Intel Intelligent Storage Acceleration Library

2.31.0

Generated by Doxygen 1.9.1

Chapter 1

Intel(R) Intelligent Storage Acceleration Library

ISA-L is a collection of optimized low-level functions targeting storage applications. ISA-L includes:

- Erasure codes - Fast block Reed-Solomon type erasure codes for any encode/decode matrix in GF(2^8).
- CRC - Fast implementations of cyclic redundancy check. Six different polynomials supported.
  - iscsi32, ieee32, t10dif, ecma64, iso64, jones64, rocksoft64.
- Raid - calculate and operate on XOR and P+Q parity found in common RAID implementations.
- Compression - Fast deflate-compatible data compression.
- De-compression - Fast inflate-compatible data compression.
- igzip - A command line application like gzip, accelerated with ISA-L.

Also see:

- `ISA-L for updates`.
- For crypto functions see `isa-l_crypto on github`.
- The `github wiki` including a list of `distros/ports` offering binary packages as well as a list of `language bindings`.
- Contributing.
- Security Policy.
- Docs on units, tests, or build details.

1.1 Building ISA-L

1.1.1 Prerequisites

- Make: GNU 'make' or 'nmake' (Windows).
- Optional: Building with autotools requires autoconf/automake/libtool packages.
- Optional: Manual generation requires help2man package.

x86_64:

- Assembler: nasm. Version 2.15 or later suggested (other versions of nasm and yasm may build but with limited function support).
- Compiler: gcc, clang, icc or VC compiler.

aarch64:

- Assembler: gas v2.24 or later.
- Compiler: gcc v4.7 or later.

other:

- Compiler: Portable base functions are available that build with most C compilers.

1.1.2 Autotools

To build and install the library with autotools it is usually sufficient to run:

```
./autogen.sh
./configure
make
sudo make install
```

1.1.3 Makefile

To use a standard makefile run:

```
make -f Makefile.unx
```

1.1.4 Windows

On Windows use nmake to build dll and static lib:

```
nmake -f Makefile.nmake
```

or see details on setting up environment here.

1.1.5 Other make targets

Other targets include:

- `make check` : create and run tests
- `make tests` : create additional unit tests
- `make perfs` : create included performance tests
- `make ex` : build examples
- `make other` : build other utilities such as compression file tests
- `make doc` : build API manual

1.2 DLL Injection Attack

1.2.1 Problem

The Windows OS has an insecure predefined search order and set of defaults when trying to locate a resource. If the resource location is not specified by the software, an attacker need only place a malicious version in one of the locations Windows will search, and it will be loaded instead. Although this weakness can occur with any resource, it is especially common with DLL files.

1.2.2 Solutions

Applications using libisal DLL library may need to apply one of the solutions to prevent from DLL injection attack.

Two solutions are available:

- Using a Fully Qualified Path is the most secure way to load a DLL
- Signature verification of the DLL

1.2.3 Resources and Solution Details

- Security remarks section of LoadLibraryEx documentation by Microsoft: https://docs.microsoft.com/en-us/windows/win32/api/libloaderapi/nf-libloaderapi-loadlibraryexa#security-remark
- Microsoft Dynamic Link Library Security article: https://docs.microsoft.com/en-us/windows/win32/dlls/d
- Hijack Execution Flow: DLL Search Order Hijacking: https://attack.mitre.org/techniques/T1574/001
- Hijack Execution Flow: DLL Side-Loading: https://attack.mitre.org/techniques/T1574/002

Chapter 2

Contributing to ISA-L

Everyone is welcome to contribute. Patches may be submitted using GitHub pull requests (PRs). All commits must be signed off by the developer (–signoff) which indicates that you agree to the Developer Certificate of Origin. Patch discussion will happen directly on the GitHub PR. Design pre-work and general discussion occurs on the mailing list. Anyone can provide feedback in either location and all discussion is welcome. Decisions on whether to merge patches will be handled by the maintainer.

2.1 License

ISA-L is licensed using a BSD 3-clause [license]. All code submitted to the project is required to carry that license.

2.2 Certificate of Origin

In order to get a clear contribution chain of trust we use the signed-off-by language used by the Linux kernel project.

2.3 Mailing List

Contributors and users are welcome to submit new request on our roadmap, submit patches, file issues, and ask questions on our mailing list.

2.4 Coding Style

The coding style for ISA-L C code roughly follows linux kernel guidelines. Use the included indent script to format C code.

```
./tools/iindent your_files.c
```

And use check format script before submitting.

```
./tools/check_format.sh
```

Chapter 3

ISA-L Security Policy

3.1 Report a Vulnerability

Please report security issues or vulnerabilities to the Intel Security Center.

For more information on how Intel works to resolve security issues, see Vulnerability Handling Guidelines.

Chapter 4

v2.31 Intel Intelligent Storage Acceleration Library Release Notes

RELEASE NOTE CONTENTS

4.1 1. KNOWN ISSUES

- Perf tests do not run in Windows environment.
- 32-bit lib is not supported in Windows.
- 32-bit lib is not validated.

