;
for ( i = 0; i < NTstrings; ++i )
    {
    err = TestR ( "string", fnvSuccess,
                  FNV32string ( teststring[i], &eUint32 ) );
    if ( err == fnvSuccess )
        Test32Value ( "string", teststring[i], eUint32,
                      FNV32svalues[i] );
    err = TestR ( "block", fnvSuccess,
                  FNV32block ( (uint8_t *)teststring[i],
                               (unsigned long)(strlen(teststring[i])+1),
                               &eUint32 ) );
    if ( err == fnvSuccess )
        Test32Value ( "block", teststring[i], eUint32,
                      FNV32bvalues[i] );
    /* now try testing the incremental stuff */
    err = TestR ( "init", fnvSuccess, FNV32init ( &eContext ) );
    if ( err ) break;
    iLen = strlen ( teststring[i] );
    err = TestR ( "blockin", fnvSuccess,
                  FNV32blockin ( &eContext,
                                 (uint8_t *)teststring[i],
                                 iLen/2 ) );
    if ( err ) break;
    err = TestR ( "stringin", fnvSuccess,
```

```
                FNV32stringin ( &eContext,
                                teststring[i] + iLen/2 ) );
    err = TestR ( "result", fnvSuccess,
                FNV32result ( &eContext, &eUint32 ) );
    if ( err ) break;
    Test32Value ( " incremental", teststring[i], eUint32,
                FNV32svalues[i] );
    }
if ( Terr )
    printf ( "%s test of return values failed %i times.0,
             funcName, Terr );
else
    printf ( "%s test of return values passed.0, funcName );
}    /* end Test32 */

/* start Test32Value
 ****************************************************************/
void Test32Value ( char *subfunc,
                   char *string,
                   uint32_t was,
                   uint32_t should )
{
  TestNValue(subfunc, string, sizeof(uint32_t), (uint8_t*)&was,
                                                (uint8_t*)&should);
}   /* end Test32Value */


#ifdef FNV_64bitIntegers
/***************************************************************
 * Code for FNV64 using 64-bit integers
 ***************************************************************/

void Test64 ()
{
int    i, err;
uint64_t   eUint64 = 42;
FNV64context   eContext;
uint64_t   FNV64svalues[NTstrings] = {
 0xcbf29ce484222325, 0xaf63dc4c8601ec8c, 0x85944171f73967e8 };
uint64_t   FNV64bvalues[NTstrings] = {
 0xaf63bd4c8601b7df, 0x089be207b544f1e4, 0x34531ca7168b8f38 };

funcName = "FNV64";

/* test error checks */
Terr = 0;
TestR ( "init", fnvSuccess, FNV64init (&eContext) );
    TestR ( "string", fnvNull,
             FNV64string ( (char *)0, &eUint64 ) );
    TestR ( "string", fnvNull,
```

```
            FNV64string ( errteststring, (uint64_t *)0 ) );
    TestR ( "block", fnvNull,
            FNV64block ( (uint8_t *)0, 1, &eUint64 ) );
    TestR ( "block", fnvBadParam,
            FNV64block ( errtestbytes, -1, &eUint64 ) );
    TestR ( "block", fnvNull,
            FNV64block ( errtestbytes, 1, (uint64_t *)0 ) );
    TestR ( "init", fnvNull,
            FNV64init ( (FNV64context *)0 ) );
    TestR ( "initBasis", fnvNull,
            FNV64initBasis ( (FNV64context *)0, 42 ) );
    TestR ( "blockin", fnvNull,
            FNV64blockin ( (FNV64context *)0,
                           errtestbytes, NTestBytes ) );
    TestR ( "blockin", fnvNull,
            FNV64blockin ( &eContext, (uint8_t *)0,
                           NTestBytes ) );
    TestR ( "blockin", fnvBadParam,
            FNV64blockin ( &eContext, errtestbytes, -1 ) );
    eContext.Computed = FNVclobber+FNV64state;
    TestR ( "blockin", fnvStateError,
            FNV64blockin ( &eContext, errtestbytes,
                           NTestBytes ) );
    TestR ( "stringin", fnvNull,
            FNV64stringin ( (FNV64context *)0, errteststring ) );
    TestR ( "stringin", fnvNull,
            FNV64stringin ( &eContext, (char *)0 ) );
    TestR ( "stringin", fnvStateError,
            FNV64stringin ( &eContext, errteststring ) );
    TestR ( "result", fnvNull,
            FNV64result ( (FNV64context *)0, &eUint64 ) );
    TestR ( "result", fnvNull,
            FNV64result ( &eContext, (uint64_t *)0  ) );
    TestR ( "result", fnvStateError,
            FNV64result ( &eContext, &eUint64  ) );
if ( Terr )
    printf ( "%s test of error checks failed %i times.0,
             funcName, Terr );
else
    printf ( "%s test of error checks passed0, funcName );

