Stream:	Independent Submission
RFC:	9923
Category:	Informational
Published:	February 2026
ISSN:	2070-1721

Authors: L. Noll K. Vo *Google* D. Eastlake 3rd *Independent* T. Hansen *AT&T*

RFC 9923
The FNV Non-Cryptographic Hash Algorithm

Abstract

FNV (Fowler/Noll/Vo) is a fast, non-cryptographic hash algorithm with good dispersion that has been widely used and is referenced in a number of standards documents. The purpose of this document is to make information on FNV and open-source code performing all specified sizes of FNV conveniently available to the Internet community.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

FNV (Fowler/Noll/Vo) hashes are designed to be fast and have a small code footprint. Their good dispersion makes them particularly well suited for hashing nearly identical strings, including URLs, hostnames, filenames, text, and IP and Media Access Control (MAC) addresses. Their speed allows one to quickly hash lots of data.

The purpose of this document is to make information on FNV and open-source code performing all specified sizes of FNV conveniently available to the Internet community. This work is not an Internet Standard and does not have the consensus of the IETF community.

1.1. Conventions Used in This Document

The key words "**MUST**", "**MUST NOT**", "**REQUIRED**", "**SHALL**", "**SHALL NOT**", "**SHOULD**", "**SHOULD NOT**", "**RECOMMENDED**", "**NOT RECOMMENDED**", "**MAY**", and "**OPTIONAL**" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

1.2. Applicability of Non-Cryptographic Hashes and FNV

While a general theory of hash function strength and utility is beyond the scope of this document, typical attacks on hash functions involve one of the following:

Collision: Finding two data inputs that yield the same hash output.

First Pre-Image: Given a hash output, finding a data input that hashes to that output.

Second Pre-Image: Given a first data input, finding a second input that produces the same hash output as the first.

For a hash function producing N bits, there necessarily will be collisions among the hashes of more than 2^N distinct inputs. And if the hash function can produce hashes covering all 2^N possible outputs, then there will exist first and second pre-images. FNV is **NOT RECOMMENDED** for any application that requires that it be computationally infeasible for one of the above types of attacks to succeed.

FNV hashes are generally not applicable for use when faced with an active adversary in a security scheme where the modest effort required to compute FNV hashes (see Appendix A) and their other non-cryptographic characteristics (see Section 1.4) would make the scheme ineffective against the threat model being considered. It is sometimes hard to determine whether or not there are attack vectors via a hash.

For a discussion of adversarial inducement of collisions, see Section 6.1.

1.3. FNV Hash Uses

The FNV hash has been widely used. Examples include the following:

- NFS implementations (e.g., FreeBSD 4.3 [FreeBSD], IRIX, Linux (NFS v4)),
- text-based referenced resources for video games on the PS2, Gamecube, and XBOX,
- to improve the fragment cache [FragCache] at X (formerly Twitter),
- the flatassembler open-source x86 assembler - user-defined symbol hashtree [flatassembler],
- used in the speed-sensitive guts of [twistylists], an open-source structured namespace manager,
- database indexing hashes,
- PowerBASIC inline assembly routine [BASIC],
- major web search / indexing engines,
- the "calc" C-style calculator [calc],
- netnews history file Message-ID lookup functions,
- [FRET] - a tool to identify file data structures / help understand file formats,
- anti-spam filters,
- used in an implementation of libketama [libketama] for use in items such as [memcache],
- a spellchecker programmed in Ada 95,
- used in the BSD Integrated Development Environments (IDE) project [fasmlab],
- non-cryptographic file fingerprints,
- used in the deliantra game server for its shared string implementation [deliantra],
- 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),

- DNS (Domain Name System) servers,
- used to improve [Leprechaun], an extremely fast wordlist creator,
- the Smash utility [Smash] for rapidly finding duplicate files,
- Golf language hash tables [RimStone],
- the libsir logging library [libsir],
- a standard library for modern Fortran [Fortran],

and many other uses. It is also referenced in the following standards documents: [RFC7357], [RFC7873], and [IEEE8021Q-2022].

A study has recommended FNV in connection with the IPv6 flow label value [IPv6flow]. Additionally, there was a proposal to use FNV for Bidirectional Forwarding Detection (BFD) sequence number generation [BFDseq]. [NCHF] discusses criteria for evaluating non-cryptographic hash functions.

If you use an FNV function in an application, you are kindly requested to send a note via the process outlined at <http://www.isthe.com/chongo/tech/comp/fnv/index.html#address>.

1.4. 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 accidentally becomes zero and the input is a sequence of zero bytes, 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. 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, under some circumstances, to stop an adversary from inducing a zero hash variable (see Section 6.1).
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 applied to slow down testing of possible inputs if the output is known or the like. 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 (see Section 1.3).

2. FNV Basics

This document focuses on the FNV-1a function, whose pseudocode is as follows:

```
hash = offset_basis
for each octet_of_data to be hashed
    hash = hash XOR octet_of_data
    hash = hash * FNV_Prime mod 2**HashSize
return hash
```

In the pseudocode above, hash is a power-of-2 number of bits (HashSize is 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 is 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_Primes 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 follows:

- When s is an integer and 4 < s < 11, 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 four or five
- and where

```
( p mod (2**40 - 2**24 - 1) ) > ( 2**24 + 2**8 + 2**7 )
```

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

The case where $s < 5$ is not considered 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 would be 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 six or seven one bits in it: one relatively high-order one bit, the 2^9 bit, and four or five one bits in the low-order byte. 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, 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 following 32-octet ASCII [RFC0020] character string:

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

or, in C notation [C], the following string:

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

In the general case, almost any offset_basis would serve as long as it is non-zero. However, FNV hashes calculated with different offset_basis values will not interoperate. The choice of a non-standard offset_basis may be beneficial in some limited circumstances to defend against attacks that try to induce hash collisions as discussed in Section 6.1. Any entity that can observe the FNV hash output and can cause the null string (the string of length zero) to be hashed will thereby be able to directly observe the offset_basis which will be the hash output.

2.3. FNV Endianism

For persistent storage or interoperability between different hardware platforms, an FNV hash shall be represented in the little-endian format [IEN137]. 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];
```

However, when FNV hashes are used in a single process or a group of processes sharing memory on processors with compatible endianness, the natural endianness of those processors can be used, as long as it is used consistently, regardless of its type -- little, big, or some other exotic form.

The code provided in Section 8 has FNV hash functions that return a little-endian byte vector for all lengths. Because they are more efficient, the code also provides functions that return FNV hashes as 32-bit integers or, where supported, 64-bit integers, for those sizes of FNV hash. Such integers are compatible with the same-size byte vectors on little-endian computers, but the 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 vector return values.

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 constants provided 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 )
```

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 XOR data folding using a k equal to the size of the desired output. However, if a much stronger hash is desired, cryptographic algorithms, such as those specified in [FIPS202] or [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 not a power of 2, 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 slightly larger 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

Sometimes, there are multiple 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. If the components being combined are of variable length, some information is lost by simple concatenation. For example, X = "12" and Y = "345" would not be distinguished from X = "123" and Y = "45". To preserve that information, each component should be preceded by an encoding of its length or should end with some sequence that cannot occur within the component, or some similar technique should be used.

For FNV, the same hash results if 1) X, Y, and Z are actually concatenated and the FNV hash is applied to the resulting string or 2) FNV is calculated on an initial substring and the result is used as the offset_basis when calculating the FNV hash of the remainder of the string. This can be done several times. Assuming that FNVoffset_basis (v, w) is the 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).

This means that if you have the value of FNV (X) and you want to calculate FNV (X | Y), you do not need to find X. You can simply calculate FNVoffset_basis (FNV (X), Y) and thereby get FNV (X | Y).

Sometimes, such a hash needs to be repeatedly calculated; the component values vary, but some vary more frequently than others. For example, assume that some sort of computer network traffic flow ID, such as the IPv6 Flow Label [RFC6437], is to be calculated for network packets based on the source and destination IPv6 addresses and the Traffic Class [RFC8200]. If the Flow Label is calculated in the originating host, the source IPv6 address would likely always be the same or would 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 the 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 source code in this document includes functions facilitating the use of a non-standard offset_basis.

An alternative method of hashing multiple values is to concatenate the hashes of those values and then hash the concatenation -- that is, compute something like

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

This will involve more computation than simply computing the hash of the concatenation of the values and thus, unless parallel computational resources are available, greater latency; however, if parallel computational resources are available and the values being hashed together are long enough to overcome any initial/final hash function overhead, which is very small for FNV, latency can be reduced by hashing the concatenation of the hashes of the values.

For another example of a similar technique, assume a desire to use FNV-N to hash a byte string of length L. Let B = N/8, the number of bytes of FNV-N output. If that string is divided into k successive substrings of equal length and assuming, for simplicity, that L is an integer multiple of k, hashing the substrings and then hashing the concatenation of their hashes will hash a total of L + k*B bytes, clearly more than the initial string size L. However, if sufficient parallel computational resources are available to hash all the substrings simultaneously, the elapsed time can be changed approximately from on the order of L to on the order of L/k + k*B. For sufficiently large L, this parallelization will reduce the elapsed time to produce the overall hash.

5. FNV Constants

The FNV Primes are as follows:

Size FNV Prime = Expression = 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

Size FNV Prime = Expression **= Decimal** **= Hexadecimal**
= 374,144,419,156,711,147,060,143,317,175,368,453,031,918,731,002,211
= 0x0000000000000000 0000010000000000 0000000000000000 0000000000000163
512-bit FNV_Prime = $2^{344} + 2^{8} + 0x57$
= 35,835,915,874,844,867,368,919,076,489,095,108,449,946,327,955,754,392,558,399,825,615,420, 669,938,882,575,126,094,039,892,345,713,852,759
= 0x0000000000000000 0000000000000000 0000000001000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000157
1024-bit FNV_Prime = $2^{680} + 2^{8} + 0x8D$
= 5,016,456,510,113,118,655,434,598,811,035,278,955,030,765,345,404,790,744,303,017,523,831, 112,055,108,147,451,509,157,692,220,295,382,716,162,651,878,526,895,249,385,292,291,816,524, 375,083,746,691,371,804,094,271,873,160,484,737,966,720,260,389,217,684,476,157,468,082,573
= 0x0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000010000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000 000000000000018D

Table 1

The FNV offset_basis values are as follows:

Size offset_basis **= Decimal** **= Hexadecimal**
32-bit offset_basis
= 2,166,136,261
= 0x811C9DC5
64-bit offset_basis
= 14,695,981,039,346,656,037

Size offset_basis **= Decimal** **= Hexadecimal**
= 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. Security Considerations

No assertion of suitability for cryptographic applications is made for the FNV hash algorithms.

The use of a cryptographic hash function should be considered when active adversaries are a factor (see Section 1.2).

6.1. Inducing Collisions

An attacker could attempt to induce collisions to cause denial or degradation of service. Consider the following simplified example: A hash table of n buckets is being maintained with the bucket used by some item i determined by

```
hash(i) mod n
```

and with a linked list out of each bucket of the items that all hash to that bucket. Such an arrangement might be used for the symbol table in a compiler or for some of the routing information (i.e., a RIB (Routing Information Base)) in a router. A large number of items hashing to the same bucket will then likely result in much slower times to retrieve from or update the information stored through the table for one of those items. Typically, an attacker could arrange for the number of distinct items being hashed to be orders of magnitude larger than n, even if n was tens or hundreds of thousands, so collisions are guaranteed to occur in this example regardless of the nature of the hash function.

There are a number of different circumstances that might surround this example, of which the following three are illustrative:

- If a hash function is being used in an exactly known way for the above scenario, including a known offset_basis such as a standard offset_basis specified in this document, then an adversary could test items offline and generate an arbitrary set of items whose hash table indexes would collide. Under these circumstances, the adversary would not have to conduct any trials of actually submitting items and would not have to measure performance to find collisions. Submitting such a set of items would then degrade or deny service. For FNV, the use of an offset_basis not known by the adversary is adequate to defeat this case.
- If the adversary cannot detect when collisions occur or when service is degraded, then it is sufficient for the adversary to be unable to predict the hash outcomes. For FNV, the use of an offset_basis not known by the adversary may be adequate to defend against this case.
- If the adversary can detect the degradation in service caused by collisions in the above example and can feed large numbers of variable items to the process, then they can collect sets of items that appear to collide. Even if there are limits to the number of items that can be submitted, if there can be multiple trials, the adversary can collect multiple sets of items that collide within each set or one growing set of items, all of which collide. Then, by submitting such items, the adversary can degrade or deny service. That is true regardless of whether the hash function used is a non-cryptographic hash function such as FNV or a cryptographic hash function such as those specified in [FIPS202] or [RFC6234]. One defense in this case is to detect when a large number of collisions are happening (which could, but would be unlikely to, occur by chance) and, when that is detected, rehash the items with some change to the hash algorithm and use the changed hash algorithm for subsequent items -- for example, if FNV is being used, to rehash with a different offset_basis and then

continue using that new offset_basis. There exist commercially deployed routers that use this technique to ameliorate excessive hash collisions in internal tables.

7. Historical Notes

The FNV hash algorithm originated from an idea submitted as reviewer comments to the IEEE POSIX P1003.2 committee [IEEE] 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 from their last names in alphabetical order [FNV].

The string used to calculate the offset_basis values (see Section 2.2) was selected 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 "did not see very well" [FNV] and copied it incorrectly. In fact, Landon uses

```
chongo (Landon Curt Noll) /\oo/\
```

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

8. The Source Code

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

Section 8.2 provides the C header and code source files for the FNV functions. Section 8.3 provides the test driver. Section 8.4 provides a simple makefile to build the test driver or a library file with all FNV sizes.

Alternative source code for 32- and 64-bit FNV is available at [LCN2]. Other alternative source code, including in x86 assembler, is currently available at [FNV]. In some cases, this further source code has been further optimized.

8.1. Source Code Details

8.1.1. FNV Functions Available

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. All of the FNV hash functions have as their return value an integer whose meaning is specified in FNVErrorCodes.h.

Functions providing a byte vector hash are available for all lengths. For FNV-32, versions are available that provide a 32-bit integer and are identified by replacing "xxx" with "32INT". For example, FNV32string provides a 4-byte vector, but FNV32INTstring provides a 32-bit integer. For FNV-64, if compiled with 64-bit integers enabled (i.e., FNV_64bitIntegers defined; see

FNVconfig.h), versions are available that provide a 64-bit integer and are identified by replacing "xxx" with "64INT". Versions providing an integer hash will not be compatible between systems of different endianness (see Section 2.3).

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, FNVxxxfile:

FNVxxxstringBasis, FNVxxxblockBasis, FNVxxxfileBasis:

FNVxxxINTstring, FNVxxxINTblock, FNVxxxINTfile:

FNVxxxINTstringBasis, FNVxxxINTblockBasis, FNVxxxINTfileBasis: These are simple functions for directly returning the FNV hash of a zero-terminated byte string not including that zero byte, the FNV hash of a counted block of bytes, and the FNV of a file, respectively. The functions whose names have the "Basis" suffix take an additional parameter specifying the offset_basis. Note that for applications of FNV-32 and of FNV-64 where integers of that size are supported and an integer data type output is acceptable, the code is sufficiently simple that, to maximize performance, the use of open coding or macros may be more appropriate than calling a subroutine.

FNVxxxinit, FNVxxxinitBasis:

FNVxxxINTinitBasis: 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, FNVxxxfilein: 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 including the final zero byte. FNVxxxfilein hashes in the contents of a file.

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

8.1.2. Source Files and 64-Bit Support

Code optimized for 64-bit integer support -- that is, support of 64-bit integer addition and 32-bit x 32-bit multiplication producing a 64-bit product -- is provided based on whether or not the FNV_64bitIntegers symbol is defined. By default, this is set in FNVconfig.h based on the compilation target; however, this can be overridden by editing that file or by defining certain symbols in, for example, a command line invoking a compilation.

For support of a single FNV size, say "xxx" (e.g., FNV64), in an application, the application itself needs to include the appropriate FNVxxx.h file (which will, in turn, include the FNVconfig.h and FNVErrorCodes.h files). To build the particular FNVxxx code itself, compile the FNVxxx.c file with the following files available to the compiler: FNVconfig.h, fnv-private.h, FNVErrorCodes.h, and FNVxxx.h. (See Section 8.2.) Since the test program provided in Section 8.3 uses all sizes of FNV, all the .c and .h files are needed to compile it.

8.1.3. Command Line Interface

The test program provided in Section 8.3 has a command line interface. By default, with no command line arguments, it runs tests of all FNV lengths. Command line options are as follows:

```
   FNVhash [-a] [-h] [-t nnn] [-u nnn] [-v] [-f filename] [token ...]
```

The option letters have the following meanings:

-a Run tests for all lengths.

-h Print a help message about the command line.

-v Complement the Verbose flag, which is initially off. When the flag is on, the program prints more information during tests, etc.

-t nnn Run tests for length nnn, which must be one of 32, 64, 128, 256, 512, or 1024.

-u nnn Use hash size nnn, which must be one of 32, 64, 128, 256, 512, or 1024. This is useful for setting the hash size for use by the -f option or in hashing tokens on the command line after the options.

-f filename Hash the contents of the file with name filename. The hash size must have been set by a prior -t or -u option in the command line.

token Tokens appearing on the command line after the options are hashed with the current hash size, which must have been set by a prior -t or -u option in the command line.

For example,

```
      FNVhash -t 128 -h -v -t 64 -v -u 256 -f foobar.txt RabOof 1234
```

runs tests for FNV128, then prints a command line help message, then turns on Verbose, then runs the tests for FNV64, then turns off Verbose, then sets the hash size to 256, then hashes the contents of file foobar.txt, then hashes the token "RabOof", and finally hashes the token "1234".

