Workgroup: Internet Engineering Task Force
Internet-Draft: draft-eastlake-fnv-22
Published: 31 March 2024
Intended Status: Informational
Expires: 2 October 2024
Authors: G. Fowler L. Noll K. Vo D. Eastlake
Google Cisco Systems Google Independent
T. Hansen
AT&T Laboratories

The FNV Non-Cryptographic Hash Algorithm

Abstract

FNV (Fowler/Noll/Vo) is a fast, non-cryptographic hash algorithm with good dispersion that is referenced in a number of standards documents and widely used. 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 Memo

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

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

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

This Internet-Draft will expire on 2 October 2024.

Trust Legal Provisions and are provided without warranty as described in the Revised BSD License.

Table of Contents

1. Introduction

The FNV hash algorithm originated from an idea submitted as reviewer comments to the [IEEE] POSIX P1003.2 committee in 1991 by Glenn Fowler and Phong Vo. Subsequently, during a ballot round, Landon Curt Noll proposed an enhancement to 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. Their exceptional dispersion makes them particularly well-suited for hashing nearly identical strings, including URLs, hostnames, filenames, text, and IP addresses. Their speed allows one to quickly hash lots of data while maintaining a reasonably low collision rate. They are generally not suitable for cryptographic use (see Section 7.1).

The FNV hash is widely used. For example it is referenced in the [RFC7357], [RFC7873], and [IEEE8021Qbp] standards documents. It is also used in DNS servers, the X (formerly 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]. Additionally, there was 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 took deliberate steps to disclose the algorithm in a public forum soon after it's invention. 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.

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, 128, 256, 512, or 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 >= 32 and is 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

  $$(p \bmod (2^{40} - 2^{24} - 1)) > (2^{24} + 2^8 + 2^7)$$

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 due to the resulting low hash quality. 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. The selection of FNV_Primes prioritizes the quality of the resulting hash function, not compiler optimization considerations.

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 [RFC0020]:

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 as discussed in Section 7.2.

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, as long as it is used consistantly, 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 for most lengths. Because they are more efficient, the code returns FNV hashes of 32-bit or 64-bit size as integers, on computers supporting integers of those sides. 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.

For scenarios where a fixed-size binary field of k bits is desired with k < 1024 but not among the provided constants in Section 5, the recommended approach involves using the smallest FNV hash of size S where S > k and employing 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 in a fixed order 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:

Size FNV Prime = Exression
= Decimal
= Hexadecimal
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 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 0000010000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000 000000000000018D

Table 1

The FNV offset_basis values are as follows:

Size offset_basis
= Decimal
= Hexadecimal
32 bit offset_basis
= 2,166,136,261
= 0x811C9DC5

Size offset_basis
= Decimal
= Hexadecimal
64 bit offset_basis
= 14,695,981,039,346,656,037
= 0xCBF29CE4 84222325
128 bit offset_basis
= 144,066,263,297,769,815,596,495,629,667,062,367,629
= 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

Table 2

6. The Source Code

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

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

Section 6.2 provides the test driver.

[[[Code could use some additional testing for Big Endian operation.]]]

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. The xxx in the function names is "32", "64", "128", "256", "512", or "1024" depending on the length of the FNV. The FNV-32 functions return a 32 bit integer. The FNV-64 bit functions return a 64-bit integer or an 8 byte vector depending on whether 64-bit integers are or are not supports. All longer FNV functions return a byte vector. Versions returning an integer may NOT be compatible between systems of different endian-ness.

If you want to copy the source code from this document, note that it is indented by three spaces in the ".txt" version. It may be simplest to copy from the ".html" version of this document.

FNVxxxstring, FNVxxxblock: These are simple functions for directly returning the FNV hash of a zero terminated byte string NOT including that zero byte 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: 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 FNVxxxcontext that holds the current state of the hash. FNVxxxinit initializes that context to the standard offset_basis. FNVxxxinitBasis 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: These functions hash a sequence of bytes into an FNVxxxcontext that was originally initialized by FNVxxxinit or FNVxxxinitBasis. FNVxxxblockin hashes in a counted block of bytes. FNVxxxstringin hashes in a zero terminated byte string not incuding the final zero byte.

FNVxxxresult: This function extracts the final FNV hash result from an FNVxxxcontext.

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> file "fnv-private.h"

//********************** fnv-private.h ***********************//
//****************** See RFC NNNN for details ******************//
/* Copyright (c) 2016, 2023, 2024 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,
   FNV64state = 3,
   FNV128state = 5,
   FNV256state = 7,
   FNV512state = 11,
   FNV1024state = 13
};

#endif

<CODE ENDS>
```

The following code is a simple header file to define the return value error codes for the FNV routines.

```
<CODE BEGINS> file "FNVErrorCodes.h"

//********************** FNVErrorCodes.h ************************//
//***************** See RFC NNNN for details. *******************//
/*
 * Copyright (c) 2016, 2023, 2024 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_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_ */

<CODE ENDS>
```

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

<CODE BEGINS> file "FNVconfig.h"

```
//************************ FNVconfig.h ************************//
//****************** See RFC NNNN for details. ********************//
/*
 * Copyright (c) 2016, 2024 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

/*      FNV_BigEndian - Define this ONLY if your system uses big
 *          endian representation AND your FNV hashes need to
 *          interoperate with little endian systems. If you #define
 *          this symbol when not needed, it will unnecessarily slow
 *          down and increase the code size of the FNV functions.
 */
// #define FNV_BigEndian

/*      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 */
#  undef uint64_t
#  define uint64_t no_64_bit_integers
# endif
# ifdef TEST_FNV_BigEndian
#  ifndef FNV_BigEndian
#   define FNV_BigEndian
#  endif
# else
#  undef FNV_BigEndian
# 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> file "FNV32.h"

```
//************************** FNV32.h ****************************//
//****************** See RFC NNNN for details ********************//
/*
 * Copyright (c) 2016, 2024 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)
 */

#include "FNVErrorCodes.h"

/*
 *  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 unsigned 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> file "FNV32.c"

```
//************************ FNV32.c *************************//
//**************** See RFC NNNN for details. ******************//
/* Copyright (c) 2016, 2024 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.
 */

#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

#ifdef FNV_BigEndian
/* Local prototype */
static void FNV32reverse ( uint32_t *out, uint32_t hash );
#endif

/* 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 )
        return fnvNull; /* Null input pointer */
    temp = FNV32basis;
    while ( (ch = *in++) )
        temp = FNV32prime * ( temp ^ ch );
#ifdef FNV_BigEndian
    FNV32reverse ( out, temp );
#else
    *out = temp;
#endif
    return fnvSuccess;
}   /* 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 )
        return fnvNull; /* Null input pointer */
    if ( length < 0 )
        return fnvBadParam;
    for ( temp = FNV32basis; length > 0; length-- )
        temp = FNV32prime * ( temp ^ *in++ );
#ifdef FNV_BigEndian
    FNV32reverse ( out, temp );
#else
    *out = temp;
#endif
    return fnvSuccess;
}   /* end FNV32block */

#ifdef FNV_BigEndian

/* Store a Big Endian result back as Little Endian
 ************************************************************/
static void FNV32reverse ( uint32_t *out, uint32_t hash ) {
    uint32_t    temp;

    temp = hash & 0xFF;
    hash >>= 8;
    temp = ( temp << 8 ) + ( hash & 0xFF );
    hash >>= 8;
    temp = ( temp << 8 ) + ( hash & 0xFF );
    hash >>= 8;
    *out = ( temp << 8 ) + ( hash & 0xFF );
}   /* end FNV32reverse */

#endif /* FNV_BigEndian */


//************************************************************
//       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 )
        return fnvNull;
    ctx->Hash = basis;
    ctx->Computed = FNVinited+FNV32state;
    return fnvSuccess;
}   /* 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 )
        return fnvNull;
    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;
}   /* 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 )
        return fnvNull;
    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;
}   /* end FNV32stringin */

/* return hash
 *************************************************************/
int FNV32result ( FNV32context * const ctx,
                  uint32_t * const out )
{
    if ( !ctx || !out )
        return fnvNull;
    if ( ctx->Computed != FNVcomputed+FNV32state )
        return fnvStateError;
    ctx->Computed = FNVemptied+FNV32state;
#ifdef FNV_BigEndian
    FNV32reverse ( out, ctx->Hash );
#else
    *out = ctx->Hash;
#endif
    ctx->Hash = 0;
    return fnvSuccess;
}   /* end FNV32result */

#endif    /* _FNV32_C_ */

<CODE ENDS>
```

6.1.2. FNV64 Code

The header and C source for 64-bit FNV-1a. Return a 64-bit integers are supported, otherwise return an 8-byte vectir if bytes.

<CODE BEGINS> file "FNV64.h"

```
//************************** FNV64.h ***************************//
//***************** See RFC NNNN for details. ******************//
/*
 * Copyright (c) 2016, 2024 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)
 */

#include "FNVErrorCodes.h"

/*
 *  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 unsigned 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> file "FNV64.c"

```
//************************** FNV64.c ***************************//
//****************** See RFC NNNN for details ******************//
/* Copyright (c) 2016, 2024 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 <stdio.h>

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


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

#ifdef FNV_BigEndian
/* Local prototype */
static void FNV64reverse ( uint64_t * out, uint64_t hash );
#endif /* FNV_BigEndian */

/* 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 )
        return fnvNull; /* Null input pointer */

    temp = FNV64basis;
    while ( (ch = *in++) )
```

```
        temp = FNV64prime * ( temp ^ ch );
#ifdef FNV_BigEndian
    FNV64reverse ( out, temp );
#else
    *out = temp;
#endif
    return fnvSuccess;
}   /* 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 )
        return fnvNull; /* Null input/out pointer */

    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;
}   /* 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 )
        return fnvNull;

    ctx->Hash = basis;
    ctx->Computed = FNVinited+FNV64state;
    return fnvSuccess;
}       /* 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 )
        return fnvNull;

    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;
}   /* 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 )
        return fnvNull;

    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;
}   /* end FNV64stringin */

/* return hash  (64 bit)
 ****************************************************************/
int FNV64result ( FNV64context * const ctx,
                  uint64_t * const out )
{
    if ( !ctx || !out )
        return fnvNull;

    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;
}   /* 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 not including the zero
 *************************************************************/
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
 *************************************************************/
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 )
        return fnvNull;

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

/* initialize context  (32 bit)
 *************************************************************/
int FNV64initBasis ( FNV64context * const ctx,
                     const uint8_t basis[FNV64size] )
{
    if ( !ctx )
        return fnvNull;

#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;
}   /* 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 )
        return fnvNull;
    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++ ); */
        temp[3] ^= *in++;
        temp2[1] = temp[3] << FNV64shift;
        temp2[0] = temp[2] << FNV64shift;
        temp[3] *= FNV64primeX;
        temp[2] *= FNV64primeX;
        temp[1] *= FNV64primeX;
        temp[0] *= FNV64primeX;
        temp[1] += temp2[1];
        temp[0] += temp2[0];
        temp[2] += temp[3] >> 16;
        temp[1] += temp[2] >> 16;
        temp[0] += temp[1] >> 16;
        temp[3] &= 0xFFFF;
        temp[2] &= 0xFFFF;
        temp[1] &= 0xFFFF;
        }
    for ( i=0; i<FNV64size/2; ++i )
        ctx->Hash[i] = temp[i];
    return fnvSuccess;
}   /* 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 )
        return fnvNull;
    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++ ) ) {
        /* temp = FNV64prime * ( temp ^ ch ); */
        temp[3] ^= ch;
        temp2[1] = temp[3] << FNV64shift;
        temp2[0] = temp[2] << FNV64shift;
        temp[3] *= FNV64primeX;
        temp[2] *= FNV64primeX;
        temp[1] *= FNV64primeX;
        temp[0] *= FNV64primeX;
        temp[1] += temp2[1];
        temp[0] += temp2[0];
        temp[2] += temp[3] >> 16;
        temp[1] += temp[2] >> 16;
        temp[0] += temp[1] >> 16;
        temp[3] &= 0xFFFF;
        temp[2] &= 0xFFFF;
        temp[1] &= 0xFFFF;
        }
    for ( i=0; i<FNV64size/2; ++i )
        ctx->Hash[i] = temp[i];
    return fnvSuccess;
}   /* end FNV64stringin */

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

    if ( !ctx || !out )
```

```
        return fnvNull;

    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] >> 8;
        out[2*i+1] = ctx->Hash[i];
#endif
        ctx -> Hash[i] = 0;
        }
    ctx->Computed = FNVemptied+FNV64state;
    return fnvSuccess;
}   /* end FNV64result */

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

#endif    /* _FNV64_C_ */

<CODE ENDS>
```

6.1.3. FNV128 Code

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

<CODE BEGINS> file "FNV128.h"

```
//************************** FNV128.h ************************//
//*************** See RFC NNNN for details. ******************//
/*
 * Copyright (c) 2016, 2024 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)
 */

#include "FNVErrorCodes.h"

/*
 *  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 unsigned 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> file "FNV128.c"

```
//***************************** FNV128.c ************************//
//******************* See RFC NNNN for details ******************//
/* Copyright (c) 2016, 2024 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 "FNVconfig.h"
#include "fnv-private.h"
#include "FNV128.h"