/* test actual results */
Terr = 0;
for ( i = 0; i < NTstrings; ++i )
    {
    err = TestR ( "string", fnvSuccess,
                  FNV64string ( teststring[i], &eUint64 ) );
    if ( err == fnvSuccess )
        Test64Value ( "string", teststring[i], eUint64,
                      FNV64svalues[i] );
```

```
    err = TestR ( "block", fnvSuccess,
                  FNV64block ( (uint8_t *)teststring[i],
                               (unsigned long)(strlen(teststring[i])+1),
                               &eUint64 ) );
    if ( err == fnvSuccess )
         Test64Value ( "block", teststring[i], eUint64,
                       FNV64bvalues[i] );
    /* now try testing the incremental stuff */
    err = TestR ( "init", fnvSuccess, FNV64init ( &eContext ) );

    }
if ( Terr )
    printf ( "%s test of return values failed %i times.0,
             funcName, Terr );
else
    printf ( "%s test of return values passed.0, funcName );
}    /* end Test64 */

/* start Test64Value
 ****************************************************************/
 void Test64Value ( char *subfunc,
                    char *string,
                    uint64_t should,
                    uint64_t was )
{
  TestNValue(subfunc, string, sizeof(uint64_t), (uint8_t*)&was,
.                                          (uint8_t*)&should);
}   /* end Test64Value */
#else
void Test64 ()
{
  /* TBD */
}
#endif /* FNV_64bitIntegers */


/****************************************************************
 * Code for FNV128 using 64-bit integers
 ****************************************************************/

void Test128 ()
{
//int            i, err;
uint8_t        eUint128[FNV128size];
FNV128context  eContext;

funcName = "FNV128";

/* test error checks */
Terr = 0;
```

```
TestR ( "init", fnvSuccess, FNV128init (&eContext) );
    TestR ( "string", fnvNull,
            FNV128string ( (char *)0, eUint128 ) );
    TestR ( "string", fnvNull,
            FNV128string ( errteststring, (uint8_t *)0 ) );
    TestR ( "block", fnvNull,
            FNV128block ( (uint8_t *)0, 1, eUint128 ) );
    TestR ( "block", fnvBadParam,
            FNV128block ( errtestbytes, -1, eUint128 ) );
    TestR ( "block", fnvNull,
            FNV128block ( errtestbytes, 1, (uint8_t *)0 ) );
    TestR ( "init", fnvNull,
            FNV128init ( (FNV128context *)0 ) );
    TestR ( "initBasis", fnvNull,
            FNV128initBasis ( (FNV128context *)0, eUint128 ) );
    TestR ( "blockin", fnvNull,
            FNV128blockin ( (FNV128context *)0,
                            errtestbytes, NTestBytes ) );
    TestR ( "blockin", fnvNull,
            FNV128blockin ( &eContext, (uint8_t *)0,
                            NTestBytes ) );
    TestR ( "blockin", fnvBadParam,
            FNV128blockin ( &eContext, errtestbytes, -1 ) );
    eContext.Computed = FNVclobber+FNV128state;
    TestR ( "blockin", fnvStateError,
            FNV128blockin ( &eContext, errtestbytes,
                            NTestBytes ) );
    TestR ( "stringin", fnvNull,
            FNV128stringin ( (FNV128context *)0, errteststring ) );
    TestR ( "stringin", fnvNull,
            FNV128stringin ( &eContext, (char *)0 ) );
    TestR ( "stringin", fnvStateError,
            FNV128stringin ( &eContext, errteststring ) );
    TestR ( "result", fnvNull,
            FNV128result ( (FNV128context *)0, eUint128 ) );
    TestR ( "result", fnvNull,
            FNV128result ( &eContext, (uint8_t *)0  ) );
    TestR ( "result", fnvStateError,
            FNV128result ( &eContext, eUint128  ) );
if ( Terr )
    printf ( "%s test of error checks failed %i times.0,
             funcName, Terr );
else
    printf ( "%s test of error checks passed0, funcName );