8.2. 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 following is a configuration header to set whether 64-bit integers are supported and establish an enum used for return values.

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

//************************ FNVconfig.h **************************//
//**************** See RFC 9923 for details. ********************//
/* Copyright (c) 2016-2025 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. */

/*      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.
 */

#include <stdint.h>

/* Check if 64-bit integers are supported in the target */

#ifdef UINT64_MAX
    #define FNV_64bitIntegers
#else
    #undef FNV_64bitIntegers
#endif

/*      The following allows overriding the
 *      above configuration setting.
 */

#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
#endif

#endif /* _FNVconfig_H_ */
```

```
<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 9923 for details. **********************//
/* Copyright (c) 2016-2025 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, except the FNVxxxfile functions,
//    return an integer as follows:
//       0 -> success
//      >0 -> error as listed below
//
enum {    /* success and errors */
    fnvSuccess = 0,
    fnvNull,        // 1 Null pointer parameter
    fnvStateError,  // 2 called Input after Result or before Init
    fnvBadParam     // 3 passed a bad parameter
};
#endif /* _FNV_ErrCodes_ */

<CODE ENDS>
```

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

```
<CODE BEGINS> file "fnv-private.h"

//************************ fnv-private.h **************************//
//****************** See RFC 9923 for details. ********************//
/* Copyright (c) 2016-2025 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 /* _FNV_PRIVATE_H_ */
```

```
<CODE ENDS>
```

8.2.1. FNV32 Code

The following code is the header and C source for 32-bit FNV-1a providing a 32-bit integer or 4-byte vector hash.

```
<CODE BEGINS> file "FNV32.h"

//************************** FNV32.h ****************************//
//****************** See RFC 9923 for details. *******************//
/* Copyright (c) 2016-2025 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_

/*
 *  Description:
 *      This file provides headers for the 32-bit version of
 *      the FNV-1a non-cryptographic hash algorithm.
 */

#include "FNVconfig.h"
#include "FNVErrorCodes.h"

#include <stdint.h>
#define FNV32size (32/8)
#define FNV32basis 0x811C9DC5

/* 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)
 */

/*
 *  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
 *    FNV32stringBasis: also takes an offset_basis parameter
 *
 *    FNV32block: hash a byte vector of a specified length
 *    FNV32blockBasis: also takes an offset_basis parameter
 *
 *    FNV32file: hash the contents of a file
 *    FNV32fileBasis: also takes an offset_basis parameter
 *
 *    FNV32init:  initializes an FNV32 context
 *    FNV32initBasis: initializes an FNV32 context with a
 *                    provided 4-byte vector basis
 *    FNV32blockin:  hash in a byte vector of a specified length
 *    FNV32stringin: hash in a zero-terminated string not
 *                   including the terminating zero
 *    FNV32filein: hash in the contents of a file
 *    FNV32result: returns the hash value
 *
 * Hash is returned as a 4-byte vector by the functions above,
 *    and the following return a 32-bit unsigned integer:
 *
 *    FNV32INTstring: hash a zero-terminated string not including
 *                 the terminating zero
 *    FNV32INTstringBasis: also takes an offset_basis parameter
 *
 *    FNV32INTblock: hash a byte vector of a specified length
 *    FNV32INTblockBasis: also takes an offset_basis parameter
 *
 *    FNV32INTfile: hash the contents of a file
 *    FNV32INTfileBasis: also takes an offset_basis parameter
 *
 *    FNV32INTinitBasis: initializes an FNV32 context with a
 *                     provided 32-bit integer basis
 *    FNV32INTresult: returns the hash value
 */

#ifdef __cplusplus
extern "C" {
#endif

/* FNV32 */
extern int FNV32INTstring ( const char *in,
                            uint32_t * const out );
extern int FNV32INTstringBasis ( const char *in,
                                 uint32_t * const out,
                                 uint32_t basis );
extern int FNV32string ( const char *in,
                         uint8_t out[FNV32size] );
extern int FNV32stringBasis ( const char *in,
                              uint8_t out[FNV32size],
                              const uint8_t basis[FNV32size] );
extern int FNV32INTblock ( const void *vin,
                           long int length,
                           uint32_t * const out );
extern int FNV32INTblockBasis ( const void *vin,
                                long int length,
                                uint32_t * const out,
                                uint32_t basis );
extern int FNV32block ( const void *vin,
                        long int length,
                        uint8_t out[FNV32size] );
extern int FNV32blockBasis ( const void *vin,
                             long int length,
                             uint8_t out[FNV32size],
                             const uint8_t basis[FNV32size] );
extern int FNV32INTfile ( const char *fname,
                          uint32_t * const out );
extern int FNV32INTfileBasis ( const char *fname,
                               uint32_t * const out,
                               uint32_t basis );
extern int FNV32file ( const char *fname,
                       uint8_t out[FNV32size] );
extern int FNV32fileBasis ( const char *fname,
                            uint8_t out[FNV32size],
                            const uint8_t basis[FNV32size] );
extern int FNV32init ( FNV32context * const );
extern int FNV32INTinitBasis ( FNV32context * const,
                               uint32_t basis );
extern int FNV32initBasis ( FNV32context * const,
                            const uint8_t basis[FNV32size] );
extern int FNV32blockin ( FNV32context * const,
                          const void *vin,
                          long int length );
extern int FNV32stringin ( FNV32context * const,
                           const char *in );
extern int FNV32filein ( FNV32context * const,
                         const char *fname );
extern int FNV32INTresult ( FNV32context * const,
                            uint32_t * const out );
extern int FNV32result ( FNV32context * const,
                         uint8_t out[FNV32size] );

#ifdef __cplusplus
}
#endif

#endif /* _FNV32_H_ */

<CODE ENDS>
```

```
<CODE BEGINS> file "FNV32.c"

//*************************** FNV32.c **************************//
//***************** See RFC 9923 for details. ******************//
/* Copyright (c) 2016-2025 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.
 */

#include <stdio.h>

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

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

/* FNV32: hash a zero-terminated string not including the zero
******************************************************************/
int FNV32INTstring ( const char *in, uint32_t * const out ) {
    return FNV32INTstringBasis ( in, out, FNV32basis );
}   /* end FNV32INTstring */

/* FNV32: hash a zero-terminated string not including the zero
 * with a non-standard basis
******************************************************************/
int FNV32INTstringBasis ( const char *in,
                          uint32_t * const out,
                          uint32_t basis ) {
    uint8_t     ch;

    if ( !in || !out )
        return fnvNull; /* Null input pointer */
    while ( (ch = *in++) )
        basis = FNV32prime * ( basis ^ ch );
    *out = basis;
    return fnvSuccess;
}   /* end FNV32INTstringBasis */

/* FNV32: hash a zero-terminated string not including the zero
******************************************************************/
int FNV32string ( const char *in, uint8_t out[FNV32size] ) {
    uint32_t    temp;
    uint8_t     ch;

    if ( !in || !out )
        return fnvNull; /* Null input pointer */
    temp = FNV32basis;
    while ( (ch = *in++) )
        temp = FNV32prime * ( temp ^ ch );
    for ( int i=0; i<FNV32size; ++i )
        out[i] = ((uint8_t *)&temp)[i];
    return fnvSuccess;
}   /* end FNV32string */

/* FNV32: hash a zero-terminated string not including the zero
 * with a non-standard basis
******************************************************************/
int FNV32stringBasis ( const char *in,
                       uint8_t out[FNV32size],
                       const uint8_t basis[FNV32size] ) {
    uint32_t temp;
    int i;
    uint8_t ch;

    if ( !in || !out || !basis )
        return fnvNull; /* Null input pointer */
    temp = basis[0]+(basis[1]<<8)+(basis[2]<<16)+(basis[3]<<24);
    while ( (ch = *in++) )
        temp = FNV32prime * ( temp ^ ch );
    out[0] = temp & 0xFF;
    for ( i=1; i<FNV32size; ++i ) {
        temp >>= 8;
        out[i] = temp & 0xFF;
    }
    return fnvSuccess;
}   /* end FNV32stringBasis */

/* FNV32: hash a counted block returning an integer
 ****************************************************************/
int FNV32INTblock ( const void *vin,
                    long int length,
                    uint32_t * const out ) {
    return FNV32INTblockBasis ( vin, length, out, FNV32basis );
}   /* end FNV32INTblock */

/* FNV32: hash a counted block with a non-standard basis
 ****************************************************************/
int FNV32INTblockBasis ( const void *vin,
                  long int length,
                  uint32_t * const out,
                  uint32_t basis ) {
    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 = basis; length > 0; length-- )
        temp = FNV32prime * ( temp ^ *in++ );
    *out = temp;
    return fnvSuccess;
}   /* end FNV32INTblockBasis */

/* FNV32: hash a counted block returning a 4-byte vector
 ****************************************************************/
int FNV32block ( const void *vin,
                  long int length,
                  uint8_t out[FNV32size] ) {
    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++ );
    for ( int i=0; i<FNV32size; ++i )
        out[i] = ((uint8_t *)&temp)[i];
    return fnvSuccess;
}   /* end FNV32block */

/* FNV32: hash a counted block with a non-standard basis
 ****************************************************************/
int FNV32blockBasis ( const void *vin,
                  long int length,
                  uint8_t out[FNV32size],
                  const uint8_t basis[FNV32size] ) {
```

```
    const uint8_t *in = (const uint8_t*)vin;
    uint32_t temp;

    if ( !in || !out || !basis )
        return fnvNull; /* Null input pointer */
    if ( length < 0 )
        return fnvBadParam;
    temp = basis[0]+(basis[1]<<8)+(basis[2]<<16)+(basis[3]<<24);
    for ( ; length > 0; length-- )
        temp = FNV32prime * ( temp ^ *in++ );
    for ( int i=0; i<FNV32size; ++i )
        out[i] = ((uint8_t *)&temp)[i];
    return fnvSuccess;
}   /* end FNV32blockBasis */

/* hash the contents of a file, return 32-bit integer
 ******************************************************************/
int FNV32INTfile ( const char *fname,
                   uint32_t * const out ) {
    return FNV32INTfileBasis ( fname, out, FNV32basis );
}   /* end FNV32INTfile */

/* hash the contents of a file, return 32-bit integer
 * with a non-standard basis
 ******************************************************************/
int FNV32INTfileBasis ( const char *fname,
                        uint32_t * const out,
                        uint32_t basis ) {
    FNV32context e32Context;
    int error;

    if ( !out )
        return fnvNull;
    if ( (error = FNV32INTinitBasis ( &e32Context, basis )) )
        return error;
    if ( (error = FNV32filein ( &e32Context, fname )) )
        return error;
    return FNV32INTresult ( &e32Context, out );
}   /* end FNV32INTfileBasis */

/* hash the contents of a file, return 4-byte vector
 ******************************************************************/
int FNV32file ( const char *fname,
                uint8_t out[FNV32size] ) {
    FNV32context e32Context;
    int error;

    if ( !out )
        return fnvNull;
    if ( (error = FNV32init (&e32Context)) )
        return error;
    if ( (error = FNV32filein ( &e32Context, fname)) )
        return error;
    return FNV32result ( &e32Context, out );
}   /* end FNV32file */

/* hash the contents of a file, return 4-byte vector
 * with a non-standard basis
 *****************************************************************/
int FNV32fileBasis ( const char *fname,
                     uint8_t out[FNV32size],
                     const uint8_t basis[FNV32size] ) {
    FNV32context e32Context;
    int error;

    if ( !out )
        return fnvNull;
    if ( (error = FNV32initBasis (&e32Context, basis)) )
        return error;
    if ( (error = FNV32filein ( &e32Context, fname)) )
        return error;
    return FNV32result ( &e32Context, out );
}   /* end FNV32fileBasis */

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

/* initialize context
 *****************************************************************/
int FNV32init ( FNV32context * const ctx ) {
    return FNV32INTinitBasis ( ctx, FNV32basis );
}   /* end FNV32init */

/* initialize context with a provided 32-bit integer basis
 *****************************************************************/
int FNV32INTinitBasis ( FNV32context * const ctx,
                        uint32_t basis ) {
    if ( !ctx )
        return fnvNull;
    ctx->Hash = basis;
    ctx->Computed = FNVinited+FNV32state;
    return fnvSuccess;
}   /* end FNV32INTinitBasis */

/* initialize context with a provided 4-byte vector basis
 *****************************************************************/
int FNV32initBasis ( FNV32context * const ctx,
                     const uint8_t basis[FNV32size] ) {
    if ( !ctx || !basis )
        return fnvNull;
    ctx->Hash =
        basis[0]+(basis[1]<<8)+(basis[2]<<16)+(basis[3]<<24);
    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;
            break;
        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;
            break;
        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 */

/* hash in the contents of a file
 *****************************************************************/
int FNV32filein ( FNV32context * const e32Context,
                  const char *fname ) {
    FILE *fp;
    long int i;
    char buf[1024];
    int error;

    if ( !e32Context || !fname )
        return fnvNull;
    switch ( e32Context->Computed ) {
        case FNVinited+FNV32state:
            e32Context->Computed = FNVcomputed+FNV32state;
            break;
        case FNVcomputed+FNV32state:
            break;
        default:
            return fnvStateError;
    }
    if ( ( fp = fopen ( fname, "rb") ) == NULL )
        return fnvBadParam;
    if ( (error = FNV32blockin ( e32Context, "", 0)) ) {
        fclose(fp);
        return error;
    }
    while ( ( i = fread ( buf, 1, sizeof(buf), fp ) ) > 0 )
        if ( (error = FNV32blockin ( e32Context, buf, i)) ) {
            fclose(fp);
            return error;
        }
    error = ferror(fp);
    fclose(fp);
    if (error) return fnvBadParam;
    return fnvSuccess;
}   /* end FNV32filein */

/* return hash as an integer
 ***************************************************************/
int FNV32INTresult ( FNV32context * const ctx,
                     uint32_t * const out ) {
    if ( !ctx || !out )
        return fnvNull;
    if ( ctx->Computed != FNVcomputed+FNV32state )
        return fnvStateError;
    ctx->Computed = FNVemptied+FNV32state;
    *out = ctx->Hash;
    ctx->Hash = 0;
    return fnvSuccess;
}   /* end FNV32INTresult */

/* return hash as a 4-byte vector
 ***************************************************************/
int FNV32result ( FNV32context * const ctx,
                  uint8_t out[FNV32size] ) {
    if ( !ctx || !out )
        return fnvNull;
    if ( ctx->Computed != FNVcomputed+FNV32state )
        return fnvStateError;
    ctx->Computed = FNVemptied+FNV32state;
    for ( int i=0; i<FNV32size; ++i )
        out[i] = ((uint8_t *)&ctx->Hash)[i];
    ctx->Hash = 0;
    return fnvSuccess;
}   /* end FNV32result */

<CODE ENDS>
```

8.2.2. FNV64 Code

The following code is the header and C source for 64-bit FNV-1a providing an 8-byte vector or, optionally, if 64-bit integers are supported, a 64-bit integer hash.