//*****************************************************************
// COMMON CODE FOR 64 AND 32 BIT INTEGER MODES
//*****************************************************************

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

    if ( (err = FNV128init ( &ctx )) != fnvSuccess )
        return err;
    if ( (err = FNV128stringin ( &ctx, in )) != 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;

   if ( (err = FNV128init ( &ctx )) != fnvSuccess )
       return err;
   if ( (err = FNV128blockin ( &ctx, in, length )) != 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 )
        return fnvNull;

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

#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;
}   /* 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 )
        return fnvNull;
    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++ ); */
        temp[FNV128size/4-1] ^= *in++;
        temp2[1] = temp[3] << FNV128shift;
        temp2[0] = temp[2] << FNV128shift;
```

```
        for ( i=0; i < FNV128size/4; ++i )
            temp[i] *= FNV128primeX;
        temp[1] += temp2[1];
        temp[0] += temp2[0];
        temp[2] += temp[3] >> 32;
        temp[1] += temp[2] >> 32;
        temp[0] += temp[1] >> 32;
        temp[3] &= 0xFFFFFFFF;
        temp[2] &= 0xFFFFFFFF;
        temp[1] &= 0xFFFFFFFF;
        }
    for ( i=0; i<FNV128size/4; ++i )
        ctx->Hash[i] = (uint32_t)temp[i];
    return fnvSuccess;
}   /* end FNV128blockin */

/* 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 )
        return fnvNull;
    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 ); */
        temp[3] ^= ch;
        temp2[1] = temp[3] << FNV128shift;
        temp2[0] = temp[2] << FNV128shift;
        for ( i=0; i < FNV128size/4; ++i )
            temp[i] *= FNV128primeX;
        temp[1] += temp2[1];
        temp[0] += temp2[0];
        temp[2] += temp[3] >> 32;
        temp[1] += temp[2] >> 32;
        temp[0] += temp[1] >> 32;
        temp[3] &= 0xFFFFFFFF;
```

```
        temp[2] &= 0xFFFFFFFF;
        temp[1] &= 0xFFFFFFFF;
        }
    for ( i=0; i<FNV128size/4; ++i )
        ctx->Hash[i] = (uint32_t)temp[i];
    return fnvSuccess;
}   /* end FNV128stringin */

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

    if ( !ctx || !out )
        return fnvNull;
    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] >> 24;
        out[4*i+1] = ctx->Hash[i] >> 16;
        out[4*i+2] = ctx->Hash[i] >> 8;
        out[4*i+3] = ctx->Hash[i];
#endif
        ctx -> Hash[i] = 0;
        }
    ctx->Computed = FNVemptied+FNV128state;
    return fnvSuccess;
}   /* 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 )
        return fnvNull;
    for ( i=0; i<FNV128size/2; ++i )
        ctx->Hash[i] = FNV128basis[i];
    ctx->Computed = FNVinited+FNV128state;
    return fnvSuccess;
}   /* 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 )
        return fnvNull;
#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;
```

```
}   /* 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 )
        return fnvNull;
    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[FNV128size/2-1] ^= *in++;
        temp2[2] = temp[7] << FNV128shift;
        temp2[1] = temp[6] << FNV128shift;
        temp2[0] = temp[5] << FNV128shift;
        for ( i=0; i < (FNV128size/2); ++i )
            temp[i] *= FNV128primeX;
        temp[2] += temp2[2];
        temp[1] += temp2[1];
        temp[0] += temp2[0];
        for ( i=FNV128size/2-1; 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;
}   /* 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 )
        return fnvNull;
    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[FNV128size/2-1] ^= ch;
        temp2[2] = temp[7] << FNV128shift;
        temp2[1] = temp[6] << FNV128shift;
        temp2[0] = temp[5] << FNV128shift;
        for ( i=0; i<(FNV128size/2); ++i )
            temp[i] *= FNV128primeX;
        temp[2] += temp2[2];
        temp[1] += temp2[1];
        temp[0] += temp2[0];
        for ( i=FNV128size/2-1; 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;
}   /* end FNV128stringin */

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

    if ( !ctx || !out )
        return fnvNull;
    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] >> 8;
        out[2*i+1] = ctx->Hash[i];
#endif
        ctx -> Hash[i] = 0;
        }
    ctx->Computed = FNVemptied+FNV128state;
    return fnvSuccess;
}   /* end FNV128result */

#endif    /*  FNV_64bitIntegers */
//*****************************************************************
//        END VERSION FOR WHEN YOU ONLY HAVE 32-BIT ARITHMETIC
//*****************************************************************
#endif    /* _FNV128_C_ */

<CODE ENDS>
```

6.1.4. FNV256 Code

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

```
<CODE BEGINS> file "FNV256.h"


//************************ FNV256.h ***********************//
//************** See RFC NNNN for details. *****************//
/*
 * Copyright (c) 2016, 2024 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)
 */

#include "FNVErrorCodes.h"

/*
 *  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 unsigned 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> file "FNV256.c"

```
//***************************** FNV256.c ************************//
//******************* See RFC NNNN for details ******************//
/* Copyright (c) 2016, 2024 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

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 )
        return fnvNull;
    for ( i=0; i<FNV256size/4; ++i )
        ctx->Hash[i] = FNV256basis[i];
    ctx->Computed = FNVinited+FNV256state;
    return fnvSuccess;
}   /* 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 )
        return fnvNull;
#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;
}   /* 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 )
        return fnvNull;
    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[FNV256size/4-1] ^= *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] >> 32;
            temp[i] &= 0xFFFFFFFF;
            }
       }
    for ( i=0; i<FNV256size/4; ++i )
        ctx->Hash[i] = (uint32_t)temp[i];
    return fnvSuccess;
}   /* 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 )
        return fnvNull;
    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[FNV256size/4-1] ^= 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] >> 32;
            temp[i] &= 0xFFFFFFFF;
            }
       }
    for ( i=0; i<FNV256size/4; ++i )
```

```
        ctx->Hash[i] = (uint32_t)temp[i];
    return fnvSuccess;
}   /* end FNV256stringin */

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

    if ( !ctx || !out )
        return fnvNull;
    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] >> 24;
        out[4*i+1] = ctx->Hash[i] >> 16;
        out[4*i+2] = ctx->Hash[i] >> 8;
        out[4*i+3] = ctx->Hash[i];
#endif
        ctx -> Hash[i] = 0;
        }
    ctx->Computed = FNVemptied+FNV256state;
    return fnvSuccess;
}   /* 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 )
        return fnvNull;
    for ( i=0; i<FNV256size/2; ++i )
        ctx->Hash[i] = FNV256basis[i];
    ctx->Computed = FNVinited+FNV256state;
    return fnvSuccess;
}   /* 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 )
        return fnvNull;
#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;
}   /* 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 )
        return fnvNull;
    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[FNV256size/2-1] ^= *in++;
        for ( i=0; i<6; ++i )
            temp2[5-i] = temp[FNV256size/2-1-i] << FNV256shift;
        for ( i=0; i<FNV256size/2; ++i )
            temp[i] *= FNV256primeX;
        for ( i=0; i<6; ++i )
            temp[i] += temp2[i];
        for ( i=FNV256size/2-1; 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;
}   /* 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 )
        return fnvNull;
    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++ ) ) {
        /* temp = FNV256prime * ( temp ^ *in++ ); */
        temp[FNV256size/2-1] ^= ch;
        for ( i=0; i<6; ++i )
            temp2[5-i] = temp[FNV256size/2-1-i] << FNV256shift;
        for ( i=0; i<FNV256size/2; ++i )
            temp[i] *= FNV256primeX;
        for ( i=0; i<6; ++i )
            temp[i] += temp2[i];
        for ( i=FNV256size/2-1; 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;
}   /* end FNV256stringin */

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

    if ( !ctx || !out )
        return fnvNull;
    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] >> 8;
        out[2*i+1] = ctx->Hash[i];
#endif
        ctx->Hash[i] = 0;
        }
    ctx->Computed = FNVemptied+FNV256state;
    return fnvSuccess;
}   /* end FNV256result */

#endif    /*  FNV_64bitIntegers */
//*************************************************************
//        END VERSION FOR WHEN YOU ONLY HAVE 32-BIT ARITHMETIC
//*************************************************************
#endif    /* _FNV256_C_ */

<CODE ENDS>
```

6.1.5. FNV512 Code

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

<CODE BEGINS> file "FNV512.h"

```
//************************ FNV512.h ***********************//
//************** See RFC NNNN for details. *****************//
/*
 * Copyright (c) 2016, 2024 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)
 */

#include "FNVErrorCodes.h"

/*
 *  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 unsigned 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> file "FNV512.c"

```
//***************************** FNV512.c ************************//
//******************* See RFC NNNN for details ******************/
/* Copyright (c) 2016, 2024 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, 0x309990AC,
     0xAC87D059, 0xC9000000, 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 ) {
    int i;

    if ( !ctx )
        return fnvNull;
    for ( i=0; i<FNV512size/4; ++i )
        ctx->Hash[i] = FNV512basis[i];
    ctx->Computed = FNVinited+FNV512state;
    return fnvSuccess;
}   /* 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 )
        return fnvNull;
#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;
}   /* 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[6];
    int         i;

    if ( !ctx || !in )
        return fnvNull;
    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/4; ++i )
```

```
         temp[i] = ctx->Hash[i];  // copy into temp
    for ( ; length > 0; length-- ) {
        /* temp = FNV512prime * ( temp ^ *in++ ); */
        temp[FNV512size/4-1] ^= *in++;
        for ( i=0; i<6; ++i )
            temp2[5-i] = temp[FNV512size/4-1-i] << FNV512shift;
        for ( i=0; i<FNV512size/4; ++i )
            temp[i] *= FNV512primeX;
        for ( i=0; i<6; ++i )
            temp[i] += temp2[i];
        for ( i=FNV512size/4-1; i>0; --i ) {
            temp[i-1] += temp[i] >> 32; // propagate carries
            temp[i] &= 0xFFFFFFFF;
            }
       }   /* end for length */
    for ( i=0; i<FNV512size/4; ++i )
        ctx->Hash[i] = (uint32_t)temp[i];  // store back into hash
    return fnvSuccess;
}   /* end FNV512blockin */

/* hash in a string  (64 bit)
 ***************************************************************/
int FNV512stringin ( FNV512context *ctx, const char *in ) {
    uint64_t   temp[FNV512size/4];
    uint64_t   temp2[6];
    int        i;
    uint8_t    ch;

    if ( !ctx || !in )
        return fnvNull;
    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];  // copy into temp
    while ( (ch = (uint8_t)*in++) ) {
        /* temp = FNV512prime * ( temp ^ ch ); */
        temp[FNV512size/4-1] ^= ch;
        for ( i=0; i<6; ++i )
            temp2[5-i] = temp[FNV512size/4-1-i] << FNV512shift;
        for ( i=0; i<FNV512size/4; ++i )
            temp[i] *= FNV512primeX;
        for ( i=0; i<6; ++i )
            temp[i] += temp2[i];
```

```
        for ( i=FNV512size/4-1; i>0; --i ) {
            temp[i-1] += temp[i] >> 32; // propagate carries
            temp[i] &= 0xFFFFFFFF;
            }
       }
    for ( i=0; i<FNV512size/4; ++i )
        ctx->Hash[i] = (uint32_t)temp[i];  // store back into hash
    return fnvSuccess;
}   /* end FNV512stringin */

/* return hash  (64 bit)
 ****************************************************************/
int FNV512result ( FNV512context *ctx, uint8_t out[FNV512size] ) {
    int i;

    if ( !ctx || !out )
        return fnvNull;
    if ( ctx->Computed != FNVcomputed+FNV512state )
        return fnvStateError;
    for ( i=0; i<FNV512size/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] >> 24;
        out[4*i+1] = ctx->Hash[i] >> 16;
        out[4*i+2] = ctx->Hash[i] >> 8;
        out[4*i+3] = ctx->Hash[i];
#endif
        ctx -> Hash[i] = 0;
        }
    ctx->Computed = FNVemptied+FNV512state;
    return fnvSuccess;
}   /* end FNV512result */

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

/*
  512 bit FNV_prime = 2^344 + 2^8 + 0x57 =
   0x00000000 00000000 00000000 00000000
     00000000 01000000 00000000 00000000
```

```
     00000000 00000000 00000000 00000000
     00000000 00000000 00000000 00000157 */
#define FNV512primeX 0x0157
#define FNV512shift 8

/* 0xB86DB0B1 171F4416 DCA1E50F 309990AC
     AC87D059 C9000000 00000000 00000D21
     E948F68A 34C192F6 2EA79BC9 42DBE7CE
     18203641 5F56E34B AC982AAC 4AFE9FD9 */

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 )
        return fnvNull;
    for ( i=0; i<FNV512size/2; ++i )
        ctx->Hash[i] = FNV512basis[i];
    ctx->Computed = FNVinited+FNV512state;
    return fnvSuccess;
}   /* 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 )
        return fnvNull;
#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;
}   /* 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[11];
    int        i;

    if ( !ctx || !in )
        return fnvNull;
    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];  // copy into temp
    for ( ; length > 0; length-- ) {
        /* temp = FNV512prime * ( temp ^ *in++ ); */
        temp[FNV512size/2-1] ^= *in++;
        for ( i=0; i<11; ++i )
            temp2[10-i] = temp[FNV512size/2-1-i] << FNV512shift;
        for ( i=0; i<FNV512size/2; ++i )
            temp[i] *= FNV512primeX;
        for ( i=0; i<11; ++i )
            temp[i] += temp2[i];
        for ( i=FNV512size/2-1; i>0; --i ) {
            temp[i-1] += temp[i] >> 16; // propagate carries
            temp[i] &= 0xFFFF;
```