/* test actual results */
Terr = 0;
/* tbd */
}    /* end Test128 */
```

```
/*****************************************************************
 * Code for FNV256 using 64-bit integers
 *****************************************************************/

void Test256 ()
{
//int            i, err;
uint8_t        eUint256[FNV256size];
FNV256context  eContext;

funcName = "FNV256";

/* test error checks */
Terr = 0;
TestR ( "init", fnvSuccess, FNV256init (&eContext) );
    TestR ( "string", fnvNull,
            FNV256string ( (char *)0, eUint256 ) );
    TestR ( "string", fnvNull,
            FNV256string ( errteststring, (uint8_t *)0 ) );
    TestR ( "block", fnvNull,
            FNV256block ( (uint8_t *)0, 1, eUint256 ) );
    TestR ( "block", fnvBadParam,
            FNV256block ( errtestbytes, -1, eUint256 ) );
    TestR ( "block", fnvNull,
            FNV256block ( errtestbytes, 1, (uint8_t *)0 ) );
    TestR ( "init", fnvNull,
            FNV256init ( (FNV256context *)0 ) );
    TestR ( "initBasis", fnvNull,
            FNV256initBasis ( (FNV256context *)0, eUint256 ) );
    TestR ( "blockin", fnvNull,
            FNV256blockin ( (FNV256context *)0,
                            errtestbytes, NTestBytes ) );
    TestR ( "blockin", fnvNull,
            FNV256blockin ( &eContext, (uint8_t *)0,
                            NTestBytes ) );
    TestR ( "blockin", fnvBadParam,
            FNV256blockin ( &eContext, errtestbytes, -1 ) );
    eContext.Computed = FNVclobber+FNV256state;
    TestR ( "blockin", fnvStateError,
            FNV256blockin ( &eContext, errtestbytes,
                            NTestBytes ) );
    TestR ( "stringin", fnvNull,
            FNV256stringin ( (FNV256context *)0, errteststring ) );
    TestR ( "stringin", fnvNull,
            FNV256stringin ( &eContext, (char *)0 ) );
    TestR ( "stringin", fnvStateError,
            FNV256stringin ( &eContext, errteststring ) );
    TestR ( "result", fnvNull,
            FNV256result ( (FNV256context *)0, eUint256 ) );
    TestR ( "result", fnvNull,
```

```
            FNV256result ( &eContext, (uint8_t *)0  ) );
    TestR ( "result", fnvStateError,
            FNV256result ( &eContext, eUint256  ) );
if ( Terr )
    printf ( "%s test of error checks failed %i times.0,
             funcName, Terr );
else
    printf ( "%s test of error checks passed0, funcName );

/* test actual results */
Terr = 0;
/* tbd */
}    /* end Test256 */


/*****************************************************************
 * Code for FNV512 using 64-bit integers
 *****************************************************************/

void Test512 ()
{
//int            i, err;
uint8_t        eUint512[FNV512size];
FNV512context  eContext;

funcName = "FNV512";