```
<CODE BEGINS> file "FNV64.h"

//*************************** FNV64.h ****************************//
//***************** See RFC 9923 for details. ********************//
/* Copyright (c) 2016-2025 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 "FNVErrorCodes.h"

#include <stdint.h>
#define FNV64size (64/8)
#ifdef FNV_64bitIntegers
#define FNV64basis 0xCBF29CE484222325
#endif

/* 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)
 */

/*
 *  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
 *    FNV64stringBasis: also takes an offset_basis parameter
 *
 *    FNV64block: hash a byte vector of a specified length
 *    FNV64blockBasis: also takes an offset_basis parameter
 *
 *    FNV64file: hash the contents of a file
 *    FNV64fileBasis: also takes an offset_basis parameter
 *
 *    FNV64init: initializes an FNV64 context
 *    FNV64initBasis: initializes an FNV64 context with a
 *                    provided 8-byte vector basis
 *    FNV64blockin: hash in a byte vector of a specified length
 *    FNV64stringin: hash in a zero-terminated string not
 *                   including the terminating zero
 *    FNV64filein: hash in the contents of a file
 *    FNV64result: returns the hash value
 *
 * Hash is returned as an 8-byte vector by the functions above.
 *    If 64-bit integers are supported, the following return
 *    a 64-bit integer.
 *
 *    FNV64INTstring: hash a zero-terminated string not including
 *                 the terminating zero
 *    FNV64INTstringBasis: also takes an offset_basis parameter
 *
 *    FNV64INTblock: hash a byte vector of a specified length
 *    FNV64INTblockBasis: also takes an offset_basis parameter
 *
 *    FNV64INTfile: hash the contents of a file
 *    FNV64INTfileBasis: also takes an offset_basis parameter
 *
 *    FNV64INTinitBasis: initializes an FNV64 context with a
 *                     provided 64-bit integer basis
 *    FNV64INTresult: returns the hash value
 */

#ifdef __cplusplus
extern "C" {
#endif

/* FNV64 */
extern int FNV64string ( const char *in,
                         uint8_t out[FNV64size] );
extern int FNV64stringBasis ( const char *in,
                              uint8_t out[FNV64size],
                              const uint8_t basis[FNV64size] );
extern int FNV64block ( const void *vin,
                        long int length,
                        uint8_t out[FNV64size] );
extern int FNV64blockBasis ( const void *vin,
                             long int length,
                             uint8_t out[FNV64size],
                             const uint8_t basis[FNV64size] );
extern int FNV64file ( const char * fname,
                       uint8_t out[FNV64size] );
extern int FNV64fileBasis ( const char * fname,
                            uint8_t out[FNV64size],
                            const uint8_t basis[FNV64size] );
extern int FNV64init ( FNV64context * const );
extern int FNV64initBasis ( FNV64context * const,
                            const uint8_t basis[FNV64size] );
extern int FNV64blockin ( FNV64context * const,
                          const void * vin,
                          long int length );
extern int FNV64stringin ( FNV64context * const,
                           const char * in );
extern int FNV64filein ( FNV64context * const,
                         const char *fname );
extern int FNV64result ( FNV64context * const,
                         uint8_t out[FNV64size] );

#ifdef FNV_64bitIntegers
  extern int FNV64INTstring ( const char *in,
                              uint64_t * const out );
  extern int FNV64INTstringBasis ( const char *in,
                                   uint64_t * const out,
                                   uint64_t basis );
  extern int FNV64INTblock ( const void *vin,
                             long int length,
                             uint64_t * const out );
  extern int FNV64INTblockBasis ( const void *vin,
                                  long int length,
                                  uint64_t * const out,
                                  uint64_t basis );
  extern int FNV64INTfile ( const char * fname,
                            uint64_t * const out );
  extern int FNV64INTfileBasis ( const char * fname,
                                 uint64_t * const out,
                                 uint64_t basis );
  extern int FNV64INTinitBasis ( FNV64context * const,
                                 uint64_t basis );
  extern int FNV64INTresult ( FNV64context * const,
                              uint64_t * const out );
#endif /* FNV_64bitIntegers */

#ifdef __cplusplus
}
#endif

#endif /* _FNV64_H_ */

<CODE ENDS>
```

```
<CODE BEGINS> file "FNV64.c"

//*************************** FNV64.c ***************************//
//****************** See RFC 9923 for details. ******************//
/* Copyright (c) 2016-2025 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.
 */

#include <stdio.h>

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

//*****************************************************************
// CODE THAT IS THE SAME FOR 32-BIT and 64-BIT ARITHMETIC
//*****************************************************************

/* hash the contents of a file, return byte vector
 ******************************************************************/
int FNV64file ( const char *fname,
                uint8_t out[FNV64size] ) {
    FNV64context e64Context;
    int error;

    if ( !out )
        return fnvNull;
    if ( (error = FNV64init (&e64Context)) )
        return error;
    if ( (error = FNV64filein (&e64Context, fname)) )
        return error;
    return FNV64result (&e64Context, out);
}   /* end FNV64file */

/* hash the contents of a file, return 64-bit integer
 * with a non-standard basis
 ******************************************************************/
int FNV64fileBasis ( const char *fname,
                     uint8_t out[FNV64size],
                     const uint8_t basis[FNV64size] ) {
    FNV64context e64Context;
    int error;

    if ( !out )
        return fnvNull;
    if ( (error = FNV64initBasis (&e64Context, basis)) )
        return error;
    if ( (error = FNV64filein (&e64Context, fname)) )
        return error;
    return FNV64result (&e64Context, out);
}   /* end FNV64fileBasis */

/* hash in the contents of a file
 ******************************************************************/
int FNV64filein ( FNV64context * const e64Context,
                  const char *fname ) {
    FILE *fp;
    long int i;
    char buf[1024];
    int error;
    if ( !e64Context || !fname )
        return fnvNull;
    switch ( e64Context->Computed ) {
        case FNVinited+FNV64state:
            e64Context->Computed = FNVcomputed+FNV64state;
            break;
        case FNVcomputed+FNV64state:
            break;
        default:
            return fnvStateError;
    }
    if ( ( fp = fopen ( fname, "rb") ) == NULL )
        return fnvBadParam;
    if ( (error = FNV64blockin ( e64Context, "", 0)) ) {
        fclose(fp);
        return error;
    }
    while ( ( i = fread ( buf, 1, sizeof(buf), fp ) ) > 0 )
        if ( (error = FNV64blockin ( e64Context, buf, i)) ) {
            fclose(fp);
            return error;
        }
    error = ferror(fp);
    fclose(fp);
    if (error)
        return fnvBadParam;
    return fnvSuccess;
}

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

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

/* FNV64: hash a zero-terminated string not including the zero
 * to a 64-bit integer  (64-bit)
 *****************************************************************/
int FNV64INTstring ( const char *in, uint64_t * const out ) {
    return FNV64INTstringBasis ( in, out, FNV64basis );
}   /* end FNV64INTstring */

/* FNV64: hash a zero-terminated string not including the zero
 * to a 64-bit integer  (64-bit) with a non-standard basis
 *****************************************************************/
int FNV64INTstringBasis ( const char *in,
                          uint64_t * const out,
                          uint64_t basis ) {
    uint64_t temp;
    uint8_t ch;

    if ( !in || !out )
        return fnvNull; /* Null input pointer */
    temp = basis;
    while ( (ch = *in++) )
        temp = FNV64prime * ( temp ^ ch );
    *out = temp;
    return fnvSuccess;
}   /* end FNV64INTstringBasis */

/* FNV64: hash a zero-terminated string to a 64-bit integer
 * to a byte vector  (64-bit)
 ******************************************************************/
int FNV64string ( const char *in, uint8_t out[FNV64size] ) {
    uint64_t temp;
    uint8_t ch;

    if ( !in || !out )
        return fnvNull; /* Null input pointer */
    temp = FNV64basis;
    while ( (ch = *in++) )
        temp = FNV64prime * ( temp ^ ch );
    for ( int i=0; i<FNV64size; ++i )
        out[i] = ((uint8_t *)&temp)[i];
    return fnvSuccess;
}   /* end FNV64string */

/* FNV64: hash a zero-terminated string to a 64-bit integer
 * to a byte vector  (64-bit) with a non-standard basis
 ******************************************************************/
int FNV64stringBasis ( const char *in,
                       uint8_t out[FNV64size],
                       const uint8_t basis[FNV64size] ) {
    uint64_t temp;
    int i;
    uint8_t ch;

    if ( !in || !out || !basis )
        return fnvNull; /* Null input pointer */
    temp = basis[7];
    for ( i = FNV64size-2; i>=0; --i )
        temp = (temp<<8) + basis[i];
    while ( (ch = *in++) )
        temp = FNV64prime * ( temp ^ ch );
    for ( i=0; i<FNV64size; ++i )
        out[i] = ((uint8_t *)&temp)[i];
    return fnvSuccess;
}   /* end FNV64stringBasis */

/* FNV64: hash a counted block to a 64-bit integer  (64-bit)
 ******************************************************************/
int FNV64INTblock ( const void *vin,
                    long int length,
                    uint64_t * const out ) {
    return FNV64INTblockBasis ( vin, length, out, FNV64basis );
}   /* end FNV64INTblock */

/* FNV64: hash a counted block to a 64-bit integer  (64-bit)
 * with a non-standard basis
 ******************************************************************/
int FNV64INTblockBasis ( const void *vin,
                         long int length,
                         uint64_t * const out,
                         uint64_t basis ) {
```

```
    const uint8_t *in = (const uint8_t*)vin;
    uint64_t temp;

    if ( !in || !out )
        return fnvNull;
               /* Null input pointer or null output pointer */
    if ( length < 0 )
        return fnvBadParam;
    for ( temp = basis; length > 0; length-- )
        temp = FNV64prime * ( temp ^ *in++ );
    *out = temp;
    return fnvSuccess;
}   /* end FNV64INTblockBasis */

/* FNV64: hash a counted block to a byte vector  (64-bit)
 ***************************************************************/
int FNV64block ( const void *vin,
                 long int length,
                 uint8_t out[FNV64size] ) {
    const uint8_t *in = (const uint8_t*)vin;
    uint64_t temp;

    if ( !in || !out )
        return fnvNull;
               /* Null input pointer or null output pointer */
    if ( length < 0 )
        return fnvBadParam;
    for ( temp = FNV64basis; length > 0; length-- )
        temp = FNV64prime * ( temp ^ *in++ );
    for ( int i=0; i<FNV64size; ++i )
        out[i] = ((uint8_t *)&temp)[i];
    return fnvSuccess;
}   /* end FNV64block */

/* FNV64: hash a counted block to a byte vector  (64-bit)
 * with a non-standard basis
 ***************************************************************/
int FNV64blockBasis ( const void *vin,
                      long int length,
                      uint8_t out[FNV64size],
                      const uint8_t basis[FNV64size] ) {
    const uint8_t *in = (const uint8_t*)vin;
    uint64_t temp;
    int i;

    if ( !in || !out || !basis )
        return fnvNull;
               /* Null input pointer or null output pointer */
    if ( length < 0 )
        return fnvBadParam;
    temp = basis[7];
    for ( i = FNV64size-2; i>=0; --i )
        temp = (temp<<8) + basis[i];
    for (; length > 0; length-- )
        temp = FNV64prime * ( temp ^ *in++ );
    for ( i=0; i<FNV64size; ++i )
        out[i] = ((uint8_t *)&temp)[i];
    return fnvSuccess;
```

```
}   /* end FNV64blockBasis */

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

/* initialize context  (64-bit)
 ******************************************************************/
int FNV64init( FNV64context * const ctx ) {
    return FNV64INTinitBasis ( ctx, FNV64basis );
}       /* end FNV64init */

/* initialize context with a provided 64-bit integer basis  (64-bit)
 ******************************************************************/
int FNV64INTinitBasis( FNV64context * const ctx, uint64_t basis ) {
    if ( !ctx )
        return fnvNull;
    ctx->Hash = basis;
    ctx->Computed = FNVinited+FNV64state;
    return fnvSuccess;
}   /* end FNV64INTinitBasis */

/* initialize context with a provided 8-byte vector basis  (64-bit)
 ******************************************************************/
int FNV64initBasis( FNV64context * const ctx,
                    const uint8_t basis[FNV64size] ) {
    if ( !ctx || !basis )
        return fnvNull;
    for ( int i=0; i<FNV64size; ++i )
        ((uint8_t *)&ctx->Hash)[i] = basis[i];
    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;
            break;
        case FNVcomputed+FNV64state:
            break;
        default:
            return fnvStateError;
    }
    for ( temp = ctx->Hash; length > 0; length-- )
        temp = FNV64prime * ( temp ^ *in++ );
    ctx->Hash = temp;
    return fnvSuccess;
}   /* end FNV64blockin */

/* 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;
            break;
        case FNVcomputed+FNV64state:
            break;
        default:
             return fnvStateError;
    }
    temp = ctx->Hash;
    while ( (ch = (uint8_t)*in++) )
        temp = FNV64prime * ( temp ^ ch );
    ctx->Hash = temp;
    return fnvSuccess;
}   /* end FNV64stringin */

/* return hash as 64-bit int  (64-bit)
 ******************************************************************/
int FNV64INTresult ( FNV64context * const ctx,
                     uint64_t * const out ) {
    if ( !ctx || !out )
        return fnvNull;
    if ( ctx->Computed != FNVcomputed+FNV64state )
        return fnvStateError;
    ctx->Computed = FNVemptied+FNV64state;
    *out = ctx->Hash;
    ctx->Hash = 0;
    return fnvSuccess;
}   /* end FNV64INTresult */

/* return hash as 8-byte vector  (64-bit)
 ******************************************************************/
int FNV64result ( FNV64context * const ctx,
                 uint8_t out[FNV64size] ) {
    if ( !ctx || !out )
        return fnvNull;
    if ( ctx->Computed != FNVcomputed+FNV64state )
        return fnvStateError;
    ctx->Computed = FNVemptied+FNV64state;
    for ( int i=0; i<FNV64size; ++i )
        out[i] = ((uint8_t *)&ctx->Hash)[i];
    ctx->Hash = 0;
    return fnvSuccess;
}   /* end FNV64result */

/* hash the contents of a file, return 64-bit integer
```

```
 *****************************************************************/
int FNV64INTfile ( const char *fname,
                   uint64_t * const out ) {
    FNV64context e64Context;
    int error;

    if ( !out )
        return fnvNull;
    if ( (error = FNV64init (&e64Context)) )
        return error;
    if ( (error = FNV64filein (&e64Context, fname)) )
        return error;
    return FNV64INTresult ( &e64Context, out );
}   /* end FNV64INTfile */

/* hash the contents of a file, return 64-bit integer
 * with a non-standard basis
 *****************************************************************/
int FNV64INTfileBasis ( const char *fname,
                        uint64_t * const out,
                        uint64_t basis ) {
    FNV64context e64Context;
    int error;

    if ( !out )
        return fnvNull;
    if ( (error = FNV64INTinitBasis (&e64Context, basis)) )
        return error;
    if ( (error = FNV64filein (&e64Context, fname)) )
        return error;
    return FNV64INTresult ( &e64Context, out );
}   /* end FNV64INTfileBasis */

//*****************************************************************
// 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

/* FNV64: hash a zero-terminated string not including the zero
 *****************************************************************/
int FNV64string ( const char *in, uint8_t out[FNV64size] ) {
    FNV64context ctx;
    int error;

    if ( (error = FNV64init (&ctx)) )
        return error;
    if ( (error = FNV64stringin (&ctx, in)) )
        return error;
    return FNV64result (&ctx, out);
}   /* end FNV64string */
```

```
/* FNV64: hash a zero-terminated string not including the zero
 * with a non-standard offset_basis
 *******************************************************************/
int FNV64stringBasis ( const char *in,
                  uint8_t out[FNV64size],
                  const uint8_t basis[FNV64size] ) {
    FNV64context ctx;
    int error;

    if ( (error = FNV64initBasis (&ctx, basis)) )
        return error;
    if ( (error = FNV64stringin (&ctx, in)) )
        return error;
    return FNV64result (&ctx, out);
}   /* end FNV64stringBasis */

/* FNV64: hash a counted block
 *******************************************************************/
int FNV64block ( const void *vin,
                 long int length,
                 uint8_t out[FNV64size] ) {
    FNV64context ctx;
    int error;

    if ( (error = FNV64init (&ctx)) )
        return error;
    if ( (error = FNV64blockin (&ctx, vin, length)) )
        return error;
    return FNV64result (&ctx, out);
}   /* end FNV64block */

/* FNV64: hash a counted block with a non-standard offset_basis
 *******************************************************************/
int FNV64blockBasis ( const void *vin,
                      long int length,
                      uint8_t out[FNV64size],
                      const uint8_t basis[FNV64size] ) {
    FNV64context ctx;
    int error;

    if ( (error = FNV64initBasis (&ctx, basis)) )
        return error;
    if ( (error = FNV64blockin (&ctx, vin, length)) )
        return error;
    return FNV64result (&ctx, out);
}   /* end FNV64blockBasis */

//******************************************************************
//        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] = 0xCBF2;
    ctx->Hash[1] = 0x9CE4;
    ctx->Hash[2] = 0x8422;
    ctx->Hash[3] = 0x2325;
    ctx->Computed = FNVinited+FNV64state;
    return fnvSuccess;
}   /* end FNV64init */

/* initialize context with a non-standard basis  (32-bit)
 *******************************************************************/
int FNV64initBasis ( FNV64context * const ctx,
                     const uint8_t basis[FNV64size] ) {
    if ( !ctx || !basis )
        return fnvNull;
    for ( int i=0; i < FNV64size/2; ++i ) {
        uint32_t temp = *basis++;
        ctx->Hash[i] = ( temp<<8 ) + *basis++;
    }
    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;
            break;
        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;
        for ( i=0; i<4; ++i )
            temp[i] *= FNV64primeX;
        temp[1] += temp2[1];
        temp[0] += temp2[0];
        for ( i=2; i>=0; --i ) {
            temp[i] += temp[i+1] >> 16;
            temp[i+1] &= 0xFFFF;
        }
    }
    for ( i=0; i<FNV64size/2; ++i )
        ctx->Hash[i] = temp[i];
    return fnvSuccess;
}   /* end FNV64blockin */

/* hash in a zero-terminated string not including the zero  (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;
            break;
        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;
        for ( i=0; i<4; ++i )
            temp[i] *= FNV64primeX;
        temp[1] += temp2[1];
        temp[0] += temp2[0];
        for ( i=2; i>=0; --i ) {
            temp[i] += temp[i+1] >> 16;
            temp[i+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] ) {
    if ( !ctx || !out )
        return fnvNull;
    if ( ctx->Computed != FNVcomputed+FNV64state )
        return fnvStateError;
    for ( int i=0; i<FNV64size/2; ++i ) {
        out[2*i] = ctx->Hash[i] >> 8;
        out[2*i+1] = ctx->Hash[i];
        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
//*******************************************************************

<CODE ENDS>
```

8.2.3. FNV128 Code

The following code is the header and C source for 128-bit FNV-1a providing a byte vector hash.