```
            }
        }   /* end for length */
    for ( i=0; i<FNV512size/2; ++i )
        ctx->Hash[i] = (uint16_t)temp[i];  // store back into hash
    return fnvSuccess;
}   /* end FNV512blockin */

/* hash in a string  (32 bit)
 ***************************************************************/
int FNV512stringin ( FNV512context *ctx, const char *in ) {
    uint32_t   temp[FNV512size/2];
    uint32_t   temp2[11];
    int        i;
    uint8_t    ch;

    if ( !ctx || !in )
        return fnvNull;
    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];  // copy into temp
    while ( (ch = (uint8_t)*in++) ) {
        /* temp = FNV512prime * ( temp ^ *in++ ); */
        temp[FNV512size/2-1] ^= ch;
        for ( i=0; i<11; ++i )
            temp2[10-i] = temp[FNV512size/2-1-i] << FNV512shift;
        for ( i=0; i<FNV512size/2; ++i )
            temp[i] *= FNV512primeX;
        for ( i=0; i<11; ++i )
            temp[i] += temp2[i];
        for ( i=FNV512size/2-1; i>0; --i ) {
            temp[i-1] += temp[i] >> 16; // propagate carries
            temp[i] &= 0xFFFF;
            }
         }
    for ( i=0; i<FNV512size/2; ++i )
        ctx->Hash[i] = temp[i];  // store back into hash
    return fnvSuccess;
}   /* end FNV512stringin */

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

```
    int    i;

    if ( !ctx || !out )
        return fnvNull;
    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] >> 8;
        out[2*i+1] = ctx->Hash[i];
#endif
        ctx->Hash[i] = 0;
        }
    ctx->Computed = FNVemptied+FNV512state;
    return fnvSuccess;
}   /* end FNV512result */

#endif    /*  FNV_64bitIntegers */
//****************************************************************
//        END VERSION FOR WHEN YOU ONLY HAVE 32-BIT ARITHMETIC
//****************************************************************
#endif    /* _FNV512_C_ */

<CODE ENDS>
```

6.1.6. FNV1024 Code

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

<CODE BEGINS> file "FNV1024.h"

```
//************************ FNV1024.h ************************//
//*************** See RFC NNNN for details. *****************//
/*
 * Copyright (c) 2016, 2024 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)
 */

#include "FNVErrorCodes.h"

/*
 *  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 unsigned 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> file "FNV1024.c"

```
//***************************** FNV1024.c ************************//
//******************* See RFC NNNN for details *******************//
/* Copyright (c) 2016, 2024 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 0000000000000000
     0000000000000000 0000000000000000
     0000000000000000 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 */

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 ) {
    int i;
```

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

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

    if ( !ctx )
        return fnvNull;
#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;
}   /* 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[11];
    int         i;
```

```
    if ( !ctx || !in )
        return fnvNull;
    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/4; ++i )
         temp[i] = ctx->Hash[i];  // copy into temp
    for ( ; length > 0; length-- ) {
        /* temp = FNV1024prime * ( temp ^ *in++ ); */
        temp[FNV1024size/4-1] ^= *in++;
        for ( i=0; i<11; ++i )
            temp2[10-i] = temp[FNV1024size/4-1-i] << FNV1024shift;
        for ( i=0; i<FNV1024size/4; ++i )
            temp[i] *= FNV1024primeX;
        for ( i=0; i<11; ++i )
            temp[i] += temp2[i];
        for ( i=FNV1024size/4-1; i>0; --i ) {
            temp[i-1] += temp[i] >> 32;  // propagate carries
            temp[i] &= 0xFFFFFFFF;
            }
       }
    for ( i=0; i<FNV1024size/4; ++i )
        ctx->Hash[i] = (uint32_t)temp[i];  // store back into hash
    return fnvSuccess;
}   /* end FNV1024input */

/* hash in a string  (64 bit)
 ***************************************************************/
int FNV1024stringin ( FNV1024context *ctx, const char *in ) {
    uint64_t   temp[FNV1024size/4];
    uint64_t   temp2[11];
    int        i;
    uint8_t    ch;

    if ( !ctx || !in )
        return fnvNull;
    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];  // copy into temp
    while ( (ch = (uint8_t)*in++) ) {
        /* temp = FNV1024prime * ( temp ^ ch ); */
        temp[FNV1024size/4-1] ^= ch;
        for ( i=0; i<11; ++i )
            temp2[10-i] = temp[FNV1024size/4-1-i] << FNV1024shift;
        for ( i=0; i<FNV1024size/4; ++i )
            temp[i] *= FNV1024primeX;
        for ( i=0; i<11; ++i )
                temp[i] += temp2[i];
        for ( i=FNV1024size/4-1; i>0; --i ) {
            temp[i-1] += temp[i] >> 32;
            temp[i] &= 0xFFFFFFFF;
            }
       }
    for ( i=0; i<FNV1024size/4; ++i )
        ctx->Hash[i] = (uint32_t)temp[i];  // store back into hash
    return fnvSuccess;
}   /* end FNV1024stringin */

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

    if ( !ctx || !out )
        return fnvNull;
    if ( ctx->Computed != FNVcomputed+FNV1024state )
        return fnvStateError;
    for ( i=0; i<FNV1024size/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] >> 24;
        out[4*i+1] = ctx->Hash[i] >> 16;
        out[4*i+2] = ctx->Hash[i] >> 8;
        out[4*i+3] = ctx->Hash[i];
#endif
        ctx -> Hash[i] = 0;
        }
    ctx->Computed = FNVemptied+FNV1024state;
    return fnvSuccess;
```

```
}   /* 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 =
   0x00000000 00000000 00000000 00000000
     00000000 00000000 00000000 00000000
     00000000 00000000 00000100 00000000
     00000000 00000000 00000000 00000000
     00000000 00000000 00000000 00000000
     00000000 00000000 00000000 00000000
     00000000 00000000 00000000 00000000
     00000000 00000000 00000000 0000018D */
#define FNV1024primeX 0x018D
#define FNV1024shift 8

/* 0x00000000 00000000 005F7A767 58ECC4D
     32E56D5A 591028B7 4B29FC42 23FDADA1
     6C3BF34E DA3674DA 9A21D900 00000000
     00000000 00000000 00000000 00000000
     00000000 00000000 00000000 00000000
     00000000 00000000 00000000 0004C6D7
     EB6E7380 2734510A 555F256C C005AE55
     6BDE8CC9 C6A93B21 AFF4B16C 71EE90B3 */

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 )
        return fnvNull;
    for ( i=0; i<FNV1024size/2; ++i )
        ctx->Hash[i] = FNV1024basis[i];
    ctx->Computed = FNVinited+FNV1024state;
    return fnvSuccess;
}   /* 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 )
        return fnvNull;
#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;
}   /* 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[22];
    int        i;

    if ( !ctx || !in )
        return fnvNull;
    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];  // copy into temp
    for ( ; length > 0; length-- ) {
        /* temp = FNV1024prime * ( temp ^ *in++ ); */
        temp[FNV1024size/2-1] ^= *in++;
        for ( i=0; i<22; ++i )
            temp2[21-i] = temp[FNV1024size/2-1-i] << FNV1024shift;
        for ( i=0; i<FNV1024size/2; ++i )
            temp[i] *= FNV1024primeX;
        for ( i=0; i<22; ++i )
            temp[i] += temp2[i];
        for ( i=FNV1024size/2-1; i>0; --i ) {
            temp[i-1] += temp[i] >> 16; // propagate carries
            temp[i] &= 0xFFFF;
            }
        }
    for ( i=0; i<FNV1024size/2; ++i )
        ctx->Hash[i] = temp[i];  // store back into hash
    return fnvSuccess;
}   /* end FNV1024blockin */

/* hash in a string  (32 bit)
 ***************************************************************/
int FNV1024stringin ( FNV1024context *ctx, const char *in )
{
    uint32_t   temp[FNV1024size/2];
    uint32_t   temp2[22];
    int        i;
    uint8_t    ch;

    if ( !ctx || !in )
        return fnvNull;
    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];  // copy into temp
    while ( (ch = (uint8_t)*in++) ) {
        /* temp = FNV1024prime * ( temp ^ *in++ ); */
        temp[FNV1024size/2-1] ^= ch;
        for ( i=0; i<22; ++i )
            temp2[21-i] = temp[FNV1024size/2-1-i] << FNV1024shift;
        for ( i=0; i<FNV1024size/2; ++i )
            temp[i] *= FNV1024primeX;
        for ( i=0; i<22; ++i )
             temp[i] += temp2[i];
        for ( i=FNV1024size/2-1; i>0; --i ) {
            temp[i-1] += temp[i] >> 16; // propagate carries
            temp[i] &= 0xFFFF;
            }
         }
    for ( i=0; i<FNV1024size/2; ++i )
        ctx->Hash[i] = temp[i];  // store back into hash
    return fnvSuccess;
}   /* end FNV1024stringin */

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

    if ( !ctx || !out )
        return fnvNull;
    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] >> 8;
        out[2*i+1] = ctx->Hash[i];
#endif
        ctx->Hash[i] = 0;
        }
    ctx->Computed = FNVemptied+FNV1024state;
    return fnvSuccess;
```

```
}   /* 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 with a command line interface:

<CODE BEGINS> file "main.c"

```
//************************ MAIN.c **************************//
//*************** See RFC NNNN for details. *****************//
/*
 * Copyright (c) 2016, 2024 IETF Trust and the persons
 * identified as authors of the code.  All rights reserved.
 * See fnv-private.h for terms of use and redistribution.
 */

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

#include <unistd.h>
extern char *optarg;
extern int optind, opterr, optopt;

/* To do a thorough test you need to run will all four
 * combinations of the following defined/undefined:
 * FNV_64bitIntegers FNV_BigEndian
 */
#include "FNVconfig.h"

#include "fnv-private.h"
#include "FNV32.h"
#include "FNV64.h"
#include "FNV128.h"
#include "FNV256.h"
#include "FNV512.h"
#include "FNV1024.h"

/* global variables */
char    *funcName = "not set?";
char    *errteststring = "foo";
int     Terr = -1; /* Total errors */
int     verbose = 0; /* verbose flag */
#define NTestBytes 3
uint8_t errtestbytes[NTestBytes] = { (uint8_t)1,
    (uint8_t)2, (uint8_t)3 };
char    *teststring[] = {
        "",
        "a",
        "foobar",
        "Hello!\x01\xFF\xED"
};
#define NTstrings (sizeof(teststring)/sizeof(char *))

FNV32context    e32Context;
```

```
uint32_t        eUint32 = 42;
#ifdef FNV_64bitIntegers
uint64_t        eUint64 = 42;
#else
uint8_t         Vec64[FNV64size];
#endif
FNV64context    e64Context;
uint8_t         Vec128[FNV128size];
FNV128context   e128Context;
uint8_t         Vec256[FNV256size];
FNV256context   e256Context;
uint8_t         Vec512[FNV512size];
FNV512context   e512Context;
uint8_t         Vec1024[FNV1024size];
FNV1024context  e1024Context;


//***************************************************************
// local prototypes in alphabetic order
//***************************************************************
void ErrTestReport ( void );
void HexPrint ( int i, unsigned char *p );
int lcstrcmp ( char *, char * );  // case independent ~strcmp
void Test32 ( void );
void Test64 ( void );
void Test128 ( void );
void Test256 ( void );
void Test512 ( void );
void Test1024 ( void );
void TestNValue ( char *subfunc,  // test calculated value
                  char *string,
                  int N,  // size
                  uint8_t *was,
                  uint8_t should[N] );
int TestR ( char *, int should, int was );  // test return code
void ValueTestReport ( void );

#ifndef FNV_64bitIntegers
# undef uint64
# define uint64_t no_64_bit_integers
#endif /* FNV_64bitIntegers */

struct {
    int length;
    void (*func)( void );

} funcmap[] = {  // valid sizes
    { 32, Test32 },
    { 64, Test64 },
```

```
    { 128, Test128 },
    { 256, Test256 },
    { 512, Test512 },
    { 1024, Test1024 },
    { 0, Test32 }  // fence post
};


//*************************************************************
//  main
//*************************************************************
int main ( int argc, const char **argv ) {
    int i,
        option,  // command line option letter
        count,
        argval,
        selected=0;  // selected hash size
          // 0 = none, 1 = 32, 2 = 64, ..., 6 = 1024
    FILE *fp;// file pointer
#ifdef FNV_64bitIntegers
    uint64_t    i64;
#endif
    uint32_t    i32;
    uint16_t    endianness = 5*256 + 11;
    uint8_t     hash[FNV1024size];
    uint8_t     ch;