/* test error checks */
Terr = 0;
TestR ( "init", fnvSuccess, FNV512init (&eContext) );
    TestR ( "string", fnvNull,
            FNV512string ( (char *)0, eUint512 ) );
    TestR ( "string", fnvNull,
            FNV512string ( errteststring, (uint8_t *)0 ) );
    TestR ( "block", fnvNull,
            FNV512block ( (uint8_t *)0, 1, eUint512 ) );
    TestR ( "block", fnvBadParam,
            FNV512block ( errtestbytes, -1, eUint512 ) );
    TestR ( "block", fnvNull,
            FNV512block ( errtestbytes, 1, (uint8_t *)0 ) );
    TestR ( "init", fnvNull,
            FNV512init ( (FNV512context *)0 ) );
    TestR ( "initBasis", fnvNull,
            FNV512initBasis ( (FNV512context *)0, eUint512 ) );
    TestR ( "blockin", fnvNull,
            FNV512blockin ( (FNV512context *)0,
                            errtestbytes, NTestBytes ) );
    TestR ( "blockin", fnvNull,
            FNV512blockin ( &eContext, (uint8_t *)0,
                            NTestBytes ) );
```

```
    TestR ( "blockin", fnvBadParam,
            FNV512blockin ( &eContext, errtestbytes, -1 ) );
    eContext.Computed = FNVclobber+FNV512state;
    TestR ( "blockin", fnvStateError,
            FNV512blockin ( &eContext, errtestbytes,
                            NTestBytes ) );
    TestR ( "stringin", fnvNull,
            FNV512stringin ( (FNV512context *)0, errteststring ) );
    TestR ( "stringin", fnvNull,
            FNV512stringin ( &eContext, (char *)0 ) );
    TestR ( "stringin", fnvStateError,
            FNV512stringin ( &eContext, errteststring ) );
    TestR ( "result", fnvNull,
            FNV512result ( (FNV512context *)0, eUint512 ) );
    TestR ( "result", fnvNull,
            FNV512result ( &eContext, (uint8_t *)0  ) );
    TestR ( "result", fnvStateError,
            FNV512result ( &eContext, eUint512  ) );
if ( Terr )
    printf ( "%s test of error checks failed %i times.0,
             funcName, Terr );
else
    printf ( "%s test of error checks passed0, funcName );

/* test actual results */
Terr = 0;
/* tbd */
}    /* end Test512 */


/*************************************************************
 * Code for FNV1024 using 64-bit integers
 *************************************************************/

void Test1024 ()
{
//int            i, err;
uint8_t        eUint1024[FNV1024size];
FNV1024context  eContext;

funcName = "FNV1024";

/* test error checks */
Terr = 0;
TestR ( "init", fnvSuccess, FNV1024init (&eContext) );
    TestR ( "string", fnvNull,
            FNV1024string ( (char *)0, eUint1024 ) );
    TestR ( "string", fnvNull,
            FNV1024string ( errteststring, (uint8_t *)0 ) );
    TestR ( "block", fnvNull,
```