```
<CODE BEGINS> file "FNV128.h"

//**************************** FNV128.h *************************//
//*************** See RFC 9923 for details. ******************//
/* Copyright (c) 2016-2025 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 "FNVErrorCodes.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)
 */

/*
 *  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
 *    FNV128stringBasis: also takes an offset_basis parameter
 *
 *    FNV128block: hash a byte vector of a specified length
 *    FNV128blockBasis: also takes an offset_basis parameter
 *
 *    FNV128file: hash the contents of a file
 *    FNV128fileBasis: also takes an offset_basis parameter
 *
 *    FNV128init: initializes an FNV128 context
 *    FNV128initBasis: initializes an FNV128 context with a
 *                     provided 16-byte vector basis
 *    FNV128blockin: hash in a byte vector of a specified length
 *    FNV128stringin: hash in a zero-terminated string not
 *                    including the terminating zero
 *    FNV128filein: hash in the contents of a file
 *    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 FNV128stringBasis ( const char *in,
                               uint8_t out[FNV128size],
                               const uint8_t basis[FNV128size] );
extern int FNV128block ( const void *vin,
                         long int length,
                         uint8_t out[FNV128size] );
extern int FNV128blockBasis ( const void *vin,
                              long int length,
                              uint8_t out[FNV128size],
                              const uint8_t basis[FNV128size] );
extern int FNV128file ( const char *fname,
                        uint8_t out[FNV128size] );
extern int FNV128fileBasis ( const char *fname,
                             uint8_t out[FNV128size],
                             const uint8_t basis[FNV128size] );
extern int FNV128init ( FNV128context * const );
extern int FNV128initBasis ( FNV128context * const,
                             const uint8_t basis[FNV128size] );
extern int FNV128blockin ( FNV128context * const,
                           const void *vin,
                           long int length );
extern int FNV128stringin ( FNV128context * const,
                            const char *in );
extern int FNV128filein ( FNV128context * const,
                          const char *fname );
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 9923 for details. ******************//
/* Copyright (c) 2016-2025 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 128-bit hashes.
 */

#include <stdio.h>

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

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

/* FNV128: hash a zero-terminated string not including the zero
 ******************************************************************/
int FNV128string ( const char *in, uint8_t out[FNV128size] ) {
    FNV128context ctx;
    int error;

    if ( (error = FNV128init ( &ctx )) )
        return error;
    if ( (error = FNV128stringin ( &ctx, in )) )
```

```
        return error;
    return FNV128result (&ctx, out);
}   /* end FNV128string */

/* FNV128: hash a zero-terminated string not including the zero
 ******************************************************************/
int FNV128stringBasis ( const char *in,
                        uint8_t out[FNV128size],
                        const uint8_t basis[FNV128size] ) {
    FNV128context ctx;
    int error;

    if ( (error = FNV128initBasis ( &ctx, basis )) )
        return error;
    if ( (error = FNV128stringin ( &ctx, in )) )
        return error;
    return FNV128result ( &ctx, out );
}   /* end FNV128stringBasis */

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

    if ( (error = FNV128init ( &ctx )) )
        return error;
    if ( (error = FNV128blockin ( &ctx, vin, length )) )
        return error;
    return FNV128result ( &ctx, out );
}   /* end FNV128block */

/* FNV128: hash a counted block  (64/32-bit)
 ******************************************************************/
int FNV128blockBasis ( const void *vin,
                       long int length,
                       uint8_t out[FNV128size],
                       const uint8_t basis[FNV128size] ) {
    FNV128context ctx;
    int error;

    if ( (error = FNV128initBasis ( &ctx, basis )) )
        return error;
    if ( (error = FNV128blockin ( &ctx, vin, length )) )
        return error;
    return FNV128result ( &ctx, out );
}   /* end FNV128blockBasis */

/* hash the contents of a file
 ******************************************************************/
int FNV128file ( const char *fname,
                 uint8_t out[FNV128size] ) {
    FNV128context e128Context;
    int error;

    if ( !out )
```

```
        return fnvNull;
    if ( (error = FNV128init (&e128Context)) )
        return error;
    if ( (error = FNV128filein (&e128Context, fname)) )
        return error;
    return FNV128result ( &e128Context, out );
}   /* end FNV128file */

/* hash the contents of a file with a non-standard basis
 ******************************************************************/
int FNV128fileBasis ( const char *fname,
                      uint8_t out[FNV128size],
                      const uint8_t basis[FNV128size] ) {
    FNV128context e128Context;
    int error;

    if ( !out )
        return fnvNull;
    if ( (error = FNV128initBasis (&e128Context, basis)) )
        return error;
    if ( (error = FNV128filein (&e128Context, fname)) )
        return error;
    return FNV128result ( &e128Context, out );
}   /* end FNV128fileBasis */

/* hash in the contents of a file
 ******************************************************************/
int FNV128filein ( FNV128context * const e128Context,
                   const char *fname ) {
    FILE *fp;
    long int i;
    char buf[1024];
    int error;

    if ( !e128Context || !fname )
        return fnvNull;
    switch ( e128Context->Computed ) {
        case FNVinited+FNV128state:
            e128Context->Computed = FNVcomputed+FNV128state;
            break;
        case FNVcomputed+FNV128state:
            break;
        default:
            return fnvStateError;
    }
    if ( ( fp = fopen ( fname, "rb") ) == NULL )
        return fnvBadParam;
    if ( (error = FNV128blockin ( e128Context, "", 0)) ) {
        fclose(fp);
        return error;
    }
    while ( ( i = fread ( buf, 1, sizeof(buf), fp ) ) > 0 )
        if ( (error = FNV128blockin ( e128Context, buf, i)) ) {
            fclose(fp);
            return error;
        }
    error = ferror(fp);
    fclose(fp);
    if (error) return fnvBadParam;
    return fnvSuccess;
}   /* end FNV128filein */

//*****************************************************************
// 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

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

/* initialize context  (64-bit)
 ******************************************************************/
int FNV128init ( FNV128context * const ctx ) {
    const uint32_t FNV128basis[FNV128size/4] =
        { 0x6C62272E, 0x07BB0142, 0x62B82175, 0x6295C58D };

    if ( !ctx )
        return fnvNull;
    for ( int i=0; i<4; ++i )
        ctx->Hash[i] = FNV128basis[i];
    ctx->Computed = FNVinited+FNV128state;
    return fnvSuccess;
}   /* end FNV128init */

/* initialize context with a provided 16-byte vector basis  (64-bit)
 ******************************************************************/
int FNV128initBasis ( FNV128context * const ctx,
                      const uint8_t basis[FNV128size] ) {
   if ( !ctx || !basis )
        return fnvNull;
    for ( int i=0; i < FNV128size/4; ++i ) {
        uint32_t temp = *basis++<<24;
        temp += *basis++<<16;
        temp += *basis++<<8;
        ctx->Hash[i] = temp + *basis++;
    }
    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;
            break;
        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];
        for ( i = 3; i > 0; --i ) {
            temp[i-1] += temp[i] >> 32;
            temp[i] &= 0xFFFFFFFF;
        }
    }
    for ( i=0; i<FNV128size/4; ++i )
        ctx->Hash[i] = (uint32_t)temp[i];
    return fnvSuccess;
}   /* end FNV128blockin */

/* hash in a zero-terminated string not including the zero  (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;
            break;
        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];
        for ( i = 3; i > 0; --i ) {
            temp[i-1] += temp[i] >> 32;
            temp[i] &= 0xFFFFFFFF;
        }
    }
    for ( i=0; i<FNV128size/4; ++i )
        ctx->Hash[i] = (uint32_t)temp[i];
    return fnvSuccess;
}   /* end FNV128stringin */

/* return hash as 16-byte vector  (64-bit)
 *****************************************************************/
int FNV128result ( FNV128context * const ctx,
                   uint8_t out[FNV128size] ) {
    if ( !ctx || !out )
        return fnvNull;
    if ( ctx->Computed != FNVcomputed+FNV128state )
        return fnvStateError;
    for ( int i=0; i<FNV128size/4; ++i ) {
        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];
        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

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

/* initialize context  (32-bit)
 *****************************************************************/
int FNV128init ( FNV128context * const ctx ) {
    const uint16_t FNV128basis[FNV128size/2] =
              { 0x6C62, 0x272E, 0x07BB, 0x0142,
                0x62B8, 0x2175, 0x6295, 0xC58D };

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

/* initialize context with a provided 16-byte vector basis  (32-bit)
 *******************************************************************/
int FNV128initBasis ( FNV128context * const ctx,
                      const uint8_t basis[FNV128size] ) {
    if ( !ctx || !basis )
        return fnvNull;
    for ( int i=0; i < FNV128size/2; ++i ) {
        uint32_t temp = *basis++;
        ctx->Hash[i] = ( temp<<8 ) + *basis++;
    }
    ctx->Computed = FNVinited+FNV128state;
    return fnvSuccess;
}   /* end FNV128initBasis */

/* hash in a counted block  (32-bit)
 ******************************************************************/
int FNV128blockin ( FNV128context * const 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;
            break;
        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++;
        for ( i=2; i >= 0; --i )
            temp2[i] = temp[i+5] << FNV128shift;
        for ( i=0; i < (FNV128size/2); ++i )
            temp[i] *= FNV128primeX;
        for ( i=2; i >= 0; --i )
```

```
            temp[i] += temp2[i];
        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 zero-terminated string not including the zero  (32-bit)
 ******************************************************************/
int FNV128stringin ( FNV128context * const 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;
            break;
        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;
        for ( i=2; i >= 0; --i )
            temp2[i] = temp[i+5] << FNV128shift;
        for ( i=0; i<(FNV128size/2); ++i )
            temp[i] *= FNV128primeX;
        for ( i=2; i >= 0; --i )
            temp[i] += temp2[i];
        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 * const ctx,
                   uint8_t out[FNV128size] ) {
    if ( !ctx || !out )
        return fnvNull;
    if ( ctx->Computed != FNVcomputed+FNV128state )
```

```
        return fnvStateError;
    for ( int i=0; i<FNV128size/2; ++i ) {
        out[2*i] = ctx->Hash[i] >> 8;
        out[2*i+1] = ctx->Hash[i];
        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
//*****************************************************************

<CODE ENDS>
```

8.2.4. FNV256 Code

The following code is the header and C source for 256-bit FNV-1a providing a byte vector hash.

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

//************************** FNV256.h ***********************//
//************** See RFC 9923 for details. *****************//
/* Copyright (c) 2016-2025 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 "FNVErrorCodes.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)
 */

/*
```

```
 *  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 terminating zero
 *    FNV256stringBasis: also takes an offset_basis parameter
 *
 *    FNV256block: hash a byte vector of a specified length
 *    FNV256blockBasis: also takes an offset_basis parameter
 *
 *    FNV256file: hash the contents of a file
 *    FNV256fileBasis: also takes an offset_basis parameter
 *
 *    FNV256init: initializes an FNV256 context
 *    FNV256initBasis: initializes an FNV256 context with a
 *                     provided 32-byte vector basis
 *    FNV256blockin: hash in a byte vector of a specified length
 *    FNV256stringin: hash in a zero-terminated string not
 *                    including the terminating zero
 *    FNV256filein: hash in the contents of a file
 *    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 FNV256stringBasis ( const char *in,
                               uint8_t out[FNV256size],
                               const uint8_t basis[FNV256size] );
extern int FNV256block ( const void *vin,
                         long int length,
                         uint8_t out[FNV256size] );
extern int FNV256blockBasis ( const void *vin,
                              long int length,
                              uint8_t out[FNV256size],
                              const uint8_t basis[FNV256size] );
extern int FNV256file ( const char *fname,
                        uint8_t out[FNV256size] );
extern int FNV256fileBasis ( const char *fname,
                             uint8_t out[FNV256size],
                             const uint8_t basis[FNV256size] );
extern int FNV256init ( FNV256context * const );
extern int FNV256initBasis ( FNV256context * const,
                             const uint8_t basis[FNV256size] );
extern int FNV256blockin ( FNV256context * const,
                           const void *vin,
                           long int length );
extern int FNV256stringin ( FNV256context * const,
                            const char *in );
extern int FNV256filein ( FNV256context * const,
                          const char *fname );
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 9923 for details. ******************//
/* Copyright (c) 2016-2025 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 256-bit hashes.
 */

#include <stdio.h>

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

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

/* FNV256: hash a zero-terminated string not including the zero
 *****************************************************************/
int FNV256string ( const char *in, uint8_t out[FNV256size] ) {
    FNV256context ctx;
    int error;

    if ( (error = FNV256init ( &ctx )) )
```

```
        return error;
    if ( (error = FNV256stringin ( &ctx, in )) )
        return error;
    return FNV256result ( &ctx, out );
}   /* end FNV256string */

/* FNV256: hash a zero-terminated string not including the zero
 * with a non-standard basis
 ******************************************************************/
int FNV256stringBasis ( const char *in,
                        uint8_t out[FNV256size],
                        const uint8_t basis[FNV256size] ) {
    FNV256context ctx;
    int error;

    if ( (error = FNV256initBasis ( &ctx, basis )) )
        return error;
    if ( (error = FNV256stringin ( &ctx, in )) )
        return error;
    return FNV256result ( &ctx, out );
}   /* end FNV256stringBasis */

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

    if ( (error = FNV256init ( &ctx )) )
        return error;
    if ( (error = FNV256blockin ( &ctx, vin, length)) )
        return error;
    return FNV256result ( &ctx, out );
}   /* end FNV256block */

/* FNV256: hash a counted block  (64/32-bit)
 * with a non-standard basis
 ******************************************************************/
int FNV256blockBasis ( const void *vin,
                       long int length,
                       uint8_t out[FNV256size],
                       const uint8_t basis[FNV256size] ) {
    FNV256context ctx;
    int error;

    if ( (error = FNV256initBasis ( &ctx, basis )) )
        return error;
    if ( (error = FNV256blockin ( &ctx, vin, length)) )
        return error;
    return FNV256result ( &ctx, out );
}   /* end FNV256blockBasis */

/* hash the contents of a file
 ******************************************************************/
int FNV256file ( const char *fname,
                        uint8_t out[FNV256size] ) {
```

```
    FNV256context e256Context;
    int error;

    if ( !out )
        return fnvNull;
    if ( (error = FNV256init (&e256Context)) )
        return error;
    if ( (error = FNV256filein (&e256Context, fname)) )
        return error;
    return FNV256result ( &e256Context, out );
}   /* end FNV256file */

/* hash the contents of a file with a non-standard basis
 ****************************************************************/
int FNV256fileBasis ( const char *fname,
                      uint8_t out[FNV256size],
                      const uint8_t basis[FNV256size]) {
    FNV256context e256Context;
    int error;

    if ( !out )
        return fnvNull;
    if ( (error = FNV256initBasis (&e256Context, basis)) )
        return error;
    if ( (error = FNV256filein (&e256Context, fname)) )
        return error;
    return FNV256result ( &e256Context, out );
}   /* end FNV256fileBasis */

/* hash in the contents of a file
 ****************************************************************/
int FNV256filein ( FNV256context * const e256Context,
                   const char *fname ) {
    FILE *fp;
    long int i;
    char buf[1024];
    int error;

    if ( !e256Context || !fname )
        return fnvNull;
    switch ( e256Context->Computed ) {
        case FNVinited+FNV256state:
            e256Context->Computed = FNVcomputed+FNV256state;
            break;
        case FNVcomputed+FNV256state:
            break;
        default:
             return fnvStateError;
    }
    if ( ( fp = fopen ( fname, "rb") ) == NULL )
        return fnvBadParam;
    if ( (error = FNV256blockin ( e256Context, "", 0)) ) {
        fclose(fp);
        return error;
    }
    while ( ( i = fread ( buf, 1, sizeof(buf), fp ) ) > 0 )
        if ( (error = FNV256blockin ( e256Context, buf, i)) ) {
            fclose(fp);
            return error;
        }
    error = ferror(fp);
    fclose(fp);
    if (error) return fnvBadParam;
    return fnvSuccess;
}   /* end FNV256filein */

//*****************************************************************
// 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

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

/* initialize context  (64-bit)
 ******************************************************************/
int FNV256init ( FNV256context * const ctx ) {
    const uint32_t FNV256basis[FNV256size/4] = {
             0xDD268DBC, 0xAAC55036, 0x2D98C384, 0xC4E576CC,
             0xC8B15368, 0x47B6BBB3, 0x1023B4C8, 0xCAEE0535 };

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

/* initialize context with a provided 32-byte vector basis  (64-bit)
 * with a non-standard basis
 ******************************************************************/
int FNV256initBasis ( FNV256context * const ctx,
                      const uint8_t basis[FNV256size] ) {
    if ( !ctx || !basis )
        return fnvNull;
    for ( int i=0; i < FNV256size/4; ++i ) {
        uint32_t temp = *basis++<<24;
        temp += *basis++<<16;
        temp += *basis++<<8;
        ctx->Hash[i] = temp + *basis++;
    }
    ctx->Computed = FNVinited+FNV256state;
    return fnvSuccess;
}   /* end FNV256initBasis */

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

```
int FNV256blockin ( FNV256context * const 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;
            break;
        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++;
        for ( i=2; i >= 0; --i )
            temp2[i] = temp[i+5] << FNV256shift;
        for ( i=0; i < FNV256size/4; ++i )
            temp[i] *= FNV256primeX;
        for ( i=2; i >= 0; --i )
            temp[i] += temp2[i];
        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 FNV256blockin */

/* hash in a zero-terminated string not including the zero  (64-bit)
 ******************************************************************/
int FNV256stringin ( FNV256context * const 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;
            break;
        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;
        for ( i=2; i >= 0; --i )
            temp2[i] = temp[i+5] << FNV256shift;
        for ( i=0; i<FNV256size/4; ++i )
            temp[i] *= FNV256primeX;
        for ( i=2; i >= 0; --i )
            temp[i] += temp2[i];
        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 as 8-byte vector  (64-bit)
 ******************************************************************/
int FNV256result ( FNV256context * const ctx,
                   uint8_t out[FNV256size] ) {
    if ( !ctx || !out )
        return fnvNull;
    if ( ctx->Computed != FNVcomputed+FNV256state )
        return fnvStateError;
    for ( int i=0; i<FNV256size/4; ++i ) {
        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];
        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
//******************************************************************

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

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

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

/* initialize context  (32-bit)
 ******************************************************************/
int FNV256init ( FNV256context * const ctx ) {
    const uint16_t FNV256basis[FNV256size/2] = {
0xDD26, 0x8DBC, 0xAAC5, 0x5036, 0x2D98, 0xC384, 0xC4E5, 0x76CC,
0xC8B1, 0x5368, 0x47B6, 0xBBB3, 0x1023, 0xB4C8, 0xCAEE, 0x0535 };

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

/* initialize context with a provided 32-byte vector basis  (32-bit)
 ******************************************************************/
int FNV256initBasis ( FNV256context * const ctx,
                      const uint8_t basis[FNV256size] ) {
    if ( !ctx || !basis )
        return fnvNull;
    for ( int i=0; i < FNV256size/2; ++i ) {
        uint32_t temp = *basis++;
        ctx->Hash[i] = ( temp<<8 ) + *basis++;
    }
    ctx->Computed = FNVinited+FNV256state;
    return fnvSuccess;
}   /* end FNV256initBasis */

/* hash in a counted block  (32-bit)
 ******************************************************************/
int FNV256blockin ( FNV256context * const 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;
            break;
        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 zero-terminated string not including the zero  (32-bit)
 *******************************************************************/
int FNV256stringin ( FNV256context * const 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;
            break;
        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 * const ctx,
                   uint8_t out[FNV256size] ) {
    if ( !ctx || !out )
        return fnvNull;
    if ( ctx->Computed != FNVcomputed+FNV256state )
        return fnvStateError;
    for ( int i=0; i<FNV256size/2; ++i ) {
        out[2*i] = ctx->Hash[i] >> 8;
        out[2*i+1] = ctx->Hash[i];
        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
//****************************************************************

<CODE ENDS>
```

8.2.5. FNV512 Code

The following code is the header and C source for 512-bit FNV-1a providing a byte vector hash.