#ifdef FNV_BigEndian
    printf ("Calculating with Big Endian.\n");
    if ( ((uint8_t *)&endianness)[0] != 5 )
        printf ("But computer seems to be Little Endian!\n");
#else
    printf ("Calculating with Little Endian.\n");
    if ( ((uint8_t *)&endianness)[0] != 11 )
        printf ("But computer seems to be Big Endian!\n");
#endif
    if ( argc == 1 ) {  // if no arguments
printf ( "FNVhash [-a] [-d] [-tnnn] [-unnn] [-ffilename] [*token]\n"
                "  -a = test All\n"
                "  -ffilename = hash file contents\n"
                "  -tnnn = Test hash size nnn\n"
                "  -unnn = Use hash size nnn\n"
                "  -v = toggle Verbose flag\n"
                "  Each token is hashed.\n" );
        return 0;
    }

// process command line options
// *************************************************************
```

```
    while ((option = getopt(argc, (char *const *)argv, ":af:t:u:v"))
           != -1) {
        if ( verbose )
            printf ( "Got option %c\n", option );
        switch ( option ) {
            case 'a':  // run all tests
                Test32();
                Test64();
                Test128();
                Test256();
                Test512();
                Test1024();
                break;
            case 'f':   // followed by name of file to hash
                if ( ( fp = fopen ( optarg, "r") ) == NULL ) {
                  printf ( "File open of '%s' fails.\n", optarg );
                  break;
                }
                if ( !selected ) {
                    printf ( "No hash size selected.\n" );
                    break;
                }
                printf ( "FNV-%i Hash of contents of file '%s':\n",
                         funcmap[selected-1].length, optarg );
                switch ( selected ) {
                  case 1:
                    FNV32init (&e32Context);
                    while ( ( i = getc ( fp ) ) != EOF ) {
                        ch = (uint8_t)i;
                        FNV32blockin ( &e32Context, &ch, 1);
                    }
                    FNV32result ( &e32Context, &eUint32 );
                    HexPrint ( FNV32size,
                               (unsigned char *)&eUint32 );
                    break;
                  case 2:
                    FNV64init (&e64Context);
                    while ( ( i = getc ( fp ) ) != EOF ) {
                        ch = (uint8_t)i;
                        FNV64blockin ( &e64Context, &ch, 1);
                    }
#ifdef FNV_64bitIntegers
                    FNV64result ( &e64Context, &eUint64 );
                    HexPrint ( FNV64size,
                               (unsigned char *)&eUint64 );
#else
                    FNV64result ( &e64Context, Vec64 );
                    HexPrint ( FNV64size, Vec64 );
#endif
```

```
          break;
        case 3:
          FNV128init (&e128Context);
          while ( ( i = getc ( fp ) ) != EOF ) {
              ch = (uint8_t)i;
              FNV128blockin ( &e128Context, &ch, 1);
              }
          FNV128result ( &e128Context, Vec128 );
          HexPrint ( FNV128size, Vec128 );
          break;
        case 4:
          FNV256init (&e256Context);
          while ( ( i = getc ( fp ) ) != EOF ) {
              ch = (uint8_t)i;
              FNV256blockin ( &e256Context, &ch, 1);
          }
          FNV256result ( &e256Context, Vec256 );
          HexPrint ( FNV256size, Vec256 );
          break;
        case 5:
          FNV512init (&e512Context);
          while ( ( i = getc ( fp ) ) != EOF ) {
              ch = (uint8_t)i;
              FNV512blockin ( &e512Context, &ch, 1);
          }
          while ( ( i = getc ( fp ) ) != EOF ) {
              ch = (uint8_t)i;
              FNV512blockin ( &e512Context, &ch, 1);
          }
          FNV512result ( &e512Context, Vec512 );
          HexPrint ( FNV512size, Vec512 );
          break;
        case 6:
          FNV1024init (&e1024Context);
          while ( ( i = getc ( fp ) ) != EOF ) {
              ch = (uint8_t)i;
              FNV1024blockin ( &e1024Context, &ch, 1);
          }
          FNV1024result ( &e1024Context, Vec1024 );
          HexPrint ( FNV1024size, Vec1024 );
          break;
        default:
          return 1;  // internal error
      }  // end switch selected
      printf ( "\n" );
      break;
  case 't':   // follow by size of FNV to test, 0->all
      count = sscanf ( optarg, "%i", &argval );
      selected = 0;
```

```
                if ( count != EOF || !count )
                    for ( i = 0; funcmap[i].length; ++i ) {
                        if ( funcmap[i].length == argval ) {
                            selected = i + 1;
                            funcmap[i].func(); // selected hash test
                            break;  // out of for
                        }  /* end if */
                    }  /* end for */
                if ( !selected )
                    printf ( "Bad argument to option -t\n"
                             "Valid sizes are 32, 64, 128,"
                             " 256,512, and 1024\n");
                break;
            case 'u':   // follow by size of FNV to use
                count = sscanf ( optarg, "%i", &argval );
                selected = 0;
                if ( count != EOF || !count )
                    for ( i = 0; funcmap[i].length; ++i ) {
                        if ( funcmap[i].length == argval ) {
                            selected = i + 1;
                            break;  // out of for
                        }
                    }
                if ( !selected )
                    printf ( "Bad argument to option -u\n"
                             "Valid sizes are 32, 64, 128,"
                             "256, 512, and 1024\n" );
                break;
            case 'v':   // toggle Verbose flag
                if ( (verbose ^= 1) ) {// tests the TestR function
                    printf ( "Verbose on.\n" );
#ifdef FNV_64bitIntegers
                    printf ("Has 64-bit Integers. ");
#else
                    printf ("Does not have 64-bit integers. ");
#endif
                    funcName = "Testing TestR";
                    TestR ( "should fail", 1, 2 );
                    TestR ( "should not have failed", 3, 3 );
                }
                else
                    printf ( "Verbose off.\n" );
                break;
            case '?':   //
                printf ( "Unknown option %c\n", optopt );
                return 1;
        }  /* end switch */
    }  /* end while */
    if ( ( option == -1 ) && verbose )
```

```
        printf ( "No more options.\n" );