```
            FNV1024block ( (uint8_t *)0, 1, eUint1024 ) );
    TestR ( "block", fnvBadParam,
            FNV1024block ( errtestbytes, -1, eUint1024 ) );
    TestR ( "block", fnvNull,
            FNV1024block ( errtestbytes, 1, (uint8_t *)0 ) );
    TestR ( "init", fnvNull,
            FNV1024init ( (FNV1024context *)0 ) );
    TestR ( "initBasis", fnvNull,
            FNV1024initBasis ( (FNV1024context *)0, eUint1024 ) );
    TestR ( "blockin", fnvNull,
            FNV1024blockin ( (FNV1024context *)0,
                        errtestbytes, NTestBytes ) );
    TestR ( "blockin", fnvNull,
            FNV1024blockin ( &eContext, (uint8_t *)0,
                        NTestBytes ) );
    TestR ( "blockin", fnvBadParam,
            FNV1024blockin ( &eContext, errtestbytes, -1 ) );
    eContext.Computed = FNVclobber+FNV1024state;
    TestR ( "blockin", fnvStateError,
            FNV1024blockin ( &eContext, errtestbytes,
                        NTestBytes ) );
    TestR ( "stringin", fnvNull,
            FNV1024stringin ( (FNV1024context *)0, errteststring ) );
    TestR ( "stringin", fnvNull,
            FNV1024stringin ( &eContext, (char *)0 ) );
    TestR ( "stringin", fnvStateError,
            FNV1024stringin ( &eContext, errteststring ) );
    TestR ( "result", fnvNull,
            FNV1024result ( (FNV1024context *)0, eUint1024 ) );
    TestR ( "result", fnvNull,
            FNV1024result ( &eContext, (uint8_t *)0  ) );
    TestR ( "result", fnvStateError,
            FNV1024result ( &eContext, eUint1024  ) );
if ( Terr )
    printf ( "%s test of error checks failed %i times.0,
             funcName, Terr );
else
    printf ( "%s test of error checks passed0, funcName );