```
<CODE BEGINS> file "FNV512.h"

//************************* FNV512.h **********************//
//************** See RFC 9923 for details. ****************//
/* Copyright (c) 2016-2025 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 "FNVErrorCodes.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)
 */

/*
 *  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
 *    FNV512stringBasis: also takes an offset_basis parameter
 *
 *    FNV512block: hash a byte vector of a specified length
 *    FNV512blockBasis: also takes an offset_basis parameter
 *
 *    FNV512file: hash the contents of a file
 *    FNV512fileBasis: also takes an offset_basis parameter
 *
 *    FNV512init: initializes an FNV512 context
 *    FNV512initBasis: initializes an FNV512 context with a
 *                     provided 64-byte vector basis
 *    FNV512blockin: hash in a byte vector of a specified length
 *    FNV512stringin: hash in a zero-terminated string not
 *                    including the terminating zero
 *    FNV512filein: hash in the contents of a file
 *    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 FNV512stringBasis ( const char *in,
                               uint8_t out[FNV512size],
                               const uint8_t basis[FNV512size] );
extern int FNV512block ( const void *vin,
                         long int length,
                         uint8_t out[FNV512size] );
extern int FNV512blockBasis ( const void *vin,
                              long int length,
                              uint8_t out[FNV512size],
                              const uint8_t basis[FNV512size] );
extern int FNV512file ( const char *fname,
                        uint8_t out[FNV512size] );
extern int FNV512fileBasis ( const char *fname,
                             uint8_t out[FNV512size],
                             const uint8_t basis[FNV512size] );
extern int FNV512init ( FNV512context * const );
extern int FNV512initBasis ( FNV512context * const,
                             const uint8_t basis[FNV512size] );
extern int FNV512blockin ( FNV512context * const,
                           const void *vin,
                           long int length );
extern int FNV512stringin ( FNV512context * const,
                            const char *in );
extern int FNV512filein ( FNV512context * const,
                          const char *fname );
extern int FNV512result ( FNV512context * const,
                          uint8_t out[FNV512size] );

#ifdef __cplusplus
}
#endif

#endif /* _FNV512_H_ */

<CODE ENDS>
```

```
<CODE BEGINS> file "FNV512.c"

//****************************** FNV512.c **************************//
//******************** See RFC 9923 for details. ******************//
/* Copyright (c) 2016-2025 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 512-bit hashes.
 */

#include <stdio.h>

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

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

```
/* FNV512: hash a zero-terminated string not including the zero
 ****************************************************************/
int FNV512string ( const char *in, uint8_t out[FNV512size] ) {
    FNV512context ctx;
    int error;

    if ( (error = FNV512init ( &ctx )) )
        return error;
    if ( (error = FNV512stringin ( &ctx, in )) )
        return error;
    return FNV512result ( &ctx, out );
}   /* end FNV512string */

/* FNV512: hash a zero-terminated string not including the zero
 * with a non-standard basis
 ****************************************************************/
int FNV512stringBasis ( const char *in,
                        uint8_t out[FNV512size],
                        const uint8_t basis[FNV512size] ) {
    FNV512context ctx;
    int error;

    if ( (error = FNV512initBasis ( &ctx, basis )) )
        return error;
    if ( (error = FNV512stringin ( &ctx, in )) )
        return error;
    return FNV512result ( &ctx, out );
}   /* end FNV512stringBasis */

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

    if ( (error = FNV512init ( &ctx )) )
        return error;
    if ( (error = FNV512blockin ( &ctx, vin, length)) )
        return error;
    return FNV512result ( &ctx, out );
}   /* end FNV512block */

/* FNV512: hash a counted block  (64/32-bit)
 * with a non-standard basis
 ****************************************************************/
int FNV512blockBasis ( const void *vin,
                       long int length,
                       uint8_t out[FNV512size],
                       const uint8_t basis[FNV512size] ) {
    FNV512context ctx;
    int error;

    if ( (error = FNV512initBasis ( &ctx, basis )) )
        return error;
    if ( (error = FNV512blockin ( &ctx, vin, length)) )
```

```
        return error;
    return FNV512result ( &ctx, out );
}   /* end FNV512blockBasis */

/* hash the contents of a file
 ******************************************************************/
int FNV512file ( const char *fname,
                 uint8_t out[FNV512size] ) {
    FNV512context e512Context;
    int error;

    if ( !out )
        return fnvNull;
    if ( (error = FNV512init (&e512Context)) )
        return error;
    if ( (error = FNV512filein (&e512Context, fname)) )
        return error;
    return FNV512result ( &e512Context, out );
}   /* end FNV512file */

/* hash the contents of a file with a non-standard basis
 ******************************************************************/
int FNV512fileBasis ( const char *fname,
                      uint8_t out[FNV512size],
                      const uint8_t basis[FNV512size] ) {
    FNV512context e512Context;
    int error;

    if ( !out )
        return fnvNull;
    if ( (error = FNV512initBasis (&e512Context, basis)) )
        return error;
    if ( (error = FNV512filein (&e512Context, fname)) )
        return error;
    return FNV512result ( &e512Context, out );
}   /* end FNV512fileBasis */

/* hash in the contents of a file
 ******************************************************************/
int FNV512filein ( FNV512context * const e512Context,
                   const char *fname ) {
    FILE *fp;
    long int i;
    char buf[1024];
    int error;

    if ( !e512Context || !fname )
        return fnvNull;
    switch ( e512Context->Computed ) {
        case FNVinited+FNV512state:
            e512Context->Computed = FNVcomputed+FNV512state;
            break;
        case FNVcomputed+FNV512state:
            break;
        default:
             return fnvStateError;
    }
    if ( ( fp = fopen ( fname, "rb") ) == NULL )
```

```
        return fnvBadParam;
    if ( (error = FNV512blockin ( e512Context, "", 0)) ) {
        fclose(fp);
        return error;
    }
    while ( ( i = fread ( buf, 1, sizeof(buf), fp ) ) > 0 )
        if ( (error=FNV512blockin ( e512Context, buf, i)) ) {
            fclose(fp);
            return error;
        }
    error = ferror(fp);
    fclose(fp);
    if (error) return fnvBadParam;
    return fnvSuccess;
}   /* end FNV512filein */

//*****************************************************************
// 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

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

/* initialize context  (64-bit)
 ******************************************************************/
int FNV512init ( FNV512context * const ctx ) {
    const uint32_t FNV512basis[FNV512size/4] = {
         0xB86DB0B1, 0x171F4416, 0xDCA1E50F, 0x309990AC,
         0xAC87D059, 0xC9000000, 0x00000000, 0x00000D21,
         0xE948F68A, 0x34C192F6, 0x2EA79BC9, 0x42DBE7CE,
         0x18203641, 0x5F56E34B, 0xAC982AAC, 0x4AFE9FD9 };

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

/* initialize context with a provided 64-byte vector basis  (64-bit)
 ******************************************************************/
int FNV512initBasis ( FNV512context * const ctx,
                      const uint8_t basis[FNV512size] ) {
     if ( !ctx || !basis )
        return fnvNull;
    for ( int i=0; i < FNV512size/4; ++i ) {
        uint32_t temp = *basis++<<24;
        temp += *basis++<<16;
        temp += *basis++<<8;
        ctx->Hash[i] = temp + *basis++;
    }
    ctx->Computed = FNVinited+FNV512state;
    return fnvSuccess;
}   /* end FNV512initBasis */

/* hash in a counted block  (64-bit)
 *****************************************************************/
int FNV512blockin ( FNV512context * const 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;
    if ( length < 0 )
        return fnvBadParam;
    switch ( ctx->Computed ) {
        case FNVinited+FNV512state:
            ctx->Computed = FNVcomputed+FNV512state;
            break;
        case FNVcomputed+FNV512state:
            break;
        default:
            return fnvStateError;
    }
    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 zero-terminated string not including the zero  (64-bit)
 *****************************************************************/
int FNV512stringin ( FNV512context * const 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;
            break;
        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 * const ctx,
                   uint8_t out[FNV512size] ) {
    if ( !ctx || !out )
        return fnvNull;
    if ( ctx->Computed != FNVcomputed+FNV512state )
        return fnvStateError;
    for ( int i=0; i<FNV512size/4; ++i ) {
        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];
        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

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

/* initialize context  (32-bit)
 ******************************************************************/
int FNV512init ( FNV512context * const ctx ) {
    const 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 };

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

/* initialize context with a provided 64-byte vector basis  (32-bit)
 ******************************************************************/
int FNV512initBasis ( FNV512context * const ctx,
                      const uint8_t basis[FNV512size] ) {
    if ( !ctx || !basis )
        return fnvNull;
    for ( int i=0; i < FNV512size/2; ++i ) {
        uint32_t temp = *basis++;
        ctx->Hash[i] = ( temp<<8 ) + *basis++;
    }
    ctx->Computed = FNVinited+FNV512state;
    return fnvSuccess;
}   /* end FNV512initBasis */

/* hash in a counted block  (32-bit)
 ******************************************************************/
int FNV512blockin ( FNV512context * const 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;
    if ( length < 0 )
        return fnvBadParam;
   switch ( ctx->Computed ) {
        case FNVinited+FNV512state:
            ctx->Computed = FNVcomputed+FNV512state;
            break;
        case FNVcomputed+FNV512state:
            break;
        default:
            return fnvStateError;
    }
    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 zero-terminated string not including the zero  (32-bit)
 *****************************************************************/
int FNV512stringin ( FNV512context * const 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;
            break;
        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 * const ctx,
                   uint8_t out[FNV512size] ) {
    if ( !ctx || !out )
        return fnvNull;
    if ( ctx->Computed != FNVcomputed+FNV512state )
        return fnvStateError;
    for ( int i=0; i<FNV512size/2; ++i ) {
        out[2*i] = ctx->Hash[i] >> 8;
        out[2*i+1] = ctx->Hash[i];
        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
//******************************************************************

<CODE ENDS>
```

8.2.6. FNV1024 Code

The following code is the header and C source for 1024-bit FNV-1a providing a byte vector hash.

```
<CODE BEGINS> file "FNV1024.h"

//************************ FNV1024.h ***********************//
//************* See RFC 9923 for details. *****************//
/* Copyright (c) 2016-2025 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 "FNVErrorCodes.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)
 */

/*
 *  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
 *    FNV1024stringBasis: also takes an offset_basis parameter
 *
 *    FNV1024block: hash a byte vector of a specified length
 *    FNV1024blockBasis: also takes an offset_basis parameter
 *
 *    FNV1024file: hash the contents of a file
 *    FNV1024fileBasis: also takes an offset_basis parameter
 *
 *    FNV1024init: initializes an FNV1024 context
 *    FNV1024initBasis: initializes an FNV1024 context with a
 *                      provided 128-byte vector basis
 *    FNV1024blockin: hash in a byte vector of a specified length
 *    FNV1024stringin: hash in a zero-terminated string not
 *                      including the terminating zero
 *    FNV1024filein: hash in the contents of a file
 *    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,
                           uint8_t out[FNV1024size] );
extern int FNV1024stringBasis ( const char *in,
                                uint8_t out[FNV1024size],
                                const uint8_t basis[FNV1024size] );
extern int FNV1024block ( const void *vin,
                          long int length,
                          uint8_t out[FNV1024size] );
extern int FNV1024blockBasis ( const void *vin,
                               long int length,
                               uint8_t out[FNV1024size],
                               const uint8_t basis[FNV1024size] );
extern int FNV1024file ( const char *fname,
                         uint8_t out[FNV1024size] );
extern int FNV1024fileBasis ( const char *fname,
                              uint8_t out[FNV1024size],
                              const uint8_t basis[FNV1024size] );
extern int FNV1024init ( FNV1024context * const );
extern int FNV1024initBasis ( FNV1024context * const,
                              const uint8_t basis[FNV1024size] );
extern int FNV1024blockin ( FNV1024context * const,
                            const void *vin,
                            long int length );
extern int FNV1024stringin ( FNV1024context * const,
                            const char *in );
extern int FNV1024filein ( FNV1024context * const,
                           const char *fname );
extern int FNV1024result ( FNV1024context * const,
                           uint8_t out[FNV1024size] );

#ifdef __cplusplus
}
#endif

#endif /* _FNV1024_H_ */

<CODE ENDS>
```

```
<CODE BEGINS> file "FNV1024.c"

//************************** FNV1024.c **************************//
//****************** See RFC 9923 for details. ******************//
/* Copyright (c) 2016-2025 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 1024-bit hashes.
 */

#include <stdio.h>

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

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

/* FNV1024: hash a zero-terminated string not including the zero
 ******************************************************************/
int FNV1024string ( const char *in, uint8_t out[FNV1024size] ) {
    FNV1024context ctx;
    int error;

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

/* FNV1024: hash a zero-terminated string not including the zero
 * with a non-standard basis
 ******************************************************************/
int FNV1024stringBasis ( const char *in,
                         uint8_t out[FNV1024size],
                         const uint8_t basis[FNV1024size] ) {
    FNV1024context ctx;
    int error;

    if ( (error = FNV1024initBasis ( &ctx, basis )) )
        return error;
    if ( (error = FNV1024stringin ( &ctx, in )) )
        return error;
    return FNV1024result ( &ctx, out );
}   /* end FNV1024stringBasis */

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

    if ( (error = FNV1024init ( &ctx )) )
        return error;
    if ( (error = FNV1024blockin ( &ctx, vin, length)) )
        return error;
    return FNV1024result ( &ctx, out );
```

```
}   /* end FNV1024block */

/* FNV1024: hash a counted block  (64/32-bit)
 * with a non-standard basis
 *******************************************************************/
int FNV1024blockBasis ( const void *vin,
                        long int length,
                        uint8_t out[FNV1024size],
                        const uint8_t basis[FNV1024size] ) {
    FNV1024context ctx;
    int error;

    if ( (error = FNV1024initBasis ( &ctx, basis )) )
        return error;
    if ( (error = FNV1024blockin ( &ctx, vin, length)) )
        return error;
    return FNV1024result ( &ctx, out );
}   /* end FNV1024blockBasis */

/* hash the contents of a file
 *******************************************************************/
int FNV1024file ( const char *fname,
                  uint8_t out[FNV1024size] ) {
    FNV1024context e1024Context;
    int error;

    if ( !out )
        return fnvNull;
    if ( (error = FNV1024init (&e1024Context)) )
        return error;
    if ( (error = FNV1024filein (&e1024Context, fname)) )
        return error;
    return FNV1024result ( &e1024Context, out );
}   /* end FNV1024file */

/* hash the contents of a file with a non-standard basis
 *******************************************************************/
int FNV1024fileBasis ( const char *fname,
                       uint8_t out[FNV1024size],
                       const uint8_t basis[FNV1024size] ) {
    FNV1024context e1024Context;
    int error;

    if ( !out )
        return fnvNull;
    if ( (error = FNV1024initBasis (&e1024Context, basis)) )
        return error;
    if ( (error = FNV1024filein (&e1024Context, fname)) )
        return error;
    return FNV1024result ( &e1024Context, out );
}   /* end FNV1024fileBasis */

/* hash in the contents of a file
 *******************************************************************/
int FNV1024filein ( FNV1024context * const e1024Context,
                    const char *fname ) {
    FILE *fp;
    long int i;
    char buf[1024];
    int error;

    if ( !e1024Context || !fname )
        return fnvNull;
    switch ( e1024Context->Computed ) {
        case FNVinited+FNV1024state:
            e1024Context->Computed = FNVcomputed+FNV1024state;
            break;
        case FNVcomputed+FNV1024state:
            break;
        default:
             return fnvStateError;
    }
    if ( ( fp = fopen ( fname, "rb") ) == NULL )
        return fnvBadParam;
    if ( (error = FNV1024blockin ( e1024Context, "", 0)) ) {
        fclose(fp);
        return error;
    }
    while ( ( i = fread ( buf, 1, sizeof(buf), fp ) ) > 0 )
        if ( (error = FNV1024blockin ( e1024Context, buf, i)) ) {
            fclose(fp);
            return error;
        }
    error = ferror(fp);
    fclose(fp);
    if (error) return fnvBadParam;
    return fnvSuccess;
}   /* end FNV1024filein */

//*****************************************************************//
// 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

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

/* initialize context  (64-bit)
 *****************************************************************/
int FNV1024init ( FNV1024context * const ctx ) {
    const 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 };

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

/* initialize context with a provided 128-byte vector basis  (64-bit)
 *******************************************************************/
int FNV1024initBasis ( FNV1024context * const ctx,
                       const uint8_t basis[FNV1024size] ) {
    if ( !ctx || !basis )
        return fnvNull;
    for ( int i=0; i < FNV1024size/4; ++i ) {
        uint32_t temp = *basis++<<24;
        temp += *basis++<<16;
        temp += *basis++<<8;
        ctx->Hash[i] = temp + *basis++;
    }
    ctx->Computed = FNVinited+FNV1024state;
    return fnvSuccess;
}   /* end FNV1024initBasis */

/* hash in a counted block  (64-bit)
 ******************************************************************/
int FNV1024blockin ( FNV1024context * const 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;
    if ( length < 0 )
        return fnvBadParam;
    switch ( ctx->Computed ) {
        case FNVinited+FNV1024state:
            ctx->Computed = FNVcomputed+FNV1024state;
            break;
        case FNVcomputed+FNV1024state:
            break;
        default:
            return fnvStateError;
    }
    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;
        }
    }   /* end for length */
    for ( i=0; i<FNV1024size/4; ++i )
        ctx->Hash[i] = (uint32_t)temp[i];  // store back into hash
    return fnvSuccess;
}   /* end FNV1024blockin */

/* hash in a zero-terminated string not including the zero  (64-bit)
 ******************************************************************/
int FNV1024stringin ( FNV1024context * const 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;
            break;
        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 * const ctx,
                    uint8_t out[FNV1024size] ) {
    if ( !ctx || !out )
        return fnvNull;
    if ( ctx->Computed != FNVcomputed+FNV1024state )
        return fnvStateError;
    for ( int i=0; i<FNV1024size/4; ++i ) {
        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];
        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
//******************************************************************//

/*
 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

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

/* initialize context  (32-bit)
 ********************************************************************/
int FNV1024init ( FNV1024context * const ctx ) {
    const 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 };
```