// Through all the options, now, if a size is set, encrypt any
//   other tokens on the command line
//*****************************************************
    for ( i = optind; i < argc; ++i ) {
        if ( !selected ) {
            printf ( "No hash size selected.\n" );
            return 0;
        }
        printf ( "FNV-%i of '%s' is ",
                 funcmap[selected-1].length, argv[i] );
        switch ( selected ) {
            case 1:
                FNV32string ( argv[i], &i32 );
                HexPrint ( FNV32size, (uint8_t *)&i32 );
                break;
            case 2:
#ifdef FNV_64bitIntegers
                FNV64string ( argv[i], &i64 );
                HexPrint ( FNV64size, (uint8_t *)&i64 );
#else
                FNV64string ( argv[i], hash );
                HexPrint ( FNV64size, hash );
#endif /* FNV_64bitIntegers */
                break;
            case 3:
                FNV128string ( argv[i], hash );
                HexPrint ( FNV128size, hash );
                break;
            case 4:
                FNV256string ( argv[i], hash );
                HexPrint ( FNV256size, hash );
                break;
            case 5:
                FNV512string ( argv[i], hash );
                HexPrint ( FNV512size, hash );
                break;
            case 6:
                FNV1024string ( argv[i], hash );
                HexPrint ( FNV1024size, hash );
                break;
            default:
                return 1;  // internal error
        }
        printf ( "\n" );
    }
    return 0;
}     /* end main */
```

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

//****************************************************************
//  Compare reversed bytes
//****************************************************************
int revcmp (uint8_t *buf1, uint8_t *buf2, int N) {
    int i;

    uint8_t *bufc = buf2 + N;
    for ( i = 0; i < N ; i++ )
        if (*buf1++ != *--bufc)
          return 0;
    return 1;
}   /* end revcmp */

//****************************************************************
//  Case independent leading chars string compare
//  like strcmp, returns zero if all of input matches same number
//      of initial chars of value, nonzero if none match
//****************************************************************
int lcstrcmp ( char *input, char *value ) {
    char chi, chv;

    while ( ( chi = tolower(*input++) ) ) {  // loop until zero
        if ( ( chv = tolower(*value++) ) ) {
                    if ( chi != chv )
                        return 1; // mismatch
                }
            else
                return 1; // input longer than value
        }
    return 0; // reached end of input
}   /* end lcstrcmp */

//****************************************************************
//  General byte vector return value test
//****************************************************************
```

```
void TestNValue ( char *subfunc,
                  char *string,
                  int N,
                  uint8_t was[N],
                  uint8_t should[N] ) {
#ifdef FNV_BigEndian
    if ( revcmp ( was, should, N) == 0 ) {
#else
    if ( memcmp ( was, should, N) != 0 ) {
#endif
        ++Terr;
        printf ( "%s %s of '%s'",
                 funcName, subfunc, string );
        printf ( " computed " );
        HexPrint ( N, was );
        printf ( ", expected " );
        HexPrint ( N, should );
        printf ( ".\n" );
        }
    else if ( verbose ) {
        printf ( "%s %s of '%s' computed ",
                 funcName, subfunc, string );
        HexPrint ( N, was );
        printf ( " as expected.\n" );
        }
}   /* end TestNValue */

//*************************************************************
//  Report on status/value returns
//*************************************************************
void ErrTestReport ( void ) {
    if ( Terr )
        printf ( "%s test of error checks failed %i times.\n",
                 funcName, Terr );
    else if ( verbose )
        printf ( "%s test of error checks passed.\n",
                 funcName );
}  /* end ErrTestReport */

void ValueTestReport ( void ) {
    if ( Terr )
        printf ( "%s test of return values failed %i times.\n",
                 funcName, Terr );
    else
        printf ( "%s test of return values passed.\n", funcName );
}  /* end ValueTestReport */

//*************************************************************
//  Print some bytes as hexadecimal
```

```
//*************************************************************
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 ( void ) {
    int            i, err;
    long int       iLen;
    uint32_t    FNV32svalues[NTstrings] = {
      0x811c9dc5, 0xe40c292c, 0xbf9cf968, 0xfd9d3881 };
    uint32_t    FNV32bvalues[NTstrings] = {
      0x050c5d1f, 0x2b24d044, 0x0c1c9eb8, 0xbf7ff313 };

    funcName = "FNV-32";
/* test error checks */
    Terr = 0;
    TestR ( "init1", fnvSuccess, FNV32init (&e32Context) );
    TestR ( "string1", fnvNull,
            FNV32string ( (char *)0, &eUint32 ) );
    TestR ( "string2", fnvNull,
            FNV32string ( errteststring, (uint32_t *)0 ) );
    TestR ( "block1", fnvNull,
            FNV32block ( (uint8_t *)0, 1, &eUint32 ) );
    TestR ( "block2", fnvBadParam,
            FNV32block ( errtestbytes, -1, &eUint32 ) );
    TestR ( "block3", fnvNull,
            FNV32block ( errtestbytes, 1, (uint32_t *)0 ) );
    TestR ( "init2", fnvNull,
            FNV32init ( (FNV32context *)0 ) );
    TestR ( "initBasis1", fnvNull,
            FNV32initBasis ( (FNV32context *)0, eUint32 ) );
    TestR ( "blockin1", fnvNull,
            FNV32blockin ( (FNV32context *)0,
                           errtestbytes, NTestBytes ) );
    TestR ( "blockin2", fnvNull,
            FNV32blockin ( &e32Context, (uint8_t *)0,
                           NTestBytes ) );
    TestR ( "blockin3", fnvBadParam,
            FNV32blockin ( &e32Context, errtestbytes, -1 ) );
    e32Context.Computed = FNVclobber+FNV32state;
    TestR ( "blockin4", fnvStateError,
            FNV32blockin ( &e32Context, errtestbytes,
                           NTestBytes ) );
```

```
    TestR ( "stringin1", fnvNull,
            FNV32stringin ( (FNV32context *)0, errteststring ) );
    TestR ( "stringin2", fnvNull,
            FNV32stringin ( &e32Context, (char *)0 ) );
    TestR ( "stringin3", fnvStateError,
            FNV32stringin ( &e32Context, errteststring ) );
    TestR ( "result1", fnvNull,
            FNV32result ( (FNV32context *)0, &eUint32 ) );
    TestR ( "result2", fnvNull,
            FNV32result ( &e32Context, (uint32_t *)0  ) );
    TestR ( "result3", fnvStateError,
            FNV32result ( &e32Context, &eUint32  ) );
    ErrTestReport ();
/* test actual results */
    Terr = 0;
    for ( i = 0; i < NTstrings; ++i ) {
        err = TestR ( "stringa", fnvSuccess,
                      FNV32string ( teststring[i], &eUint32 ) );
        if ( err == fnvSuccess )
            TestNValue ( "stringb", teststring[i], FNV32size,
                         (uint8_t *)&eUint32,
                         (uint8_t *)&FNV32svalues[i] );
        err = TestR ( "blocka", fnvSuccess,
                      FNV32block ( (uint8_t *)teststring[i],
                           (unsigned long)(strlen(teststring[i])+1),
                           &eUint32 ) );
        if ( err == fnvSuccess )
            TestNValue ( "blockb", teststring[i], FNV32size,
                         (uint8_t *)&eUint32,
                         (uint8_t *)&FNV32bvalues[i] );
/* now try testing the incremental stuff */
        err = TestR ( "inita", fnvSuccess,
                      FNV32init (&e32Context) );
        if ( err ) break;
        iLen = strlen ( teststring[i] );
        err = TestR ( "blockina", fnvSuccess,
                      FNV32blockin ( &e32Context,
                                     (uint8_t *)teststring[i],
                                     iLen/2 ) );
        if ( err ) break;
        err = TestR ( "stringina", fnvSuccess,
                      FNV32stringin ( &e32Context,
                                    teststring[i] + iLen/2 ) );
        err = TestR ( "resulta", fnvSuccess,
                      FNV32result ( &e32Context, &eUint32 ) );
        if ( err ) break;
        TestNValue ( "incrementala", teststring[i], FNV32size,
                     (uint8_t *)&eUint32,
                     (uint8_t *)&FNV32svalues[i] );
```

```
        }
    ValueTestReport ();
}    /* end Test32 */

#ifdef FNV_64bitIntegers
//***************************************************************
//  Code for FNV64 using 64-bit integers
//***************************************************************
void Test64 ( void ) { /* with 64-bit integers */
    int        i, err;
    long int   iLen;
    FNV64context   e64Context;
    uint64_t   FNV64svalues[NTstrings] = {
        0xcbf29ce484222325, 0xaf63dc4c8601ec8c, 0x85944171f73967e8,
        0xbd51ea7094ee6fa1 };
    uint64_t   FNV64bvalues[NTstrings] = {
        0xaf63bd4c8601b7df, 0x089be207b544f1e4, 0x34531ca7168b8f38,
        0xa0a0fe4d1127ae93 };

    funcName = "FNV-64";
/* test error checks */
    Terr = 0;
    TestR ( "init1", fnvSuccess, FNV64init (&e64Context) );
    TestR ( "string1", fnvNull,
            FNV64string ( (char *)0, &eUint64 ) );
    TestR ( "string2", fnvNull,
            FNV64string ( errteststring, (uint64_t *)0 ) );
    TestR ( "block1", fnvNull,
            FNV64block ( (uint8_t *)0, 1, &eUint64 ) );
    TestR ( "block2", fnvBadParam,
            FNV64block ( errtestbytes, -1, &eUint64 ) );
    TestR ( "block3", fnvNull,
            FNV64block ( errtestbytes, 1, (uint64_t *)0 ) );
    TestR ( "init2", fnvNull,
            FNV64init ( (FNV64context *)0 ) );
    TestR ( "initBasis1", fnvNull,
            FNV64initBasis ( (FNV64context *)0, eUint64 ) );
    TestR ( "blockin1", fnvNull,
            FNV64blockin ( (FNV64context *)0,
                           errtestbytes, NTestBytes ) );
    TestR ( "blockin2", fnvNull,
            FNV64blockin ( &e64Context, (uint8_t *)0,
                           NTestBytes ) );
    TestR ( "blockin3", fnvBadParam,
            FNV64blockin ( &e64Context, errtestbytes, -1 ) );
    e64Context.Computed = FNVclobber+FNV64state;
    TestR ( "blockin4", fnvStateError,
            FNV64blockin ( &e64Context, errtestbytes,
                           NTestBytes ) );
```

```
    TestR ( "stringin1", fnvNull,
            FNV64stringin ( (FNV64context *)0, errteststring ) );
    TestR ( "stringin2", fnvNull,
            FNV64stringin ( &e64Context, (char *)0 ) );
    TestR ( "stringin3", fnvStateError,
            FNV64stringin ( &e64Context, errteststring ) );
    TestR ( "result1", fnvNull,
            FNV64result ( (FNV64context *)0, &eUint64 ) );
    TestR ( "result2", fnvNull,
            FNV64result ( &e64Context, (uint64_t *)0  ) );
    TestR ( "result3", fnvStateError,
            FNV64result ( &e64Context, &eUint64  ) );
    ErrTestReport ();
/* test actual results */
    Terr = 0;
    for ( i = 0; i < NTstrings; ++i ) {
        err = TestR ( "stringa", fnvSuccess,
                      FNV64string ( teststring[i], &eUint64 ) );
        if ( err == fnvSuccess )
            TestNValue ( "stringb", teststring[i], FNV64size,
                         (uint8_t *)&eUint64,
                         (uint8_t *)&FNV64svalues[i] );
        err = TestR ( "blocka", fnvSuccess,
                      FNV64block ( (uint8_t *)teststring[i],
                          (unsigned long)(strlen(teststring[i])+1),
                          &eUint64 ) );
        if ( err == fnvSuccess )
             TestNValue ( "blockb", teststring[i], FNV64size,
                          (uint8_t *)&eUint64,
                          (uint8_t *)&FNV64bvalues[i] );
/* now try testing the incremental stuff */
        err = TestR ( "inita", fnvSuccess,
                      FNV64init (&e64Context) );
        if ( err ) break;
        iLen = strlen ( teststring[i] );
        err = TestR ( "blockina", fnvSuccess,
                      FNV64blockin ( &e64Context,
                                     (uint8_t *)teststring[i],
                                     iLen/2 ) );
        if ( err ) break;