/* test actual results */
Terr = 0;
/* tbd */
}    /* end Test1024 */
   <CODE ENDS>
```

7. Security Considerations

This document is intended to provide convenient open source access by the Internet community to the FNV non-cryptographic hash. No assertion of suitability for cryptographic applications is made for the FNV hash algorithms.

7.1 Why is FNV Non-Cryptographic?

A full discussion of cryptographic hash requirements and strength is beyond the scope of this document. However, here are three characteristics of FNV that would generally be considered to make it non-cryptographic:

1. Sticky State - A cryptographic hash should not have a state in which it can stick for a plausible input pattern. But, in the very unlikely event that the FNV hash variable becomes zero and the input is a sequence of zeros, the hash variable will remain at zero until there is a non-zero input byte and the final hash value will be unaffected by the length of that sequence of zero input bytes. Of course, for the common case of fixed length input, this would usually not be significant because the number of non-zero bytes would vary inversely with the number of zero bytes and for some types of input, runs of zeros do not occur. Furthermore, the inclusion of even a little unpredictable input may be sufficient to stop an adversary from inducing a zero hash variable.

2. Diffusion - Every output bit of a cryptographic hash should be an equally complex function of every input bit. But it is easy to see that the least significant bit of a direct FNV hash is the XOR of the least significant bits of every input byte and does not depend on any other input bit. While more complex, the second through seventh least significant bits of an FNV hash have a similar weakness; only the top bit of the bottom byte of output, and higher order bits, depend on all input bits. If these properties are considered a problem, they can be easily fixed by XOR folding (see Section 3).

3. Work Factor - Depending on intended use, it is frequently desirable that a hash function should be computationally expensive for general purpose and graphics processors since these may be profusely available through elastic cloud services or botnets. This is to slow down testing of possible inputs if the output is known. But FNV is designed to be very inexpensive on a general-purpose processor. (See Appendix A.)

Nevertheless, none of the above have proven to be a problem in actual practice for the many applications of FNV.

7.2 Inducing Collisions

While use of a cryptographic hash should be considered when active adversaries are a factor, the following attack can be made much more difficult with very minor changes in the use of FNV.

If FNV is being used in a known way for hash tables in a network server or the like, for example some part of a web server, an adversary could send requests calculated to cause hash table collisions and induce substantial processing delays. As mentioned in Section 2.2, use of an offset_basis not knownable by the adversary will substantially eliminate this problem.

8. IANA Considerations

This document requires no IANA Actions.

Normative References

[RFC20] - Cerf, V., "ASCII format for network interchange", STD 80, RFC 20, October 1969, <http://www.rfc-editor.org/info/rfc20>.

Informative References

[BFDseq] - Jethanandani, M., S. Agarwal, A. Mishra, A. Sexena, A. Dekok, "Secure BFD Sequence Numbers", draft-ietf-bfd-secure-sequence-numbers-09, March 2022.

[FNV] - FNV web site: http://www.isthe.com/chongo/tech/comp/fnv/index.html

[IEEE] - http://www.ieee.org

[IPv6flow] - https://researchspace.auckland.ac.nz/bitstream/handle/2292/13240/flowhashRep.pdf

[RFC3174] - Eastlake 3rd, D. and P. Jones, "US Secure Hash Algorithm 1 (SHA1)", RFC 3174, DOI 10.17487/RFC3174, September 2001, <https://www.rfc-editor.org/info/rfc3174>.

[RFC6194] - Polk, T., Chen, L., Turner, S., and P. Hoffman, "Security Considerations for the SHA-0 and SHA-1 Message-Digest Algorithms", RFC 6194, DOI 10.17487/RFC6194, March 2011, <https://www.rfc-editor.org/info/rfc6194>.

[RFC6234] - Eastlake 3rd, D. and T. Hansen, "US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)", RFC 6234, DOI 10.17487/RFC6234, May 2011, <https://www.rfc-editor.org/info/rfc6234>.

[RFC6437] - Amante, S., Carpenter, B., Jiang, S., and J. Rajahalme, "IPv6 Flow Label Specification", RFC 6437, DOI 10.17487/RFC6437, November 2011, <https://www.rfc-editor.org/info/rfc6437>.

[RFC8200] - Deering, S. and R. Hinden, "Internet Protocol, Version 6 (IPv6) Specification", STD 86, RFC 8200, DOI 10.17487/RFC8200, July 2017, <https://www.rfc-editor.org/info/rfc8200>.

Acknowledgements

The contributions of the following are gratefully acknowledged:

Roman Donchenko, Frank Ellermann, Tony Finch, Bob Moskowitz, Gayle Noble, Stefan Santesson, and Mukund Sivaraman.

Appendix A: Work Comparison with SHA-1

This section provides a simplistic rough comparison of the level of effort required per input byte to compute FNV-1a and SHA-1 [RFC3174].

Ignoring transfer of control and conditional tests and equating all logical and arithmetic operations, FNV requires 2 operations per byte, an XOR and a multiply.

SHA-1 is a relatively weak cryptographic hash producing a 160-bit hash. It has been partially broken [RFC6194]. It is actually designed to accept a bit vector input although almost all computer uses apply it to an integer number of bytes. It processes blocks of 512 bits (64 bytes) and we estimate the effort involved in SHA-1 processing a full block. Ignoring SHA-1 initial set up, transfer of control, and conditional tests, but counting all logical and arithmetic operations, including counting indexing as an addition, SHA-1 requires 1,744 operations per 64 bytes block or 27.25 operations per byte. So by this rough measure, it is a little over 13 times the effort of FNV for large amounts of data. However, FNV is commonly used for small inputs. Using the above method, for inputs of N bytes, where N is <= 55 so SHA-1 will take one block (SHA-1 includes padding and an 8-byte length at the end of the data in the last block), the ratio of the effort for SHA-1 to the effort for FNV will be 872/N. For example, with an 8 byte input, SHA-1 will take 109 times as much effort as FNV.

Stronger cryptographic functions than SHA-1 generally have an even higher work factor.

Appendix B: Previous IETF Reference to FNV

FNV-1a was referenced in draft-ietf-tls-cached-info-08.txt that has since expired. It was later decided that it would be better to use a cryptographic hash for that application.

Below is the Java code for FNV64 from that TLS draft included by the kind permission of the author:

<CODE BEGINS>

```
 /**
 * Java code sample, implementing 64 bit FNV-1a
 * By Stefan Santesson
 */

import java.math.BigInteger;

public class FNV {


   static public BigInteger getFNV1aToByte(byte[] inp) {

       BigInteger m = new BigInteger("2").pow(64);
       BigInteger fnvPrime = new BigInteger("1099511628211");
       BigInteger fnvOffsetBasis =
               new BigInteger("14695981039346656037");