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

/* initialize context with a provided 128-byte vector basis  (32-bit)
 ******************************************************************/
int FNV1024initBasis ( FNV1024context * const ctx,
                       const uint8_t basis[FNV1024size] ) {
    if ( !ctx || !basis )
        return fnvNull;
    for ( int i=0; i < FNV1024size/2; ++i ) {
        uint32_t temp = *basis++;
        ctx->Hash[i] = ( temp<<8 ) + *basis++;
    }
    ctx->Computed = FNVinited+FNV1024state;
    return fnvSuccess;
}   /* end FNV1024initBasis */

/* hash in a counted block  (32-bit)
 *****************************************************************/
int FNV1024blockin ( FNV1024context * const 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;
    if ( length < 0 )
        return fnvBadParam;
    switch ( ctx->Computed ) {
        case FNVinited+FNV1024state:
            ctx->Computed = FNVcomputed+FNV1024state;
            break;
        case FNVcomputed+FNV1024state:
            break;
        default:
            return fnvStateError;
    }
    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 zero-terminated string not including the zero  (32-bit)
 ********************************************************************/
int FNV1024stringin ( FNV1024context * const 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;
            break;
        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 * const ctx,
                    uint8_t out[FNV1024size] ) {
    if ( !ctx || !out )
        return fnvNull;
    if ( ctx->Computed != FNVcomputed+FNV1024state )
        return fnvStateError;
    for ( int i=0; i<FNV1024size/2; ++i ) {
        out[2*i] = ctx->Hash[i] >> 8;
        out[2*i+1] = ctx->Hash[i];
        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
//*****************************************************************//

<CODE ENDS>
```

8.3. FNV Test Code

Below is source code for a test driver with a command line interface as documented in Section 8.1.3. By default, with no command line arguments, it runs tests of all FNV lengths.

```
<CODE BEGINS> file "main.c"

//************************** Main.c **************************//
//*************** See RFC 9923 for details. *****************//
/* Copyright (c) 2016-2025 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 <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <errno.h>

/* To do a thorough test, you need to run with
 * FNV_64bitIntegers defined and with it undefined
 */
#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 = "funcName not set?";
const char        *errteststring = "foo";
int               Terr = -1; /* Total errors */
int               verbose = 0; /* Verbose flag */
enum { FNV32selected = 0, FNV64selected, FNV128selected,
       FNV256selected, FNV512selected, FNV1024selected,
       FNVnone = -1 } selected = FNVnone;
```

```
#define NTestBytes 3
const uint8_t   errtestbytes[NTestBytes] = { (uint8_t)1,
    (uint8_t)2, (uint8_t)3 };

// initial teststring is null, so initial result is offset_basis
const char *teststring[] = {
        "",
        "a",
        "foobar",
        "Hello!\x01\xFF\xED"
};
#define NTstrings (sizeof(teststring)/sizeof(char *))

// due to FNV-1 versus FNV1a, XOR in final backslash separately
const char      BasisString[] = "chongo <Landon Curt Noll> /\\../";
FNV32context    e32Context;
uint32_t        eUint32 = 42;
#ifdef FNV_64bitIntegers
  uint64_t      eUint64 = 42;
#endif
FNV64context    e64Context;
FNV128context   e128Context;
FNV256context   e256Context;
FNV512context   e512Context;
FNV1024context  e1024Context;
uint8_t         hash[FNV1024size];  /* largest size needed */
uint8_t         FakeBasis[FNV1024size];
uint8_t         ZeroBasis[FNV1024size];
char            tempFileNameTemplate[] = "tmp.XXXXXXXXXX";
const char      *tempFileName = 0;

//*****************************************************************
// local prototypes in alphabetical order
//*****************************************************************
void CommonTest ( void );
void ErrTestReport ( void );
int find_selected(const char *optarg);
void HexPrint ( int count, const uint8_t *ptr );
void TestAll ( void );
void Test32 ( void );
void Test64 ( void );
void Test128 ( void );
void Test256 ( void );
void Test512 ( void );
void Test1024 ( void );
void TestNValue ( const char *subfunc,  // test calculated value
                  const char *string,
                  int N,                // size
                  const uint8_t *was,
                  const uint8_t should[N] );
int TestR ( const char *,
            int expect,
            int actual ); // test return code
void usage( const char *argv0 ); // print help message
void ValueTestReport ( void );      // print test results

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

// array of function pointers, etc.
struct { // sometimes indexed into by the enum variable "selected"
    int length;
    void (*Testfunc)( void );
    int (*Stringfunc)( const char *, uint8_t *); // string
    int (*Blockfunc)( const void *, long int, uint8_t *); // block
    int (*Filefunc)( const char *, uint8_t *); // file
    int (*StringBasisfunc)
        ( const char *, uint8_t *, const uint8_t *); // stringBasis
    int (*BlockBasisfunc)
        (const void *, long int, uint8_t *,
         const uint8_t *); // blockBasis
    int (*FileBasisfunc)
        (const char *, uint8_t *, const uint8_t *); // fileBlock
} funcmap[] = {  // valid sizes
    { 32, Test32, FNV32string, FNV32block, FNV32file,
      FNV32stringBasis, FNV32blockBasis, FNV32fileBasis },
    { 64, Test64, FNV64string, FNV64block, FNV64file,
      FNV64stringBasis, FNV64blockBasis, FNV64fileBasis },
    { 128, Test128, FNV128string, FNV128block, FNV128file,
      FNV128stringBasis, FNV128blockBasis, FNV128fileBasis },
    { 256, Test256, FNV256string, FNV256block, FNV256file,
      FNV256stringBasis, FNV256blockBasis, FNV256fileBasis },
    { 512, Test512, FNV512string, FNV512block, FNV512file,
      FNV512stringBasis, FNV512blockBasis, FNV512fileBasis },
    { 1024, Test1024, FNV1024string, FNV1024block, FNV1024file,
      FNV1024stringBasis, FNV1024blockBasis, FNV1024fileBasis },
    { 0, Test32, FNV32string, FNV32block, FNV32file }  // fence post
};

//*****************************************************************
//  main
//*****************************************************************
int main( int argc, const char **argv ) {
    int option;  // command line option letter
    int i;
    uint16_t endianness = 5*256 + 11;

    mkstemp(tempFileNameTemplate);
    tempFileName = tempFileNameTemplate;

    if ( ((uint8_t *)&endianness)[0] != 11 )
        printf ("Coded for little endian but computer seems\n"
                " to be big endian!  Multi-byte integer results\n"
                " may be incorrect!\n");
    for ( i=0; i<FNV1024size; ++i ) {// initialize a couple of arrays
        ZeroBasis[i] = 0;
        FakeBasis[i] = (uint8_t)i;
    }
    if ( argc == 1 ) {  // if no arguments
       TestAll();
        if ( tempFileName )
            unlink(tempFileName);
        exit(0);
     }
```

```
// process command line options
//*****************************************************************
    while ((option = getopt(argc, (char *const *)argv, ":af:ht:u:v"))
           != -1) {
        if ( verbose )
            printf ( "Got option %c\n", option );
        switch ( option ) {
            case 'a':   // run all tests
                TestAll();
                break;
            case 'f':   // followed by name of file to hash
                if ( selected == FNVnone ) {
                    printf ( "No hash size selected.\n" );
                    break;
                }
                printf ( "FNV-%i Hash of contents of file '%s':\n",
                         funcmap[selected].length, optarg );
                if ( funcmap[selected].Filefunc ( optarg, hash ))
                    printf ( "Hashing file '%s' fails: %s.\n",
                        optarg, strerror(errno) );
                else
                    HexPrint ( funcmap[selected].length/8, hash );
                printf ( "\n" );
                break;
            case 'h':   // help
                usage( argv[0] );
                break;
            case 't':   // followed by size of FNV to test, 0->all
                selected = find_selected(optarg);
                if (selected == FNVnone)
                    printf ( "Bad argument to option -t\n"
                             "Valid sizes are 32, 64, 128,"
                             " 256, 512, and 1024\n" );
                else
                    funcmap[selected].Testfunc();   // invoke test
                break;
            case 'u':   // followed by size of FNV to use
                selected = find_selected(optarg);
                if ( selected == FNVnone )
                    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) ) {
                    printf ( "Verbose on.\n" );
#ifdef FNV_64bitIntegers
                    printf ("Has 64-bit integers. ");
#else
                    printf ("Does not have 64-bit integers. ");
#endif
                    // also tests the TestR function
                    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 );
                usage( argv[0] );
                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 ) {
        int rc;  // return code

        if ( selected == FNVnone ) {
            printf ( "No hash size selected.\n" );
            break;  // out of for
        }
        rc = funcmap[selected].Stringfunc(argv[i], hash);
        if ( rc )
            printf ( "FNV-%i of '%s' returns error %i\n",
                    funcmap[selected].length,
                    argv[i], rc );
        else {
            printf ( "FNV-%i of '%s' is ",
                    funcmap[selected].length, argv[i] );
            HexPrint ( funcmap[selected].length/8, hash );
            printf ( "\n" );
        }
    }
    if ( tempFileName )
        unlink(tempFileName);
    return 0;
}   /* end main */

/* Write to a temp file
 ******************************************************************/
const char *WriteTemp( const char *str, long int iLen ) {
    FILE *fp = fopen( tempFileName, "w" );
    if (!fp) {
        printf ( "Cannot open tempfile: %s: %s\n",
                 tempFileName, strerror(errno) );
        return 0;
    }
    long int ret = fwrite( str, 1, iLen, fp );
    fclose(fp);
    if ( ret != iLen ) {
        printf ( "Cannot write tempfile: %s: %s\n",
                 tempFileName, strerror(errno) );
        return 0;
    }
    return tempFileName;
}

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

```
//  Test status return code
//*****************************************************************
int TestR ( const 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 */

//*****************************************************************
//  General byte vector return value test
//*****************************************************************
void TestNValue ( const char *subfunc,
                  const char *string, // usually what was hashed
                  int N,
                  const uint8_t was[N],
                  const uint8_t should[N] ) {
    if ( memcmp ( was, should, N ) != 0 ) {
        ++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 */

//*****************************************************************
//  Reports 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 */

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

```
//  Verify the size of hash as a command line option argument
//    and return the index in funcmap[], -1 if not found.
//*****************************************************************
int find_selected(const char *optarg) {
    int argval, count;

    count = sscanf ( optarg, "%i", &argval );
    if ( count > 0 ) {
      int i;
        for ( i = 0; funcmap[i].length; ++i ) {
            if ( funcmap[i].length == argval ) {
                return i;
            }  /* end if */
        }  /* end for */
    }
    return FNVnone;
}   /* end find_selected */

//*****************************************************************
//  Print some bytes as hexadecimal
//*****************************************************************
void HexPrint( int count, const uint8_t *ptr ) {
    for ( int i = 0; i < count; ++i )
        printf ( "%02X", ptr[i] );
}   /* end HexPrint */

//*****************************************************************
//  Test all sizes
//*****************************************************************
void TestAll ( void ) {
    for ( int i=0; funcmap[i].length; ++i )
        funcmap[i].Testfunc ();
}   /* end TestAll */

//*****************************************************************
//  Common error check tests
//*****************************************************************
void CommonTest ( void ) {
    TestR ( "string1b", fnvNull,
        funcmap[selected].Stringfunc ( (char *)0, hash ) );
    TestR ( "string2b", fnvNull,
        funcmap[selected].Stringfunc ( errteststring,
                                      (uint8_t *)0 ) );
    TestR ( "strBasis1b", fnvNull,
           funcmap[selected].StringBasisfunc ( (char *)0,
               hash, FakeBasis ) );
    TestR ( "strBasis2b", fnvNull,
           funcmap[selected].StringBasisfunc ( errteststring,
               (uint8_t *)0, FakeBasis ) );
    TestR ( "strBasis3b", fnvNull,
           funcmap[selected].StringBasisfunc ( errteststring,
               hash, (uint8_t *)0 ) );
    TestR ( "blk1", fnvNull,
        funcmap[selected].Blockfunc ( (uint8_t *)0, 1, hash ) );
    TestR ( "blk2", fnvBadParam,
        funcmap[selected].Blockfunc ( errtestbytes, -1, hash ) );
    TestR ( "blk3", fnvNull,
        funcmap[selected].Blockfunc ( errtestbytes, 1,
                                     (uint8_t *)0 ) );
    TestR ( "blk1b", fnvNull,
        funcmap[selected].BlockBasisfunc ( (uint8_t *)0, 1,
                                           hash, FakeBasis ) );
    TestR ( "blk2b", fnvBadParam,
        funcmap[selected].BlockBasisfunc ( errtestbytes, -1,
                                           hash, FakeBasis ) );
    TestR ( "blk3b", fnvNull,
        funcmap[selected].BlockBasisfunc ( errtestbytes, 1,
                                     (uint8_t *)0, FakeBasis ) );
    TestR ( "blk4b", fnvNull,
        funcmap[selected].BlockBasisfunc ( errtestbytes, 1,
                                           hash, (uint8_t *)0 ) );
    TestR ( "file1", fnvNull,
        funcmap[selected].Filefunc ( (char *)0, hash ) );
    TestR ( "file2", fnvNull,
        funcmap[selected].Filefunc ( "foo.txt", (uint8_t *)0 ) );
    TestR ( "file1b", fnvNull,
        funcmap[selected].FileBasisfunc ( (char *)0, hash,
                                          FakeBasis ) );
    TestR ( "file2b", fnvNull,
        funcmap[selected].FileBasisfunc ( "foo.txt", (uint8_t *)0,
                                          FakeBasis ) );
    TestR ( "file3b", fnvNull,
        funcmap[selected].FileBasisfunc ( "foo.txt", hash,
                                         (uint8_t *)0 ) );
}   /* end CommonTest */

//*****************************************************************
//  Print command line help
//*****************************************************************
void usage( const char *argv0 ) {
    printf (
        "%s [-a] [-t nnn] [-u nnn] [-v] [-f filename] [token ...]\n"
        "  -a = run all tests\n"
        "  -f filename = hash file contents\n"
        "  -h = help, print this message\n"
        "  -t nnn = Test hash size nnn\n"
        "  -u nnn = Use hash size nnn\n"
        "  -v = toggle Verbose flag\n"
        "  Each token is hashed.\n", argv0 );
}   /* end usage */

//*****************************************************************
//  Test Macros
//*****************************************************************

// test for return values
//************************
#define TestInit(INIT,CTX,CTXT)                                \
TestR ( "init1", fnvSuccess, INIT ( &CTX ) );                  \
TestR ( "init2", fnvNull, INIT ( (CTXT *)0 ) );

#define TestInitBasis(INITB,CTX,CTXT)                          \
TestR ( "initB1", fnvSuccess, INITB (&CTX, FakeBasis ) );      \
TestR ( "initB2", fnvNull, INITB ( (CTXT *)0, hash ) );        \
TestR ( "initB3", fnvNull, INITB ( &CTX, (uint8_t *)0 ) );
```

```
#define TestBlockin(BLKIN,CTX,CTXT)                             \
TestR ( "blockin1", fnvNull,                                    \
    BLKIN ( (CTXT *)0, errtestbytes, NTestBytes ) );            \
TestR ( "blockin2", fnvNull,                                    \
    BLKIN ( &CTX, (uint8_t *)0, NTestBytes ) );                 \
TestR ( "blockin3", fnvBadParam,                                \
    BLKIN ( &CTX, errtestbytes, -1 ) );                         \
TestR ( "blockin4", fnvStateError,                              \
    BLKIN ( &CTX, errtestbytes, NTestBytes ) );