        err = TestR ( "stringina", fnvSuccess,
                      FNV64stringin ( &e64Context,
                                    teststring[i] + iLen/2 ) );
        err = TestR ( "resulta", fnvSuccess,
                      FNV64result ( &e64Context, &eUint64 ) );
        if ( err ) break;
        TestNValue ( "incrementala", teststring[i], FNV64size,
                     (uint8_t *)&eUint64,
                     (uint8_t *)&FNV64svalues[i] );
```

```
        }
    ValueTestReport ();
}    /* end Test64 */

#else

//*************************************************************
//  Code for FNV64 without 64-bit integers
//*************************************************************
void Test64 ( void ) { /* without 64-bit integers */
    int    i, err;
    long int   iLen;
    uint8_t   FNV64svalues[NTstrings][FNV64size] = {
        { 0xcb, 0xf2, 0x9c, 0xe4, 0x84, 0x22, 0x23, 0x25 },
        { 0xaf, 0x63, 0xdc, 0x4c, 0x86, 0x01, 0xec, 0x8c },
        { 0x85, 0x94, 0x41, 0x71, 0xf7, 0x39, 0x67, 0xe8 },
        { 0xbd, 0x51, 0xea, 0x70, 0x94, 0xee, 0x6f, 0xa1 } };
    uint8_t   FNV64bvalues[NTstrings][FNV64size] = {
        { 0xaf, 0x63, 0xbd, 0x4c, 0x86, 0x01, 0xb7, 0xdf },
        { 0x08, 0x9b, 0xe2, 0x07, 0xb5, 0x44, 0xf1, 0xe4 },
        { 0x34, 0x53, 0x1c, 0xa7, 0x16, 0x8b, 0x8f, 0x38 },
        { 0xa0, 0xa0, 0xfe, 0x4d, 0x11, 0x27, 0xae, 0x93 } };

    funcName = "FNV-64";
/* test error checks */
    Terr = 0;
    TestR ( "init1", fnvSuccess, FNV64init (&e64Context) );
    TestR ( "string1", fnvNull,
            FNV64string ( (char *)0, Vec64 ) );
    TestR ( "string2", fnvNull,
            FNV64string ( errteststring, (uint8_t *)0 ) );
    TestR ( "block1", fnvNull,
            FNV64block ( (uint8_t *)0, 1, Vec64 ) );
    TestR ( "block2", fnvBadParam,
            FNV64block ( errtestbytes, -1, Vec64 ) );
    TestR ( "block3", fnvNull,
            FNV64block ( errtestbytes, 1, (uint8_t *)0 ) );
    TestR ( "init2", fnvNull,
            FNV64init ( (FNV64context *)0 ) );
    TestR ( "initBasis1", fnvNull,
            FNV64initBasis ( (FNV64context *)0, Vec64 ) );
    TestR ( "blockin1", fnvNull,
            FNV64blockin ( (FNV64context *)0,
                           errtestbytes, NTestBytes ) );
    TestR ( "blockin2", fnvNull,
            FNV64blockin ( &e64Context, (uint8_t *)0,
                           NTestBytes ) );
    TestR ( "blockin3", fnvBadParam,
            FNV64blockin ( &e64Context, errtestbytes, -1 ) );
```

```
    e64Context.Computed = FNVclobber+FNV64state;
    TestR ( "blockin4", fnvStateError,
            FNV64blockin ( &e64Context, errtestbytes,
                           NTestBytes ) );
    TestR ( "stringin1", fnvNull,
            FNV64stringin ( (FNV64context *)0, errteststring ) );
    TestR ( "stringin2", fnvNull,
            FNV64stringin ( &e64Context, (char *)0 ) );
    TestR ( "stringin3", fnvStateError,
            FNV64stringin ( &e64Context, errteststring ) );
    TestR ( "result1", fnvNull,
            FNV64result ( (FNV64context *)0, Vec64 ) );
    TestR ( "result2", fnvNull,
            FNV64result ( &e64Context, (uint8_t *)0  ) );
    TestR ( "result3", fnvStateError,
            FNV64result ( &e64Context, Vec64 ) );
    ErrTestReport ();
/* test actual results */
    Terr = 0;
    for ( i = 0; i < NTstrings; ++i ) {
        err = TestR ( "stringa", fnvSuccess,
                      FNV64string ( teststring[i], Vec64 ) );
        if ( err == fnvSuccess )
            TestNValue ( "stringb", teststring[i], FNV64size,
                         Vec64, FNV64svalues[i] );
        err = TestR ( "blocka", fnvSuccess,
                      FNV64block ( (uint8_t *)teststring[i],
                          (unsigned long)(strlen(teststring[i])+1),
                          Vec64 ) );
        if ( err == fnvSuccess )
            TestNValue ( "blockb", teststring[i], FNV64size,
                         Vec64,
                         FNV64bvalues[i] );
/* now try testing the incremental stuff */
        err = TestR ( "inita", fnvSuccess,
                      FNV64init ( &e64Context ) );
        if ( err ) break;
        iLen = strlen ( teststring[i] );
        err = TestR ( "blockina", fnvSuccess,
                      FNV64blockin ( &e64Context,
                                     (uint8_t *)teststring[i],
                                     iLen/2 ) );
        if ( err ) break;
        err = TestR ( "stringina", fnvSuccess,
                      FNV64stringin ( &e64Context,
                                      teststring[i] + iLen/2 ) );
        err = TestR ( "resulta", fnvSuccess,
                      FNV64result ( &e64Context, Vec64 ) );
        if ( err ) break;
```

```
        TestNValue ( "incrementala", teststring[i], FNV64size,
                     Vec64, (uint8_t *)&FNV64svalues[i] );
        }
    ValueTestReport ();
}
#endif /* FNV_64bitIntegers */

//*************************************************************
//  Code for FNV128
//*************************************************************
void Test128 ( void ) {
    int        i, err;
    long int   iLen;
    uint8_t   FNV128svalues[NTstrings][FNV128size] = {
        { 0x6c, 0x62, 0x27, 0x2e, 0x07, 0xbb, 0x01, 0x42,
          0x62, 0xb8, 0x21, 0x75, 0x62, 0x95, 0xc5, 0x8d },
        { 0xd2, 0x28, 0xcb, 0x69, 0x6f, 0x1a, 0x8c, 0xaf,
          0x78, 0x91, 0x2b, 0x70, 0x4e, 0x4a, 0x89, 0x64 },
        { 0x34, 0x3e, 0x16, 0x62, 0x79, 0x3c, 0x64, 0xbf,
          0x6f, 0x0d, 0x35, 0x97, 0xba, 0x44, 0x6f, 0x18 },
        { 0x74, 0x20, 0x2c, 0x60, 0x0b, 0x05, 0x1c, 0x16,
          0x5b, 0x1a, 0xca, 0xfe, 0xd1, 0x0d, 0x14, 0x19 } };
    uint8_t   FNV128bvalues[NTstrings][FNV128size] = {
        { 0xd2, 0x28, 0xcb, 0x69, 0x10, 0x1a, 0x8c, 0xaf,
          0x78, 0x91, 0x2b, 0x70, 0x4e, 0x4a, 0x14, 0x7f },
        { 0x08, 0x80, 0x95, 0x45, 0x19, 0xab, 0x1b, 0xe9,
          0x5a, 0xa0, 0x73, 0x30, 0x55, 0xb7, 0x0e, 0x0c },
        { 0xe0, 0x1f, 0xcf, 0x9a, 0x45, 0x4f, 0xf7, 0x8d,
          0xa5, 0x40, 0xf1, 0xb2, 0x32, 0x34, 0xb2, 0x88 },
        { 0xe2, 0x67, 0xa7, 0x41, 0xa8, 0x49, 0x8f, 0x82,
          0x19, 0xf7, 0xc7, 0x8b, 0x3b, 0x17, 0xba, 0xc3 } };

    funcName = "FNV-128";
/* test error checks */
    Terr = 0;
    TestR ( "init1", fnvSuccess, FNV128init (&e128Context) );
    TestR ( "string1", fnvNull,
            FNV128string ( (char *)0, Vec128 ) );
    TestR ( "string2", fnvNull,
            FNV128string ( errteststring, (uint8_t *)0 ) );
    TestR ( "block1", fnvNull,
            FNV128block ( (uint8_t *)0, 1, Vec128 ) );
    TestR ( "block2", fnvBadParam,
            FNV128block ( errtestbytes, -1, Vec128 ) );
    TestR ( "block3", fnvNull,
            FNV128block ( errtestbytes, 1, (uint8_t *)0 ) );
    TestR ( "init2", fnvNull,
            FNV128init ( (FNV128context *)0 ) );
    TestR ( "initBasis1", fnvNull,
```

```
            FNV128initBasis ( (FNV128context *)0, Vec128 ) );
    TestR ( "blockin1", fnvNull,
            FNV128blockin ( (FNV128context *)0,
                            errtestbytes, NTestBytes ) );
    TestR ( "blockin2", fnvNull,
            FNV128blockin ( &e128Context, (uint8_t *)0,
                            NTestBytes ) );
    TestR ( "blockin3", fnvBadParam,
            FNV128blockin ( &e128Context, errtestbytes, -1 ) );
    e128Context.Computed = FNVclobber+FNV128state;
    TestR ( "blockin4", fnvStateError,
            FNV128blockin ( &e128Context, errtestbytes,
                            NTestBytes ) );
    TestR ( "stringin1", fnvNull,
            FNV128stringin ( (FNV128context *)0, errteststring ) );
    TestR ( "stringin2", fnvNull,
            FNV128stringin ( &e128Context, (char *)0 ) );
    TestR ( "stringin3", fnvStateError,
            FNV128stringin ( &e128Context, errteststring ) );
    TestR ( "result1", fnvNull,
            FNV128result ( (FNV128context *)0, Vec128 ) );
    TestR ( "result2", fnvNull,
            FNV128result ( &e128Context, (uint8_t *)0  ) );
    TestR ( "result3", fnvStateError,
            FNV128result ( &e128Context, Vec128  ) );
    ErrTestReport ();
/* test actual results */
    Terr = 0;
    for ( i = 0; i < NTstrings; ++i ) {
        err = TestR ( "stringa", fnvSuccess,
                      FNV128string ( teststring[i], Vec128 ) );
        if ( err == fnvSuccess )
            TestNValue ( "stringb", teststring[i], FNV128size,
                            Vec128,
                     FNV128svalues[i] );
        err = TestR ( "blocka", fnvSuccess,
                      FNV128block ( (uint8_t *)teststring[i],
                          (unsigned long)(strlen(teststring[i])+1),
                          Vec128 ) );
        if ( err == fnvSuccess )
            TestNValue ( "blockb", teststring[i], FNV128size,
                         Vec128,
                         FNV128bvalues[i] );
/* now try testing the incremental stuff */
        err = TestR ( "inita", fnvSuccess,
                      FNV128init ( &e128Context ) );
        if ( err ) break;
        iLen = strlen ( teststring[i] );
        err = TestR ( "blockina", fnvSuccess,
```

```
                    FNV128blockin ( &e128Context,
                    (uint8_t *)teststring[i],
                    iLen/2 ) );
        if ( err ) break;
        err = TestR ( "stringina", fnvSuccess,
                    FNV128stringin ( &e128Context,
                    teststring[i] + iLen/2 ) );
        err = TestR ( "resulta", fnvSuccess,
                    FNV128result ( &e128Context, Vec128 ) );
        if ( err ) break;
        TestNValue ( "incrementala", teststring[i], FNV128size,
                   Vec128,
                    (uint8_t *)&FNV128svalues[i] );
        }
    ValueTestReport ();
}    /* end Test128 */

//*************************************************************
//  Code for FNV256
//*************************************************************
void Test256 ( void ) {
    int        i, err;
    long int   iLen;
    uint8_t   FNV256svalues[NTstrings][FNV256size] = {
        { 0xdd, 0x26, 0x8d, 0xbc, 0xaa, 0xc5, 0x50, 0x36,
          0x2d, 0x98, 0xc3, 0x84, 0xc4, 0xe5, 0x76, 0xcc,
          0xc8, 0xb1, 0x53, 0x68, 0x47, 0xb6, 0xbb, 0xb3,
          0x10, 0x23, 0xb4, 0xc8, 0xca, 0xee, 0x05, 0x35 },
        { 0x63, 0x32, 0x3f, 0xb0, 0xf3, 0x53, 0x03, 0xec,
          0x28, 0xdc, 0x75, 0x1d, 0x0a, 0x33, 0xbd, 0xfa,
          0x4d, 0xe6, 0xa9, 0x9b, 0x72, 0x66, 0x49, 0x4f,
          0x61, 0x83, 0xb2, 0x71, 0x68, 0x11, 0x63, 0x7c },
        { 0xb0, 0x55, 0xea, 0x2f, 0x30, 0x6c, 0xad, 0xad,
          0x4f, 0x0f, 0x81, 0xc0, 0x2d, 0x38, 0x89, 0xdc,
          0x32, 0x45, 0x3d, 0xad, 0x5a, 0xe3, 0x5b, 0x75,
          0x3b, 0xa1, 0xa9, 0x10, 0x84, 0xaf, 0x34, 0x28 },
        { 0x0c, 0x5a, 0x44, 0x40, 0x2c, 0x65, 0x38, 0xcf,
          0x98, 0xef, 0x20, 0xc4, 0x03, 0xa8, 0x0f, 0x65,
          0x9b, 0x80, 0xc9, 0xa5, 0xb0, 0x1a, 0x6a, 0x87,
          0x34, 0x2e, 0x26, 0x72, 0x64, 0x45, 0x67, 0xb1 } };
    uint8_t   FNV256bvalues[NTstrings][FNV256size] = {
        { 0x63, 0x32, 0x3f, 0xb0, 0xf3, 0x53, 0x03, 0xec,
          0x28, 0xdc, 0x56, 0x1d, 0x0a, 0x33, 0xbd, 0xfa,
          0x4d, 0xe6, 0xa9, 0x9b, 0x72, 0x66, 0x49, 0x4f,
          0x61, 0x83, 0xb2, 0x71, 0x68, 0x11, 0x38, 0x7f },
        { 0xf4, 0xf7, 0xa1, 0xc2, 0xef, 0xd0, 0xe1, 0xe4,
          0xbb, 0x19, 0xe3, 0x45, 0x25, 0xc0, 0x72, 0x1a,
          0x06, 0xdd, 0x32, 0x8f, 0xa3, 0xd7, 0xa9, 0x14,
          0x39, 0xa0, 0x73, 0x43, 0x50, 0x1c, 0xf4, 0xf4 },
```

```
        { 0x6a, 0x7f, 0x34, 0xab, 0xc8, 0x5d, 0xe7, 0xd9,
          0x51, 0xb5, 0x15, 0x7e, 0xb5, 0x67, 0x2c, 0x59,
          0xb6, 0x04, 0x87, 0x65, 0x09, 0x47, 0xd3, 0x91,
          0xb1, 0x2d, 0x71, 0xe7, 0xfe, 0xf5, 0x53, 0x78 },
        { 0x3b, 0x97, 0x2c, 0x31, 0xbe, 0x84, 0x3a, 0x45,
          0x59, 0x02, 0x20, 0xd1, 0x12, 0x0d, 0x59, 0xe6,
          0xa3, 0x97, 0xa0, 0xc3, 0x34, 0xa1, 0xb9, 0x7d,
          0x5b, 0xff, 0x50, 0xa1, 0x0c, 0x3e, 0xca, 0x73 } };

    funcName = "FNV-256";
/* test error checks */
    Terr = 0;
    TestR ( "init1", fnvSuccess, FNV256init (&e256Context) );
    TestR ( "string1", fnvNull,
            FNV256string ( (char *)0, Vec256 ) );
    TestR ( "string2", fnvNull,
            FNV256string ( errteststring, (uint8_t *)0 ) );