       BigInteger digest = fnvOffsetBasis;

       for (byte b : inp) {
           digest = digest.xor(BigInteger.valueOf((int) b & 255));
           digest = digest.multiply(fnvPrime).mod(m);
       }
       return digest;

   }
}
```

<CODE ENDS>

Appendix C: A Few Test Vectors

Below are a few test vectors in the form of ASCII strings and their FNV32 and FNV64 hashes using the FNV-1a algorithm.

Strings without null (zero byte) termination:

String	FNV32	FNV64
""	0x811c9dc5	0xcbf29ce484222325
"a"	0xe40c292c	0xaf63dc4c8601ec8c
"foobar"	0xbf9cf968	0x85944171f73967e8

Strings including null (zero byte) termination:

String	FNV32	FNV64
""	0x050c5d1f	0xaf63bd4c8601b7df
"a"	0x2b24d044	0x089be207b544f1e4
"foobar"	0x0c1c9eb8	0x34531ca7168b8f38

Appendix Z: Change Summary

RFC Editor Note: Please delete this appendix on publication.

From -00 to -01

1. Add Security Considerations section on why FNV is non-cryptographic.
2. Add Appendix A on a work factor comparison with SHA-1.
3. Add Appendix B concerning previous IETF draft referenced to FNV.
4. Minor editorial changes.

From -01 to -02

1. Correct FNV_Prime determination criteria and add note as to why s < 5 and s > 10 are not considered.
2. Add acknowledgements list.
3. Add a couple of references.
4. Minor editorial changes.

From -02 to -03

1. Replace direct reference to US-ASCII standard with reference to RFC 20.
2. Update dates and version number.
3. Minor editing changes.

From -03 to -04

1. Change reference to RFC 20 back to a reference to the ANSI 1968 ASCII standard.
2. Minor addition to Section 6, point 3.

3. Update dates and version number.

4. Minor editing changes.

From -04 to -05

1. Add Twitter as a use example and IPv6 flow hash study reference.

2. Update dates and version number.

From -05 to -06

1. Add code subsections.

2. Update dates and version number.

From -06 to -07 to -08

1. Update Author info.

2. Minor edits.

From -08 to -09

1. Change reference for ASCII to [RFC20].

2. Add more details on history of the string used to compute offset_basis.

3. Re-write "Work Factor" part of Section 6 to be more precise.

4. Minor editorial changes.

From -09 to -10

1. Inclusion of initial partial version of code and some documentation about the code, Section 6.

2. Insertion of new Section 4 on hashing values.

From -10 to -11

Changes based on code improvements primarily from Tony Hansen who has been added as an author. Changes based on comments from Mukund Sivaraman and Roman Donchenko.

From -11 to -12

Keep alive update.

From -12 to -13

Fixed bug in pseudocode in Section 2.3.

Change code to eliminate the BigEndian flag and so there are separate byte vector output routines for FNV32 and FNV64, equivalent to the other routines, and integer output routines for cases where Endianness consistency is not required.

From -13 to -14 to -15 to -16 to -17

Keep alive updates.
Update an author address.
Update reference.
Update author affiliation.

From -17 to -18

Add reference to draft-ietf-bfd-secure-sequence-numbers. Update author info. Minor editorial changes.

Author's Address

Glenn Fowler
Google

Email: glenn.s.fowler@gmail.com

Landon Curt Noll
Cisco Systems
170 West Tasman Drive
San Jose, CA 95134 USA

Telephone: +1-408-424-1102
Email: fnv-ietf5-mail@asthe.com
URL: http://www.isthe.com/chongo/index.html

Kiem-Phong Vo
Google

Email: phongvo@gmail.com

Donald Eastlake
Futurewei Technologies
2386 Panoramic Circle
Apopka, FL 32703 USA

Telephone: +1-508-333-2270
EMail: d3e3e3@gmail.com

Tony Hansen
AT&T Laboratories
200 Laurel Ave. South
Middletown, NJ 07748
USA

Email: tony@att.com