#define TestStringin(STRIN,CTX,CTXT)                            \
TestR ( "stringin1", fnvNull,                                   \
    STRIN ( (CTXT *)0, errteststring ) );                       \
TestR ( "stringin2", fnvNull, STRIN ( &CTX, (char *)0 ) );      \
TestR ( "stringin3", fnvStateError,                             \
    STRIN ( &CTX, errteststring ) );

#define TestFilein(FLIN,CTX,CTXT)                               \
TestR ( "file1", fnvNull, FLIN ( (CTXT *)0, errteststring ) );  \
TestR ( "file2", fnvNull, FLIN ( &CTX, (char *)0 ) );           \
TestR ( "file3", fnvStateError,                                 \
    FLIN ( &CTX, errteststring ) );

#define TestResult(RSLT,CTX,CTXT)                               \
TestR ( "result1", fnvNull, RSLT ( (CTXT *)0, hash ) );         \
TestR ( "result2", fnvNull, RSLT ( &CTX, (uint8_t *)0 ) );      \
TestR ( "result3", fnvStateError,                               \
            FNV128result ( &e128Context, hash ) );

// test return values for INT versions including non-std basis
//*************************************************************
#define TestINT(STRINT,STRINTB,BLKINT,BLKINTB,INITINTB,         \
                INTV,INTVT,ctxT)                                \
TestR ( "string1i", fnvNull, STRINT ( (char *)0, &INTV ) );     \
TestR ( "string2i", fnvNull,                                    \
        STRINT ( errteststring, (INTVT *)0 ) );                 \
TestR ( "string3i", fnvNull, STRINTB ((char *)0, &INTV, INTV) );\
TestR ( "string4i", fnvNull,                                    \
        STRINTB (errteststring, (INTVT *)0, INTV) );            \
TestR ( "block1i", fnvNull, BLKINT ( (uint8_t *)0, 1, &INTV ) );\
TestR ( "block2i", fnvBadParam,                                 \
        BLKINT ( errtestbytes, -1, &INTV ) );                   \
TestR ( "block3i", fnvNull,                                     \
        BLKINT ( errtestbytes, 1, (INTVT *)0 ) );               \
TestR ( "block4i", fnvNull,                                     \
        BLKINTB ( (uint8_t *)0, 1, &INTV, INTV ) );             \
TestR ( "block5i", fnvBadParam,                                 \
        BLKINTB ( errtestbytes, -1, &INTV, INTV ) );            \
TestR ( "block6i", fnvNull,                                     \
        BLKINTB ( errtestbytes, 1, (INTVT *)0, INTV ) );        \
TestR ( "initBasis1i", fnvNull, INITINTB ( (ctxT *)0, INTV ) );

#define TestINTrf(RSLTINT,FILEINT,FILEINTB,                     \
                  ctx,ctxT,INTV,INTVT)                          \
TestR ( "result1i", fnvNull, RSLTINT ( (ctxT *)0, &INTV ) );    \
TestR ( "result2i", fnvNull, RSLTINT ( &ctx, (INTVT *)0 ) );    \
TestR ( "result3i", fnvStateError, RSLTINT ( &ctx, &INTV ) );   \
TestR ( "file1i", fnvNull, FILEINT ( (char *)0, &INTV ) );      \
TestR ( "file2i", fnvNull, FILEINT ( "foo.txt", (INTVT *)0 ) ); \
TestR ( "file3i", fnvNull, FILEINTB ( (char *)0, &INTV, INTV) );\
TestR ( "file4i", fnvNull,                                       \
        FILEINTB ( "foo.txt", (INTVT *)0, INTV ) );

// test to calculate standard basis from basis zero FNV-1
// depends on zero basis making the initial multiply a no-op
//*****************************
#define BasisZero(STRING,SIZ,VALUE)                              \
err = TestR ( "fnv0s", fnvSuccess,                               \
              STRING ( BasisString, hash, ZeroBasis ) );         \
if ( err == fnvSuccess ) {                                       \
    hash[SIZ-1] ^= '\\';                                         \
    TestNValue ( "fnv0sv", BasisString, SIZ, hash, VALUE[0] );  \
}
#define BasisINTZero(STRINT,SIZ,VALUE,INTV,INTVT)                \
err = TestR ( "fnv0s", fnvSuccess,                               \
              STRINT ( BasisString, &INTV, (INTVT) 0 ) );        \
if ( err == fnvSuccess ) {                                       \
    INTV ^= '\\';                                                \
    TestNValue ( "fnv0svi", BasisString, SIZ,                    \
                (uint8_t *)&INTV, (uint8_t *)&VALUE[0] );        \
}

// test for return hash values
//*****************************
#define TestSTRBLKHash(STR,BLK,SVAL,BVAL,SZ)                     \
if ( TestR ( "stringa", fnvSuccess,                              \
             STR ( teststring[i], hash ) ) )                     \
    printf ( "  Index = %i\n", i );                              \
else                                                             \
    TestNValue ( "stringb", teststring[i], SZ,                   \
                 hash, (uint8_t *)&SVAL[i] );                    \
if ( TestR ( "blocka", fnvSuccess, BLK ( teststring[i],          \
             (long int)(strlen(teststring[i])+1), hash ) ) )     \
    printf ( "  Index = %i\n", i );                              \
else                                                             \
    TestNValue ( "blockb", teststring[i], SZ,                    \
                 hash, (uint8_t *)&BVAL[i] );

// Test incremental functions
//****************************
#define IncrHash(INIT,CTX,BLK,RSLT,INITB,STR,SZ,SVAL)            \
err = TestR ( "inita", fnvSuccess, INIT ( &CTX ) );              \
if ( err ) break;                                                \
iLen = strlen ( teststring[i] );                                 \
err = TestR ( "blockina", fnvSuccess,                            \
             BLK ( &CTX, (uint8_t *)teststring[i], iLen/2 ) );  \
if ( err ) break;                                                \
if ( i & 1 ) {                                                   \
    err = TestR ( "basisra", fnvSuccess, RSLT ( &CTX, hash ) ); \
    if ( err ) break;                                            \
    err = TestR ( "basisia", fnvSuccess, INITB ( &CTX, hash ) );\
    if ( err ) break;                                            \
}                                                                \
err = TestR ( "stringina", fnvSuccess, STR ( &CTX,               \
              teststring[i] + iLen/2 ) );                        \
if ( err ) break;                                                \
err = TestR ( "resulta", fnvSuccess, RSLT ( &CTX, hash ) );     \
if ( err ) break;                                               \
TestNValue ( "incrementala", teststring[i], SZ,                 \
             hash, (uint8_t *)&SVAL[i] );

// test file hash
//****************************
#define TestFILEHash(FILE,BVAL,SZ)                              \
err = TestR ( "fileafh", fnvSuccess,                            \
               FILE ( WriteTemp(teststring[i], iLen),           \
                      hash ) );                                 \
if ( err ) break;                                               \
TestNValue ( "filebfh", teststring[i], SZ, hash,                \
             (uint8_t *)&BVAL[i] );

//*****************************************************************
//  FNV32 Test
//*****************************************************************
void Test32 ( void ) {
    long int iLen;
    uint32_t FNV32svalues[NTstrings] = {
        0x811c9dc5, 0xe40c292c, 0xbf9cf968, 0xfd9d3881 };
    uint32_t FNV32bvalues[NTstrings] = {
        0x050c5d1f, 0x2b24d044, 0x0c1c9eb8, 0xbf7ff313 };
    int i, err;
    uint8_t FNV32basisT[FNV32size] = {0xC5, 0x9D, 0x1C, 0x81 };

    funcName = "FNV-32";
    selected = FNV32selected;
/* test error checks */
    Terr = 0;
    TestInit (FNV32init, e32Context, FNV32context)
    TestInitBasis (FNV32initBasis, e32Context, FNV32context)
    CommonTest();
    TestINT (FNV32INTstring, FNV32INTstringBasis, FNV32INTblock,
             FNV32INTblockBasis, FNV32INTinitBasis, eUint32,
             uint32_t, FNV32context)
    e32Context.Computed = FNVclobber+FNV32state;
    TestBlockin (FNV32blockin, e32Context, FNV32context)
    TestStringin (FNV32stringin, e32Context, FNV32context)
    TestFilein (FNV32filein, e32Context, FNV32context)
    TestResult (FNV32result, e32Context, FNV32context)
    TestINTrf(FNV32INTresult,FNV32INTfile,FNV32INTfileBasis,
              e32Context,FNV32context,eUint32,uint32_t)
    ErrTestReport ();
    Terr = 0;
    err = TestR ( "fnv0s", fnvSuccess,
                  FNV32stringBasis ( BasisString, hash, ZeroBasis ) );
    if ( err == fnvSuccess ) {
        hash[0] ^= '\\';
        TestNValue ( "fnv0sv32", BasisString, FNV32size,
                     hash, (uint8_t *)&FNV32svalues[0]);
    }
    BasisINTZero (FNV32INTstringBasis,FNV32size,FNV32svalues, \
                  eUint32,uint32_t)
    for ( i = 0; i < NTstrings; ++i ) {
/* test actual results int */
        err = TestR ( "stringai", fnvSuccess,
                      FNV32INTstring ( teststring[i], &eUint32 ) );
        if ( err == fnvSuccess )
            TestNValue ( "stringbi", teststring[i], FNV32size,
                         (uint8_t *)&eUint32,
                         (uint8_t *)&FNV32svalues[i] );
        err = TestR ( "blockai", fnvSuccess,
                      FNV32INTblock ( (uint8_t *)teststring[i],
                           (unsigned long)(strlen(teststring[i])+1),
                           &eUint32 ) );
        if ( err == fnvSuccess )
            TestNValue ( "blockbi", teststring[i], FNV32size,
                         (uint8_t *)&eUint32,
                         (uint8_t *)&FNV32bvalues[i] );
/* test actual results byte */
        TestSTRBLKHash ( FNV32string, FNV32block, FNV32svalues,
                         FNV32bvalues, FNV32size )
/* now try testing the incremental stuff */
        IncrHash (FNV32init, e32Context, FNV32blockin, FNV32result,
            FNV32initBasis, FNV32stringin, FNV32size, FNV32svalues)
/* now try testing the incremental stuff int */
        err = TestR ( "initai", fnvSuccess,
                     FNV32init (&e32Context) );
        if ( err ) break;
        iLen = strlen ( teststring[i] );
        err = TestR ( "blockinai", fnvSuccess,
                     FNV32blockin ( &e32Context,
                                   (uint8_t *)teststring[i],
                                   iLen/2 ) );
        if ( err ) break;
        err = TestR ( "stringinai", fnvSuccess,
                     FNV32stringin ( &e32Context,
                                    teststring[i] + iLen/2 ) );
        if ( err ) break;
        err = TestR ( "resultai", fnvSuccess,
                     FNV32INTresult ( &e32Context, &eUint32 ) );
        if ( err ) break;
        TestNValue ( "incrementalai", teststring[i], FNV32size,
                    (uint8_t *)&eUint32,
                    (uint8_t *)&FNV32svalues[i] );
/* now try testing the incremental stuff byte basis */
        err = TestR ( "initab", fnvSuccess,
                     FNV32initBasis (&e32Context,
                                     (uint8_t *)&FNV32basisT) );
        if ( err ) break;
        iLen = strlen ( teststring[i] );
        err = TestR ( "blockinab", fnvSuccess,
                     FNV32blockin ( &e32Context,
                                   (uint8_t *)teststring[i],
                                   iLen/2 ) );
        if ( err ) break;
        err = TestR ( "stringinab", fnvSuccess,
                     FNV32stringin ( &e32Context,
                                    teststring[i] + iLen/2 ) );
        if ( err ) break;
        err = TestR ( "resultab", fnvSuccess,
                     FNV32result ( &e32Context, hash ) );
        if ( err ) break;
        TestNValue ( "incrementala", teststring[i], FNV32size,
                    hash, (uint8_t *)&FNV32svalues[i] );
/* now try testing file hash int */
        err = TestR ( "fileafi", fnvSuccess,
                      FNV32INTfile ( WriteTemp(teststring[i], iLen),
                                     &eUint32 ) );
        if ( err ) break;
        TestNValue ( "filebfi", teststring[i], FNV32size,
                         (uint8_t *)&eUint32,
                         (uint8_t *)&FNV32svalues[i] );

/* now try testing file hash byte */
        TestFILEHash ( FNV32file, FNV32svalues, FNV32size )
    }   // end for i
    ValueTestReport ();
}    /* end Test32 */

#ifdef FNV_64bitIntegers
//*****************************************************************
//  Code for testing FNV64 using 64-bit integers
//*****************************************************************
void Test64 ( void ) { /* with 64-bit integers */
    long int iLen;
    uint64_t FNV64basisT = FNV64basis;
    uint64_t FNV64svalues[NTstrings] = {
        0xcbf29ce484222325, 0xaf63dc4c8601ec8c, 0x85944171f73967e8,
        0xbd51ea7094ee6fa1 };
    uint64_t FNV64bvalues[NTstrings] = {
        0xaf63bd4c8601b7df, 0x089be207b544f1e4, 0x34531ca7168b8f38,
        0xa0a0fe4d1127ae93 };
    int i, err;

    funcName = "FNV-64";
    selected = FNV64selected;
/* test error checks */
    Terr = 0;
    TestInit (FNV64init, e64Context, FNV64context)
    TestInitBasis (FNV64initBasis, e64Context, FNV64context)
    CommonTest();
    TestINT(FNV64INTstring,FNV64INTstringBasis,FNV64INTblock,
            FNV64INTblockBasis,FNV64INTinitBasis,
            eUint64,uint64_t,FNV64context)
    e64Context.Computed = FNVclobber+FNV64state;
    TestBlockin (FNV64blockin, e64Context, FNV64context)
    TestStringin (FNV64stringin, e64Context, FNV64context)
    TestFilein (FNV64filein, e64Context, FNV64context)
    TestResult (FNV64result, e64Context, FNV64context)
    TestINTrf(FNV64INTresult,FNV64INTfile,FNV64INTfileBasis,
              e64Context,FNV64context,eUint64,uint64_t)
    ErrTestReport ();
/* test actual results int */
    Terr = 0;
    err = TestR ( "fnv0s", fnvSuccess,
                 FNV64stringBasis ( BasisString, hash, ZeroBasis ) );
    if ( err == fnvSuccess ) {
        hash[0] ^= '\\';
        TestNValue ( "fnv0sv64", BasisString, FNV64size,
                     hash, (uint8_t *)&FNV64svalues[0]);
    }
```

```
     BasisINTZero (FNV64INTstringBasis,FNV64size,FNV64svalues, \
                   eUint64,uint64_t)
     for ( i = 0; i < NTstrings; ++i ) {
/* test actual results int */
         err = TestR ( "stringai", fnvSuccess,
                       FNV64INTstring ( teststring[i], &eUint64 ) );
         if ( err == fnvSuccess )
             TestNValue ( "stringbi", teststring[i], FNV64size,
                          (uint8_t *)&eUint64,
                          (uint8_t *)&FNV64svalues[i] );
         err = TestR ( "blockai", fnvSuccess,
                   FNV64INTblock ( (uint8_t *)teststring[i],
                        (unsigned long)(strlen(teststring[i])+1),
                                   &eUint64 ) );
         if ( err == fnvSuccess )
             TestNValue ( "blockbi", teststring[i], FNV64size,
                          (uint8_t *)&eUint64,
                          (uint8_t *)&FNV64bvalues[i] );
/* test actual results byte */
         TestSTRBLKHash ( FNV64string, FNV64block, FNV64svalues,
                          FNV64bvalues, FNV64size )
/* now try testing the incremental stuff */
         IncrHash (FNV64init, e64Context, FNV64blockin, FNV64result,
             FNV64initBasis, FNV64stringin, FNV64size, FNV64svalues)
/* now try testing the incremental stuff int */
         err = TestR ( "initai", fnvSuccess,
                      FNV64init (&e64Context) );
         if ( err ) break;
         iLen = strlen ( teststring[i] );
         err = TestR ( "blockinai", fnvSuccess,
                      FNV64blockin ( &e64Context,
                                    (uint8_t *)teststring[i],
                                    iLen/2 ) );
         if ( err ) break;
         err = TestR ( "stringinai", fnvSuccess,
                      FNV64stringin ( &e64Context,
                                     teststring[i] + iLen/2 ) );
         if ( err ) break;
         err = TestR ( "resultai", fnvSuccess,
                      FNV64INTresult ( &e64Context, &eUint64 ) );
         if ( err ) break;
         TestNValue ( "incrementalai", teststring[i], FNV64size,
                     (uint8_t *)&eUint64,
                     (uint8_t *)&FNV64svalues[i] );
/* now try testing the incremental stuff byte basis */
         err = TestR ( "initab", fnvSuccess,
                      FNV64initBasis (&e64Context,
                                      (uint8_t *)&FNV64basisT) );
         if ( err ) break;
         iLen = strlen ( teststring[i] );
         err = TestR ( "blockinab", fnvSuccess,
                      FNV64blockin ( &e64Context,
                                    (uint8_t *)teststring[i],
                                    iLen/2 ) );
         if ( err ) break;
         err = TestR ( "stringinab", fnvSuccess,
                      FNV64stringin ( &e64Context,
                                     teststring[i] + iLen/2 ) );
```