    TestR ( "block1", fnvNull,
            FNV256block ( (uint8_t *)0, 1, Vec256 ) );
    TestR ( "block2", fnvBadParam,
            FNV256block ( errtestbytes, -1, Vec256 ) );
    TestR ( "block3", fnvNull,
            FNV256block ( errtestbytes, 1, (uint8_t *)0 ) );
    TestR ( "init2", fnvNull,
            FNV256init ( (FNV256context *)0 ) );
    TestR ( "initBasis1", fnvNull,
            FNV256initBasis ( (FNV256context *)0, Vec256 ) );
    TestR ( "blockin1", fnvNull,
            FNV256blockin ( (FNV256context *)0,
                            errtestbytes, NTestBytes ) );
    TestR ( "blockin2", fnvNull,
            FNV256blockin ( &e256Context, (uint8_t *)0,
                            NTestBytes ) );
    TestR ( "blockin3", fnvBadParam,
            FNV256blockin ( &e256Context, errtestbytes, -1 ) );
    e256Context.Computed = FNVclobber+FNV256state;
    TestR ( "blockin4", fnvStateError,
            FNV256blockin ( &e256Context, errtestbytes,
                            NTestBytes ) );
    TestR ( "stringin1", fnvNull,
            FNV256stringin ( (FNV256context *)0, errteststring ) );
    TestR ( "stringin2", fnvNull,
            FNV256stringin ( &e256Context, (char *)0 ) );
    TestR ( "stringin3", fnvStateError,
            FNV256stringin ( &e256Context, errteststring ) );
    TestR ( "result1", fnvNull,
            FNV256result ( (FNV256context *)0, Vec256 ) );
    TestR ( "result2", fnvNull,
            FNV256result ( &e256Context, (uint8_t *)0  ) );
```

```
    TestR ( "result3", fnvStateError,
            FNV256result ( &e256Context, Vec256  ) );
    ErrTestReport ();
/* test actual results */
    Terr = 0;
    for ( i = 0; i < NTstrings; ++i ) {
        err = TestR ( "stringa", fnvSuccess,
                      FNV256string ( teststring[i], Vec256 ) );
        if ( err == fnvSuccess )
            TestNValue ( "stringb", teststring[i], FNV256size,
                         Vec256,
                     FNV256svalues[i] );
        err = TestR ( "blocka", fnvSuccess,
                      FNV256block ( (uint8_t *)teststring[i],
                          (unsigned long)(strlen(teststring[i])+1),
                          Vec256 ) );
        if ( err == fnvSuccess )
            TestNValue ( "blockb", teststring[i], FNV256size,
                     Vec256,
                     FNV256bvalues[i] );
/* now try testing the incremental stuff */
        err = TestR ( "inita", fnvSuccess,
                      FNV256init ( &e256Context ) );
        if ( err ) break;
        iLen = strlen ( teststring[i] );
        err = TestR ( "blockina", fnvSuccess,
                      FNV256blockin ( &e256Context,
                      (uint8_t *)teststring[i],
                      iLen/2 ) );
        if ( err ) break;
        err = TestR ( "stringina", fnvSuccess,
                      FNV256stringin ( &e256Context,
                      teststring[i] + iLen/2 ) );
        if ( err ) break;
        err = TestR ( "resulta", fnvSuccess,
                      FNV256result ( &e256Context, Vec256 ) );
        if ( err == fnvSuccess )
            TestNValue ( "incrementala", teststring[i], FNV256size,
                     Vec256,
                     (uint8_t *)&FNV256svalues[i] );
        }
    ValueTestReport ();
}    /* end Test256 */

//*************************************************************
//  Code for FNV512
//*************************************************************
void Test512 ( void ) {
    int        i, err;
```

```
long int   iLen;
uint8_t   FNV512svalues[NTstrings][FNV512size] = {
    { 0xb8, 0x6d, 0xb0, 0xb1, 0x17, 0x1f, 0x44, 0x16,
      0xdc, 0xa1, 0xe5, 0x0f, 0x30, 0x99, 0x90, 0xac,
      0xac, 0x87, 0xd0, 0x59, 0xc9, 0x00, 0x00, 0x00,
      0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x0d, 0x21,
      0xe9, 0x48, 0xf6, 0x8a, 0x34, 0xc1, 0x92, 0xf6,
      0x2e, 0xa7, 0x9b, 0xc9, 0x42, 0xdb, 0xe7, 0xce,
      0x18, 0x20, 0x36, 0x41, 0x5f, 0x56, 0xe3, 0x4b,
      0xac, 0x98, 0x2a, 0xac, 0x4a, 0xfe, 0x9f, 0xd9 },
    { 0xe4, 0x3a, 0x99, 0x2d, 0xc8, 0xfc, 0x5a, 0xd7,
      0xde, 0x49, 0x3e, 0x3d, 0x69, 0x6d, 0x6f, 0x85,
      0xd6, 0x43, 0x26, 0xec, 0x07, 0x00, 0x00, 0x00,
      0x00, 0x00, 0x00, 0x00, 0x00, 0x11, 0x98, 0x6f,
      0x90, 0xc2, 0x53, 0x2c, 0xaf, 0x5b, 0xe7, 0xd8,
      0x82, 0x91, 0xba, 0xa8, 0x94, 0xa3, 0x95, 0x22,
      0x53, 0x28, 0xb1, 0x96, 0xbd, 0x6a, 0x8a, 0x64,
      0x3f, 0xe1, 0x2c, 0xd8, 0x7b, 0x27, 0xff, 0x88 },
    { 0xb0, 0xec, 0x73, 0x8d, 0x9c, 0x6f, 0xd9, 0x69,
      0xd0, 0x5f, 0x0b, 0x35, 0xf6, 0xc0, 0xed, 0x53,
      0xad, 0xca, 0xcc, 0xcd, 0x8e, 0x00, 0x00, 0x00,
      0x4b, 0xf9, 0x9f, 0x58, 0xee, 0x41, 0x96, 0xaf,
      0xb9, 0x70, 0x0e, 0x20, 0x11, 0x08, 0x30, 0xfe,
      0xa5, 0x39, 0x6b, 0x76, 0x28, 0x0e, 0x47, 0xfd,
      0x02, 0x2b, 0x6e, 0x81, 0x33, 0x1c, 0xa1, 0xa9,
      0xce, 0xd7, 0x29, 0xc3, 0x64, 0xbe, 0x77, 0x88 },
    { 0x4f, 0xdf, 0x00, 0xec, 0xb9, 0xbc, 0x04, 0xdd,
      0x19, 0x38, 0x61, 0x8f, 0xe5, 0xc4, 0xfb, 0xb8,
      0x80, 0xa8, 0x2b, 0x15, 0xf5, 0xb6, 0xbd, 0x72,
      0x1e, 0xc2, 0xea, 0xfe, 0x03, 0xc4, 0x62, 0x48,
      0xf7, 0xa6, 0xc2, 0x47, 0x89, 0x92, 0x80, 0xd6,
      0xd2, 0xf4, 0x2f, 0xf6, 0xb4, 0x7b, 0xf2, 0x20,
      0x79, 0xdf, 0xd4, 0xbf, 0xe8, 0x7b, 0xf0, 0xbb,
      0x4e, 0x71, 0xea, 0xcb, 0x1e, 0x28, 0x77, 0x35 } };
uint8_t   FNV512bvalues[NTstrings][FNV512size] = {
    { 0xe4, 0x3a, 0x99, 0x2d, 0xc8, 0xfc, 0x5a, 0xd7,
      0xde, 0x49, 0x3e, 0x3d, 0x69, 0x6d, 0x6f, 0x85,
      0xd6, 0x43, 0x26, 0xec, 0x28, 0x00, 0x00, 0x00,
      0x00, 0x00, 0x00, 0x00, 0x00, 0x11, 0x98, 0x6f,
      0x90, 0xc2, 0x53, 0x2c, 0xaf, 0x5b, 0xe7, 0xd8,
      0x82, 0x91, 0xba, 0xa8, 0x94, 0xa3, 0x95, 0x22,
      0x53, 0x28, 0xb1, 0x96, 0xbd, 0x6a, 0x8a, 0x64,
      0x3f, 0xe1, 0x2c, 0xd8, 0x7b, 0x28, 0x2b, 0xbf },
    { 0x73, 0x17, 0xdf, 0xed, 0x6c, 0x70, 0xdf, 0xec,
      0x6a, 0xdf, 0xce, 0xd2, 0xa5, 0xe0, 0x4d, 0x7e,
      0xec, 0x74, 0x4e, 0x3c, 0xe9, 0x00, 0x00, 0x00,
      0x00, 0x00, 0x00, 0x00, 0x17, 0x93, 0x3d, 0x7a,
      0xf4, 0x5d, 0x70, 0xde, 0xf4, 0x23, 0xa3, 0x16,
      0xf1, 0x41, 0x17, 0xdf, 0x27, 0x2c, 0xd0, 0xfd,
```

```
          0x6b, 0x85, 0xf0, 0xf7, 0xc9, 0xbf, 0x6c, 0x51,
          0x96, 0xb3, 0x16, 0x0d, 0x02, 0x97, 0x5f, 0x38 },
        { 0x82, 0xf6, 0xe1, 0x04, 0x96, 0xde, 0x78, 0x34,
          0xb0, 0x8b, 0x21, 0xef, 0x46, 0x4c, 0xd2, 0x47,
          0x9e, 0x1d, 0x25, 0xe0, 0xca, 0x00, 0x00, 0x65,
          0xcb, 0x74, 0x80, 0x27, 0x39, 0xe0, 0xe5, 0x71,
          0x75, 0x22, 0xec, 0xf6, 0xd1, 0xf9, 0xa5, 0x2f,
          0x5f, 0xee, 0xfb, 0x4f, 0xab, 0x22, 0x73, 0xfd,
          0xe8, 0x31, 0x0f, 0x1b, 0x7b, 0x5c, 0x9a, 0x84,
          0x22, 0x48, 0xf4, 0xcb, 0xfb, 0x32, 0x27, 0x38 },
        { 0xfa, 0x7e, 0xb9, 0x1e, 0xfb, 0x64, 0x64, 0x11,
          0x8a, 0x73, 0x33, 0xbd, 0x96, 0x3b, 0xb6, 0x1f,
          0x2c, 0x6f, 0xe2, 0xe3, 0x6c, 0xd7, 0xd3, 0xe7,
          0x37, 0x28, 0xda, 0x57, 0x0c, 0x1f, 0xaf, 0xc3,
          0xd0, 0x6e, 0x4d, 0xd9, 0x53, 0x4a, 0x9f, 0xd4,
          0xa5, 0x2c, 0x43, 0x8b, 0xd2, 0x11, 0x69, 0x83,
          0x4a, 0xe6, 0x0d, 0x20, 0x7e, 0x0f, 0x8a, 0xf6,
          0x1a, 0xa1, 0x96, 0x25, 0x68, 0x37, 0xb8, 0x03 } };

    funcName = "FNV-512";
/* test error checks */
    Terr = 0;
    TestR ( "init1", fnvSuccess, FNV512init (&e512Context) );
    TestR ( "string1", fnvNull,
            FNV512string ( (char *)0, Vec512 ) );
    TestR ( "string2", fnvNull,
            FNV512string ( errteststring, (uint8_t *)0 ) );
    TestR ( "block1", fnvNull,
            FNV512block ( (uint8_t *)0, 1, Vec512 ) );
    TestR ( "block2", fnvBadParam,
            FNV512block ( errtestbytes, -1, Vec512 ) );
    TestR ( "block3", fnvNull,
            FNV512block ( errtestbytes, 1, (uint8_t *)0 ) );
    TestR ( "init2", fnvNull,
            FNV512init ( (FNV512context *)0 ) );
    TestR ( "initBasis1", fnvNull,
            FNV512initBasis ( (FNV512context *)0, Vec512 ) );
    TestR ( "blockin1", fnvNull,
            FNV512blockin ( (FNV512context *)0,
                            errtestbytes, NTestBytes ) );
    TestR ( "blockin2", fnvNull,
            FNV512blockin ( &e512Context, (uint8_t *)0,
                            NTestBytes ) );
    TestR ( "blockin3", fnvBadParam,
            FNV512blockin ( &e512Context, errtestbytes, -1 ) );
    e512Context.Computed = FNVclobber+FNV512state;
    TestR ( "blockin4", fnvStateError,
            FNV512blockin ( &e512Context, errtestbytes,
                            NTestBytes ) );
```

```
    TestR ( "stringin1", fnvNull,
            FNV512stringin ( (FNV512context *)0, errteststring ) );
    TestR ( "stringin2", fnvNull,
            FNV512stringin ( &e512Context, (char *)0 ) );
    TestR ( "stringin3", fnvStateError,
            FNV512stringin ( &e512Context, errteststring ) );
    TestR ( "result1", fnvNull,
            FNV512result ( (FNV512context *)0, Vec512 ) );
    TestR ( "result2", fnvNull,
            FNV512result ( &e512Context, (uint8_t *)0  ) );
    TestR ( "result3", fnvStateError,
            FNV512result ( &e512Context, Vec512  ) );
    ErrTestReport ();
/* test actual results */
    Terr = 0;
    for ( i = 0; i < NTstrings; ++i ) {
        err = TestR ( "stringa", fnvSuccess,
                      FNV512string ( teststring[i], Vec512 ) );
        if ( err == fnvSuccess )
            TestNValue ( "stringb", teststring[i], FNV512size,
                            Vec512,
                     FNV512svalues[i] );
        err = TestR ( "blocka", fnvSuccess,
                      FNV512block ( (uint8_t *)teststring[i],
                      (unsigned long)(strlen(teststring[i])+1),
                      Vec512 ) );
        if ( err == fnvSuccess )
            TestNValue ( "blockb", teststring[i], FNV512size,
                     Vec512,
                     FNV512bvalues[i] );
/* now try testing the incremental stuff */
        err = TestR ( "inita", fnvSuccess,
                      FNV512init ( &e512Context ) );
        if ( err ) break;
        iLen = strlen ( teststring[i] );
        err = TestR ( "blockina", fnvSuccess,
                      FNV512blockin ( &e512Context,
                      (uint8_t *)teststring[i],
                      iLen/2 ) );
        if ( err ) break;
        err = TestR ( "stringina", fnvSuccess,
                      FNV512stringin ( &e512Context,
                      teststring[i] + iLen/2 ) );
        if ( err ) break;
        err = TestR ( "resulta", fnvSuccess,
                      FNV512result ( &e512Context, Vec512 ) );
        if ( err == fnvSuccess )
            TestNValue ( "incrementala", teststring[i], FNV512size,
                     Vec512,
```

```
                    (uint8_t *)&FNV512svalues[i] );
        }
    ValueTestReport ();
}    /* end Test512 */

//*************************************************************
//  Code for FNV1024 using 64-bit integers
//*************************************************************
void Test1024 ( void ) {
    int        i, err;
    long int   iLen;
    uint8_t   FNV1024svalues[NTstrings][FNV1024size] = {
      { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x5f, 0x7a, 0x76, 0x75, 0x8e, 0xcc, 0x4d,
        0x32, 0xe5, 0x6d, 0x5a, 0x59, 0x10, 0x28, 0xb7,
        0x4b, 0x29, 0xfc, 0x42, 0x23, 0xfd, 0xad, 0xa1,
        0x6c, 0x3b, 0xf3, 0x4e, 0xda, 0x36, 0x74, 0xda,
        0x9a, 0x21, 0xd9, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x04, 0xc6, 0xd7,
        0xeb, 0x6e, 0x73, 0x80, 0x27, 0x34, 0x51, 0x0a,
        0x55, 0x5f, 0x25, 0x6c, 0xc0, 0x05, 0xae, 0x55,
        0x6b, 0xde, 0x8c, 0xc9, 0xc6, 0xa9, 0x3b, 0x21,
        0xaf, 0xf4, 0xb1, 0x6c, 0x71, 0xee, 0x90, 0xb3 },
      { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x98, 0xd7, 0xc1, 0x9f, 0xbc, 0xe6, 0x53, 0xdf,
        0x22, 0x1b, 0x9f, 0x71, 0x7d, 0x34, 0x90, 0xff,
        0x95, 0xca, 0x87, 0xfd, 0xae, 0xf3, 0x0d, 0x1b,
        0x82, 0x33, 0x72, 0xf8, 0x5b, 0x24, 0xa3, 0x72,
        0xf5, 0x0e, 0x57, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x07, 0x68, 0x5c, 0xd8,
        0x1a, 0x49, 0x1d, 0xbc, 0xcc, 0x21, 0xad, 0x06,
        0x64, 0x8d, 0x09, 0xa5, 0xc8, 0xcf, 0x5a, 0x78,
        0x48, 0x20, 0x54, 0xe9, 0x14, 0x70, 0xb3, 0x3d,