```
        if ( err ) break;
        err = TestR ( "resultab", fnvSuccess,
                      FNV64result ( &e64Context, hash ) );
        if ( err ) break;
        TestNValue ( "incrementala", teststring[i], FNV64size,
                     hash, (uint8_t *)&FNV64svalues[i] );
/* now try testing file int */
        err = TestR ( "fileafi", fnvSuccess,
                       FNV64INTfile ( WriteTemp(teststring[i], iLen),
                                      &eUint64 ) );
        if ( err ) break;
        TestNValue ( "filebfi", teststring[i], FNV64size,
                          (uint8_t *)&eUint64,
                          (uint8_t *)&FNV64svalues[i] );
/* now try testing file hash */
        TestFILEHash(FNV64file,FNV64svalues,FNV64size)
    }
    ValueTestReport ();
}   /* end Test64 */

#else

//*****************************************************************
//  Code for testing 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";
    selected = FNV64selected;
/* test error checks */
    Terr = 0;
    TestR ( "init1", fnvSuccess, FNV64init (&e64Context) );
    CommonTest();
    TestInit (FNV64init, e64Context, FNV64context)
    TestInitBasis (FNV64initBasis, e64Context, FNV64context)
    e64Context.Computed = FNVclobber+FNV64state;
    TestBlockin (FNV64blockin, e64Context, FNV64context)
    TestStringin (FNV64stringin, e64Context, FNV64context)
    TestFilein (FNV64filein, e64Context, FNV64context)
    TestResult (FNV64result, e64Context, FNV64context)
    ErrTestReport ();
/* test actual results */
    Terr = 0;
    BasisZero(FNV64stringBasis,FNV64size,FNV64svalues)
    for ( i = 0; i < NTstrings; ++i ) {
        TestSTRBLKHash ( FNV64string, FNV64block,
                          FNV64svalues, FNV64bvalues, FNV64size )
/* try testing the incremental stuff */
        IncrHash(FNV64init,e64Context,FNV64blockin,FNV64result,
            FNV64initBasis,FNV64stringin,FNV64size,FNV64svalues)
/* now try testing file hash */
        TestFILEHash(FNV64file,FNV64svalues,FNV64size)
    }
    ValueTestReport ();
}   /* end Test64 */
#endif /* FNV_64bitIntegers */

//*****************************************************************
//  Code for testing 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";
    selected = FNV128selected;
/* test error checks */
    Terr = 0;
    TestInit (FNV128init, e128Context, FNV128context)
    TestInitBasis (FNV128initBasis, e128Context, FNV128context)
    CommonTest();
    e128Context.Computed = FNVclobber+FNV128state;
    TestBlockin (FNV128blockin, e128Context, FNV128context)
    TestStringin (FNV128stringin, e128Context, FNV128context)
    TestFilein (FNV128filein, e128Context, FNV128context)
    TestResult (FNV128result, e128Context, FNV128context)
    ErrTestReport ();
/* test actual results */
    Terr = 0;
    BasisZero(FNV128stringBasis,FNV128size,FNV128svalues)
    for ( i = 0; i < NTstrings; ++i ) {
        TestSTRBLKHash ( FNV128string, FNV128block,
                          FNV128svalues, FNV128bvalues, FNV128size )
/* try testing the incremental stuff */
        IncrHash(FNV128init,e128Context,FNV128blockin,FNV128result,
            FNV128initBasis,FNV128stringin,FNV128size,FNV128svalues)
```

```
/* now try testing file hash */
    TestFILEHash(FNV128file,FNV128svalues,FNV128size)
    }
    ValueTestReport ();
}    /* end Test128 */

//*****************************************************************
//  Code for testing 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";
    selected = FNV256selected;
/* test error checks */
    Terr = 0;
    TestInit (FNV256init, e256Context, FNV256context)
    TestInitBasis (FNV256initBasis, e256Context, FNV256context)
    CommonTest();
    e256Context.Computed = FNVclobber+FNV256state;
    TestBlockin (FNV256blockin, e256Context, FNV256context)
    TestStringin (FNV256stringin, e256Context, FNV256context)
    TestFilein (FNV256filein, e256Context, FNV256context)
    TestResult (FNV256result, e256Context, FNV256context)
    ErrTestReport ();
/* test actual results */
    Terr = 0;
    BasisZero(FNV256stringBasis,FNV256size,FNV256svalues)
    for ( i = 0; i < NTstrings; ++i ) {
        TestSTRBLKHash ( FNV256string, FNV256block,
                         FNV256svalues, FNV256bvalues, FNV256size )
/* try testing the incremental stuff */
        IncrHash(FNV256init,e256Context,FNV256blockin,FNV256result,
            FNV256initBasis,FNV256stringin,FNV256size,FNV256svalues)
/* now try testing file hash */
        TestFILEHash(FNV256file,FNV256svalues,FNV256size)
   }
    ValueTestReport ();
}    /* end Test256 */

//*****************************************************************
//  Code for testing 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";
    selected = FNV512selected;
/* test error checks */
    Terr = 0;
    TestInit (FNV512init, e512Context, FNV512context)
    TestInitBasis (FNV512initBasis, e512Context, FNV512context)
    CommonTest();
    e512Context.Computed = FNVclobber+FNV512state;
    TestBlockin (FNV512blockin, e512Context, FNV512context)
    TestStringin (FNV512stringin, e512Context, FNV512context)
    TestFilein (FNV512filein, e512Context, FNV512context)
    TestResult (FNV512result, e512Context, FNV512context)
    ErrTestReport ();
/* test actual results */
    Terr = 0;
    BasisZero(FNV512stringBasis,FNV512size,FNV512svalues)
    for ( i = 0; i < NTstrings; ++i ) {
        TestSTRBLKHash ( FNV512string, FNV512block,
                         FNV512svalues, FNV512bvalues, FNV512size )
/* try testing the incremental stuff */
        IncrHash(FNV512init,e512Context,FNV512blockin,FNV512result,
            FNV512initBasis,FNV512stringin,FNV512size,FNV512svalues)
/* now try testing file hash */
        TestFILEHash(FNV512file,FNV512svalues,FNV512size)
    }
    ValueTestReport ();
```

```
}    /* end Test512 */

//*****************************************************************
//  Code for testing FNV1024
//*****************************************************************
void Test1024 ( void ) {
    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 } };
    long int iLen;
    int i, err;

    funcName = "FNV-1024";
    selected = FNV1024selected;
    /* test error checks */
    Terr = 0;
    TestInit (FNV1024init, e1024Context, FNV1024context)
    TestInitBasis (FNV1024initBasis, e1024Context, FNV1024context)
    CommonTest();
    e1024Context.Computed = FNVclobber+FNV1024state;
    TestBlockin (FNV1024blockin, e1024Context, FNV1024context)
    TestStringin (FNV1024stringin, e1024Context, FNV1024context)
    TestFilein (FNV1024filein, e1024Context, FNV1024context)
    TestResult (FNV1024result, e1024Context, FNV1024context)
    ErrTestReport ();
/* test actual results */
    Terr = 0;
    BasisZero(FNV1024stringBasis,FNV1024size,FNV1024svalues)
    for ( i = 0; i < NTstrings; ++i ) {
        TestSTRBLKHash ( FNV1024string, FNV1024block,
                         FNV1024svalues, FNV1024bvalues,
                                     FNV1024size )
/* try testing the incremental stuff */
        IncrHash(FNV1024init,e1024Context,FNV1024blockin,
                 FNV1024result, FNV1024initBasis,
                 FNV1024stringin,FNV1024size,FNV1024svalues)
/* now try testing file hash */
        TestFILEHash(FNV1024file,FNV1024svalues,FNV1024size)
    }
    ValueTestReport ();
}    /* end Test1024 */


<CODE ENDS>
```

8.4. Makefile

Below is a simple makefile to produce and run the test program or to provide a library with all the FNV functions supplied in it.

WARNING: When actually using the following as a makefile, the five-character sequence "<TAB>" must be changed to a tab (0x09) character!

```
<CODE BEGINS> file "makefile"

# Makefile for fnv
# If you extract this file from RFC 9923, the five-character sequence
#     <TAB> below must be replaced with a tab (0x09) character.

explanation:
<TAB>@echo Choose one of the following make targets:
<TAB>@echo make FNVhash -- test program
<TAB>@echo make libfnv.a -- library you can use
<TAB>@echo make clean -- removes all of the built targets

SRC=FNV32.c FNV64.c FNV128.c FNV256.c FNV512.c FNV1024.c
HDR=FNV32.h FNV64.h FNV128.h FNV256.h FNV512.h FNV1024.h \
<TAB>FNVconfig.h FNVErrorCodes.h fnv-private.h
OBJ=$(SRC:.c=.o)
CFLAGS=-Wall
AR=ar
ARFLAGS= rcs

FNVhash: libfnv.a main.c
<TAB>$(CC) $(CFLAGS) -o FNVhash main.c libfnv.a

libfnv.a: $(SRC) $(HDR)
<TAB>rm -f libfnv.a *.o
<TAB>$(CC) $(CFLAGS) -c $(SRC)
<TAB>$(AR) $(ARFLAGS) libfnv.a $(OBJ)

clean:
<TAB>rm -rf libfnv.a FNVhash *.o

<CODE ENDS>
```

9. IANA Considerations

This document has no IANA actions.

10. References

10.1. Normative References

[C]

Kernighan, B. W. and D. M. Ritchie, "The C Programming Language, 2nd Edition", ISBN-10 0-13-110362-8, ISBN-13 978-0131103627, 1988.

[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>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

10.2. Informative References

[BASIC] Diamond, W., "FNV32 PowerBASIC inline x86 assembler", <http://www.isthe.com/chongo/tech/comp/fnv/index.html#PowerBASIC>.

[BFDseq] Jethanandani, M., Agarwal, S., Mishra, A., Saxena, A., and A. DeKok, "Secure BFD Sequence Numbers", Work in Progress, Internet-Draft, draft-ietf-bfd-secure-sequence-numbers-09, 29 March 2022, <https://datatracker.ietf.org/doc/html/draft-ietf-bfd-secure-sequence-numbers-09>.

[calc] Bell, D. and L. Noll, "Calc - C-style arbitrary precision calculator", <http://www.isthe.com/chongo/tech/comp/calc/index.html>.

[deliantra] The Deliantra Team, "Deliantra MMORPG", 16 October 2022, <http://www.deliantra.net/>.

[fasmlab] Fasmlab, "Integrated Development Environments", <https://sourceforge.net/projects/fasmlab/>.

[FIPS202] NIST, "SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions", FIPS PUB 202, DOI 10.6028/NIST.FIPS.202, August 2015, <https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf>.

[flatassembler] Grysztar, T., "flat assembler: Assembly language resources", 2025, <https://flatassembler.net/>.

[FNV] Fowler, G., Noll, L., and K. Vo, "FNV (Fowler/Noll/Vo)", <http://www.isthe.com/chongo/tech/comp/fnv/index.html>.

[Fortran] Fortran Standard Library, "A community driven standard library for (modern) Fortran", <https://stdlib.fortran-lang.org/>.

[FragCache] Weaver, E., "Improving Running Components at Twitter", Slide 31, 2009, <https://www.slideshare.net/slideshow/improving-running-components-at-twitter/1141786>.

[FreeBSD] Baio, D. G., "FreeBSD 4.3 Release Notes (Last modified on 21 February 2021)", The Free BSD Project, 2025, <https://www.freebsd.org/releases/4.3R/notes.html>.

[FRET] McCarthy, M., "FRET: helping understand file formats", 19 January 2006, <https://fret.sourceforge.net/>.

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

[IEEE8021Q-2022] IEEE, "IEEE Standard for Local and Metropolitan Area Networks--Bridges and Bridged Networks", DOI 10.1109/IEEESTD.2022.10004498, IEEE Std 802.1Q-2022, December 2022, <https://ieeexplore.ieee.org/document/10004498>.

[IEN137] Cohen, D., "On Holy Wars and A Plea For Peace", IEN 137, 1 April 1980, <https://www.rfc-editor.org/ien/ien137.txt>.

[IPv6flow] Anderson, L., Brownlee, N., and B. E. 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://www.cs.auckland.ac.nz/~brian/flowhashRep.pdf>.

[LCN2] Noll, L. and C. Ferguson, "lcn2 / fnv", commit 953444c, 19 November 2025, <https://github.com/lcn2/fnv>.

[Leprechaun] Sanmayce project, "Sanmayce project 'Underdog Way'", <http://www.sanmayce.com/Downloads/>.

[libketama] Jones, R., "libketama: Consistent Hashing library for memcached clients", 10 April 2007, <https://www.metabrew.com/article/libketama-consistent-hashing-algo-memcached-clients>.

[libsir] Lederman, R. and J. Johnson, "libsir logging library", commit 0ae0173, 3 December 2025, <https://github.com/aremmell/libsir>.

[memcache] Dovgal, A., Joye, P., Radtke, H., Johansson, M., and T. Srnka, "PHP memcached extension", 30 April 2023, <https://pecl.php.net/package/memcache>.

[NCHF] Hayes, C. and D. Malone, "Questioning the Criteria for Evaluating Non-Cryptographic Hash Functions", Communications of the ACM, Vol. 68 No. 2, pp. 46-51, DOI 10.1145/3704255, February 2025, <https://cacm.acm.org/practice/questioning-the-criteria-for-evaluating-non-cryptographic-hash-functions/>.

Hayes, C. and D. Malone, "An Evaluation of FNV Non-Cryptographic Hash Functions", Proceedings of the 35th Irish Signals and Systems Conference (ISSC), DOI 10.1109/ISSC61953.2024.10603139, June 2024, <https://ieeexplore.ieee.org/abstract/document/10603139>.

[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>.

[RimStone] Gliim LLC, "Golf Language Hash Tables", 2025, <https://rimstone-lang.com/>.

[Smash] Emms, S., "Smash - find duplicate files super fast", 8 December 2024, <https://www.linuxlinks.com/smash-find-duplicate-files-super-fast/>.

[twistylists] Zethmayr, D., "twistylists: A no-sort namespace engine; developers invited", 6 November 2012, <https://twistylists.blogspot.com/>.

Appendix A. Work Comparison with SHA-1 and SHA-256

This appendix provides a simplistic rough comparison of the level of effort required to compute FNV-1a, SHA-1 [RFC3174], and SHA-256 [RFC6234] for short messages -- that is, those less than around 50 bytes. Some CPUs may have special instructions or other hardware to accelerate certain cryptographic operations, so if performance is particularly important for an application, benchmarking on the target platform would be appropriate.

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

SHA-1 and SHA-256 are actually designed to accept a bit vector input, although almost all computer uses apply them to an integer number of bytes. They both process blocks of 512 bits (64 bytes), and we estimate the effort involved in processing a full block. There is some overhead per message to indicate message termination and size. Assuming that the message is an even number of bytes, this overhead would be 9 bytes for SHA-1 and 17 bytes for SHA-256. So, assuming that the message with that overhead fits into one block, the message would be up to 55 bytes for SHA-1 or up to 47 bytes for SHA-256.

SHA-1 is a relatively weak cryptographic hash function producing a 160-bit hash. It has been substantially broken [RFC6194]. Ignoring SHA-1's initial setup, 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-byte block or 31.07 operations per byte for a message of 55 bytes. By this rough measure, it is a little over 15.5 times the effort of FNV.

SHA-256 is, at the time of publication, considered to be a stronger cryptographic hash function than SHA-1. Ignoring SHA-256's initial setup, transfer of control, and conditional tests, but counting all logical and arithmetic operations, SHA-1 requires 2,058 operations per 64-byte block or 48.79 operations per byte for a message of 47 bytes. By this rough measure, it is over 24 times the effort of FNV.

However, FNV is commonly used for short inputs, so doing a comparison of such inputs is relevant. Using the above comparison method, for inputs of N bytes, where N is <= 55 so SHA-1 will take one block, the ratio of the effort for SHA-1 to the effort for FNV will be 872/N. For inputs of N bytes, where N is <= 47 so SHA-256 will take one block, the ratio of the effort for SHA-256 to the effort for FNV will be 1029/N. Some examples are given below.

Example	Length in Bytes	SHA-1 Effort Relative to FNV Effort	SHA-256 Effort Relative to FNV Effort
IPv4 address	4	218	514
MAC address	6	145	171
IPv6 address	16	54	64

Table 3

Appendix B. Previous IETF FNV Code

FNV-1a was referenced in draft-ietf-tls-cached-info-08 (which was ultimately published as RFC 7924, but RFC 7924 no longer contains the code below). Herein, we provide the Java code for FNV64 from that earlier 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 getFNV1a64Digest (String inpString) {

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

       BigInteger digest = fnvOffsetBasis;

       for (int i = 0; i < inpString.length(); i++) {
           digest = digest.xor(BigInteger.valueOf(
                    (int) inpString.charAt(i)));
           digest = digest.multiply(fnvPrime).mod(m);
       }
       return (digest);

   }
}

<CODE ENDS>
```

Acknowledgements

The authors greatly appreciate the work of Glenn S. Fowler, who was part of the team that created the FNV algorithm.

The contributions of the following, listed in alphabetical order, are gratefully acknowledged:

Roman Donchenko, Frank Ellermann, Stephen Farrell, Tony Finch, Paul Hoffman, Charlie Kaufman, Eliot Lear, Bob Moskowitz, Gayle Noble, Stefan Santesson, Mukund Sivaraman, and Paul Wouters.

Authors' Addresses

Landon Curt Noll
Email: fnv-ietf8-mail@asthe.com
URI: http://www.isthe.com/chongo

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
200 Laurel Avenue South
Middletown, New Jersey 07748
United States of America
Email: tony@att.com