        0xde, 0x77, 0x25, 0x2c, 0xae, 0xf6, 0x95, 0xaa },
      { 0x00, 0x00, 0x06, 0x31, 0x17, 0x5f, 0xa7, 0xae,
        0x64, 0x3a, 0xd0, 0x87, 0x23, 0xd3, 0x12, 0xc9,
        0xfd, 0x02, 0x4a, 0xdb, 0x91, 0xf7, 0x7f, 0x6b,
        0x19, 0x58, 0x71, 0x97, 0xa2, 0x2b, 0xcd, 0xf2,
        0x37, 0x27, 0x16, 0x6c, 0x45, 0x72, 0xd0, 0xb9,
```

```
    0x85, 0xd5, 0xae, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x42,
    0x70, 0xd1, 0x1e, 0xf4, 0x18, 0xef, 0x08, 0xb8,
    0xa4, 0x9e, 0x1e, 0x82, 0x5e, 0x54, 0x7e, 0xb3,
    0x99, 0x37, 0xf8, 0x19, 0x22, 0x2f, 0x3b, 0x7f,
    0xc9, 0x2a, 0x0e, 0x47, 0x07, 0x90, 0x08, 0x88,
    0x84, 0x7a, 0x55, 0x4b, 0xac, 0xec, 0x98, 0xb0 },
  { 0xf6, 0xf7, 0x47, 0xaf, 0x25, 0xa9, 0xde, 0x26,
    0xe8, 0xa4, 0x93, 0x43, 0x1e, 0x31, 0xb4, 0xa1,
    0xed, 0x2a, 0x92, 0x30, 0x4a, 0xf6, 0xca, 0x97,
    0x6b, 0xc1, 0xd9, 0x6f, 0xfc, 0xad, 0x35, 0x24,
    0x4e, 0x8d, 0x38, 0x5d, 0x55, 0xf4, 0x2f, 0xdc,
    0xc8, 0xf2, 0x99, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0xf7, 0xca, 0x87, 0xce,
    0x43, 0x22, 0x7b, 0x98, 0xc1, 0x44, 0x60, 0x7e,
    0x67, 0xcc, 0x50, 0xaf, 0x99, 0xbc, 0xc5, 0xd1,
    0x51, 0x4b, 0xb0, 0xd9, 0x23, 0xee, 0xde, 0xdd,
    0x69, 0xe8, 0xe7, 0x47, 0x02, 0x05, 0x08, 0x3a,
    0x0c, 0x02, 0x27, 0xd0, 0xcc, 0x69, 0xde, 0x23 } };
uint8_t   FNV1024bvalues[NTstrings][FNV1024size] = {
  { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x98, 0xd7, 0xc1, 0x9f, 0xbc, 0xe6, 0x53, 0xdf,
    0x22, 0x1b, 0x9f, 0x71, 0x7d, 0x34, 0x90, 0xff,
    0x95, 0xca, 0x87, 0xfd, 0xae, 0xf3, 0x0d, 0x1b,
    0x82, 0x33, 0x72, 0xf8, 0x5b, 0x24, 0xa3, 0x72,
    0xf5, 0x0e, 0x38, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x07, 0x68, 0x5c, 0xd8,
    0x1a, 0x49, 0x1d, 0xbc, 0xcc, 0x21, 0xad, 0x06,
    0x64, 0x8d, 0x09, 0xa5, 0xc8, 0xcf, 0x5a, 0x78,
    0x48, 0x20, 0x54, 0xe9, 0x14, 0x70, 0xb3, 0x3d,
    0xde, 0x77, 0x25, 0x2c, 0xae, 0xf6, 0x65, 0x97 },
  { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xf4,
    0x6e, 0xf4, 0x1c, 0xd2, 0x3a, 0x4d, 0xcd, 0xd4,
    0x06, 0x83, 0x49, 0x63, 0xb7, 0x8e, 0x82, 0x24,
    0x1a, 0x6f, 0x5c, 0xb0, 0x6f, 0x40, 0x3c, 0xbd,
    0x5a, 0x7c, 0x89, 0x03, 0xce, 0xf6, 0xa5, 0xf4,
```

```
    0xfd, 0xd2, 0x95, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x0b, 0x7c, 0xd7, 0xfb, 0x20,
    0xc3, 0x63, 0x1d, 0xc8, 0x90, 0x39, 0x52, 0xe9,
    0xee, 0xb7, 0xf6, 0x18, 0x69, 0x8f, 0x4c, 0x87,
    0xda, 0x23, 0xad, 0x74, 0xb2, 0xc5, 0xf6, 0xf1,
    0xfe, 0xc4, 0xa6, 0x4b, 0x54, 0x66, 0x18, 0xa2 },
  { 0x00, 0x09, 0xdc, 0x92, 0x10, 0x75, 0xfd, 0x8a,
    0x5e, 0x3e, 0x1a, 0x37, 0x2c, 0x72, 0xa5, 0x9b,
    0xb1, 0x0c, 0xca, 0x1a, 0x94, 0xc8, 0xb2, 0x38,
    0x7d, 0x63, 0xa7, 0xef, 0xa7, 0xfc, 0xa7, 0xa7,
    0x17, 0xa6, 0x4e, 0x6c, 0x2d, 0x62, 0xfb, 0x61,
    0x78, 0xf7, 0x86, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x67, 0x08,
    0xf4, 0x4d, 0x00, 0x8a, 0xaa, 0xb0, 0x86, 0x57,
    0x49, 0x35, 0x50, 0x2c, 0x49, 0x08, 0x7c, 0x84,
    0x9b, 0xcb, 0xbe, 0xfa, 0x03, 0x3f, 0x45, 0x2a,
    0xf6, 0x38, 0x24, 0x26, 0xba, 0x5d, 0x3b, 0xb5,
    0x71, 0xb6, 0x46, 0x5b, 0x2a, 0xe8, 0xc8, 0xf0 },
  { 0xc8, 0x01, 0xf8, 0xe0, 0x8a, 0xe9, 0x1b, 0x18,
    0x0b, 0x98, 0xdd, 0x7d, 0x9f, 0x65, 0xce, 0xb6,
    0x87, 0xca, 0x86, 0x35, 0x8c, 0x69, 0x05, 0xf6,
    0x0a, 0x7d, 0x10, 0x14, 0xc1, 0x82, 0xb0, 0x4f,
    0xd6, 0x08, 0xa2, 0xca, 0x4d, 0xd6, 0x0a, 0x30,
    0x0a, 0x15, 0x68, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x01, 0x80, 0x45, 0x14, 0x9a, 0xde,
    0x1c, 0x79, 0xab, 0xe3, 0xb7, 0x09, 0xa4, 0x06,
    0xf7, 0xd9, 0x20, 0x51, 0x69, 0xbe, 0xc5, 0x9b,
    0x12, 0x61, 0x40, 0xbc, 0xb9, 0x6f, 0x9d, 0x5d,
    0x3e, 0x2e, 0xa9, 0x1e, 0x21, 0xcd, 0xc2, 0x04,
    0x9f, 0x57, 0xbe, 0xcd, 0x00, 0x2d, 0x7c, 0x47 } };

funcName = "FNV-1024";
/* test error checks */
Terr = 0;
TestR ( "init1", fnvSuccess, FNV1024init (&e1024Context) );
TestR ( "string1", fnvNull,
```

```
            FNV1024string ( (char *)0, Vec1024 ) );
    TestR ( "string2", fnvNull,
            FNV1024string ( errteststring, (uint8_t *)0 ) );
    TestR ( "block1", fnvNull,
            FNV1024block ( (uint8_t *)0, 1, Vec1024 ) );
    TestR ( "block2", fnvBadParam,
            FNV1024block ( errtestbytes, -1, Vec1024 ) );
    TestR ( "block3", fnvNull,
            FNV1024block ( errtestbytes, 1, (uint8_t *)0 ) );
    TestR ( "init2", fnvNull,
            FNV1024init ( (FNV1024context *)0 ) );
    TestR ( "initBasis1", fnvNull,
            FNV1024initBasis ( (FNV1024context *)0, Vec1024 ) );
    TestR ( "blockin1", fnvNull,
            FNV1024blockin ( (FNV1024context *)0,
                             errtestbytes, NTestBytes ) );
    TestR ( "blockin2", fnvNull,
            FNV1024blockin ( &e1024Context, (uint8_t *)0,
                             NTestBytes ) );
    TestR ( "blockin3", fnvBadParam,
            FNV1024blockin ( &e1024Context, errtestbytes, -1 ) );
    e1024Context.Computed = FNVclobber+FNV1024state;
    TestR ( "blockin4", fnvStateError,
            FNV1024blockin ( &e1024Context, errtestbytes,
                             NTestBytes ) );
    TestR ( "stringin1", fnvNull,
            FNV1024stringin ( (FNV1024context *)0, errteststring ) );
    TestR ( "stringin2", fnvNull,
            FNV1024stringin ( &e1024Context, (char *)0 ) );
    TestR ( "stringin3", fnvStateError,
            FNV1024stringin ( &e1024Context, errteststring ) );
    TestR ( "result1", fnvNull,
            FNV1024result ( (FNV1024context *)0, Vec1024 ) );
    TestR ( "result2", fnvNull,
            FNV1024result ( &e1024Context, (uint8_t *)0  ) );
    TestR ( "result3", fnvStateError,
            FNV1024result ( &e1024Context, Vec1024  ) );
    ErrTestReport ();
/* test actual results */
    Terr = 0;
    for ( i = 0; i < NTstrings; ++i ) {
        err = TestR ( "stringa", fnvSuccess,
                      FNV1024string ( teststring[i], Vec1024 ) );
        if ( err == fnvSuccess )
            TestNValue ( "stringb", teststring[i], FNV1024size,
                            Vec1024,
                        FNV1024svalues[i] );
        err = TestR ( "blocka", fnvSuccess,
                      FNV1024block ( (uint8_t *)teststring[i],
```

```
                        (unsigned long)(strlen(teststring[i])+1),
                        Vec1024 ) );
        if ( err == fnvSuccess )
            TestNValue ( "blockb", teststring[i], FNV1024size,
                    Vec1024,
                    FNV1024bvalues[i] );
/* now try testing the incremental stuff */
        err = TestR ( "inita", fnvSuccess,
                     FNV1024init ( &e1024Context ) );
        if ( err ) break;
        iLen = strlen ( teststring[i] );
        err = TestR ( "blockina", fnvSuccess,
                     FNV1024blockin ( &e1024Context,
                     (uint8_t *)teststring[i],
                     iLen/2 ) );
        if ( err ) break;
        err = TestR ( "stringina", fnvSuccess,
                     FNV1024stringin ( &e1024Context,
                     teststring[i] + iLen/2 ) );
        if ( err ) break;
        err = TestR ( "resulta", fnvSuccess,
                     FNV1024result ( &e1024Context, Vec1024 ) );
        if ( err == fnvSuccess )
            TestNValue ( "incrementala", teststring[i], FNV1024size,
                    Vec1024,
                    (uint8_t *)&FNV1024svalues[i] );
        }
    ValueTestReport ();
}    /* 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 use of a different offset_basis or 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 bits. 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 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 non-cryptographic 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 reduce this problem.

8. IANA Considerations

This document requires no IANA Actions.

9. Normative References

[RFC0020] Cerf, V., "ASCII format for network interchange", STD 80, RFC 20, DOI 10.17487/RFC0020, October 1969, <https://www.rfc-editor.org/info/rfc20>.

10. Informative References

[BFDseq] Jethanandani, M., Agarwal, S., Mishra, A., Saxena, A., and A. DeKok, "Secure BFD Sequence Numbers", 22 March 2022, <draft-ietf-bfd-secure-sequence-numbers-09.txt>.

[FNV] Fowler-Noll-Vo, "FNV website", <http://www.isthe.com/chongo/tech/comp/fnv/index.html>.

[IEEE] Institute for Electrical and Electronics Engineers, "IEEE website", <http:www.ieee.org>.

[IEEE8021Qbp] "Media Access Control (MAC) Bridges and Virtual Bridged Local Area Networks - Equal Cost Multiple Path (ECMP)", IEEE Std 802.1Qbp-2014, 7 April 2014.

[IPv6flow] Anderson, L., Brownlee, N., and B. Carpenter, "Comparing Hash Function Algorithms for the IPv6 Flow Label", University of Auckland Department of Computer Science Technical Report 2012-002, ISSN 1173-3500, March 2012, <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>.

[RFC7357] Zhai, H., Hu, F., Perlman, R., Eastlake 3rd, D., and O. Stokes, "Transparent Interconnection of Lots of Links

(TRILL): End Station Address Distribution Information (ESADI) Protocol", RFC 7357, DOI 10.17487/RFC7357, September 2014, <https://www.rfc-editor.org/info/rfc7357>.

[RFC7873] Eastlake 3rd, D. and M. Andrews, "Domain Name System (DNS) Cookies", RFC 7873, DOI 10.17487/RFC7873, May 2016, <https://www.rfc-editor.org/info/rfc7873>.

[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>.

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 a 4-byte input such as an IPv4 address, SHA-1 will take 218 times as much effort as FNV, and wih a 6-byte input such as a MAC address, SHA-1 will take a little over 145 times as much effort as FNV.

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

Appendix B. Previous IETF FNV Code

FNV-1a was referenced in draft-ietf-tls-cached-info-08.txt that has since expired. Below is the Java code for FNV64 from that TLS draft included with 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. Change History

RFC Editor Note: Please delete this appendix on publication.

C.1. 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.

C.2. 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.

C.3. From -02 to -03

Minor editing changes.

C.4. From -03 to -04

Minor addition to Section 6, point 3. Update dates and version number. Minor editing changes.

C.5. From -04 to -05

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

2. Update dates and version number.

C.6. From -05 to -06

1. Add code subsections.

2. Update dates and version number.

C.7. From -06 to -07 to -08

1. Update Author info.

2. Minor edits.

C.8. From -08 to -09

1. Change reference for ASCII to [RFC0020].

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.

C.9. 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.

C.10. 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.

C.11. From -11 to -12

Keep alive update.

C.12. 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.

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

Keep alive updates.

Update an author address. Update reference. Update author affiliation.

C.14. From -17 to -18 to -19

Add reference to draft-ietf-bfd-secure-sequence-numbers. Add references to the following, each of which uses FNV: RFC 7357, RFC 7873, and IEEE Std. 802.1Qbp-2014. Update author info. Minor editorial changes.

C.15. From -19 to -20

Convert to XML v3. Fix code for loger FNV hashes.

C.16. From -20 to -21

Update Twitter to X. Minor Editorial changes.

C.17. From -21 to -22

/Update Landon's email. Minor Editorial changes. Update to substantially improved code.

Acknowledgements

The contributions of the following are gratefully acknowledged:

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

Authors' Addresses

Glenn S. Fowler
Google

Email: glenn.s.fowler@gmail.com

Landon Curt Noll
Cisco Systems
170 West Tasman Drive
San Jose, California 95134
United States of America

Phone: +1-408-424-1102
Email: fnv-ietf6-mail@asthe.com
URI: http://www.isthe.com/chongo/index.html

Kiem-Phong Vo
Google

Email: phongvo@gmail.com

Donald E. Eastlake 3rd
Independent
2386 Panoramic Circle
Apopka, Florida 32703
United States of America

Phone: +1-508-333-2270
Email: d3e3e3@gmail.com

Tony Hansen
AT&T Laboratories
200 Laurel Avenue South
Middletown, New Jersey 07748
United States of America

Email: tony@att.com