4.2 2. FIXED ISSUES

v2.31

- Fixed various compilation issues/warnings for different platforms.
- Fixed documentation on xor/pq gen/check functions, with minimum number of vectors.
- Fixed potential out-of-bounds read on Adler32 Neon implementation.
- Fixed potential out-of-bounds read on gf_vect_mul Neon implementation.
- Fixed x86 load/store instructions in erasure coding functions (aligned moves that should be unaligned).
- Fixed memory leaks in unit tests.

v2.30

- Intel CET support.
- Windows nasm support fix.

v2.28

- Fix documentation on gf_vect_mad(). Min length listed as 32 instead of required min 64 bytes.

v2.27

- Fix lack of install for pkg-config files

v2.26

- Fixes for sanitizer warnings.

v2.25

- Fix for nasm on Mac OS X/darwin.

v2.24

- Fix for crc32_iscsi(). Potential read-over for small buffer. For an input buffer length of less than 8 bytes and aligned to an 8 byte boundary, function could read past length. Previously had the possibility to cause a seg fault only for length 0 and invalid buffer passed. Calculated CRC is unchanged.
- Fix for compression/decompression of $>$ 4GB files. For streaming compression of extremely large files, the total_out parameter would wrap and could potentially flag an otherwise valid lookback distance as being invalid. Total_out is still 32bit for zlib compatibility. No inconsistent compressed buffers were generated by the issue.

v2.23

- Fix for histogram generation base function.
- Fix library build warnings on macOS.
- Fix igzip to use bsf instruction when tzcnt is not available.

v2.22

- Fix ISA-L builds for other architectures. Base function and examples sanitized for non-IA builds.
- Fix fuzz test script to work with llvm 6.0 builtin libFuzz.

v2.20

- Inflate total_out behavior corrected for in-progress decompression. Previously total_out represented the total bytes decompressed into the output buffer or temp internal buffer. This is changed to be only the bytes put into the output buffer.
- Fixed issue with isal_create_hufftables_subset. Affects semi-dynamic compression use case when explicitly creating hufftables from histogram. The _hufftables_subset function could fail to generate length symbols for any length that were never seen.

v2.19

- Fix erasure code test that violates rs matrix bounds.
- Fix 0 length file and looping errors in igzip_inflate_test.

v2.18

- Mac OS X/darwin systems no longer require the –target=darwin config option. The autoconf canonical build should detect.

v2.17

- Fix igzip using 32K window and a shared object
- Fix igzip undefined instruction error on Nehalem.
- Fixed issue in crc performance tests where OS optimizations turned cold cache tests into warm tests.

v2.15

- Fix for windows register save in gf_6vect_mad_avx2.asm. Only affects windows versions of ec_encode_data_update() running with AVX2. A GP register was not properly restored resulting in corruption on return.

v2.14

- Building in unit directories is no longer supported removing the issue of leftover object files causing the top-level make build to fail.

v2.10

- Fix for windows register save overlap in gf_{3-6}vect_dot_prod_sse.asm. Only affects windows versions of erasure code. GP register saves/restore were pushed to same stack area as XMM.

4.3 3. CHANGE LOG & FEATURES ADDED

v2.31

- API changes:
  - gf_vect_mul_base() function now returns an integer, matching the return type of gf_vect_mul() function (not a breaking change).
- Igzip compression improvements:
  - Added compress/decompress with dictionary to perf test app.
  - Zlib header can be now created on the fly when starting the compression.
  - Added isal_zlib_hdr_init() function to initialize the zlib header to 0.
- Zero-memory dectection improvements:
  - Optimized AVX implementation.
  - Added new AVX2 and AVX512 implementations.
- Erasure coding improvements:
  - Added new AVX512 and AVX2 implementations using GFNI instructions.
  - Added new SVE implementation.
- CRC improvements:
  - Added new CRC64 Rocksoft algorithm.
  - CRC x86 implementations optimized using ternary logic instructions and folding of bigger data on the last bytes.
  - CRC16 T10dif aarch64 implementation improved.
  - CRC aarch64 implementations optimized using XOR fusion feature.
- Documentation:
  - Added function overview documentation page.
  - Added security file.
- Performance apps:
  - Changed performance tests to warm by default.
- Example apps:
  - Added CRC combine example `crc_combine_example` for multiple polynomials.

v2.30

- Igzip compression enhancements.
  - New functions for dictionary acceleration. Split dictionary processing and resetting can greatly accelerate the performance of compressing many small files with a dictionary.
  - New static level 0 header decode tables. Accelerates decompressing small files that are level 0 compressed by skipping the known header parsing.

  - New feature for igzip cli tool: support for concatenated .gz files. On decompression, igzip will process a series of independent, concatenated .gz files into one output stream.

- CRC Improvements
  - New vclmul version of crc32_iscsi().
  - Updates for aarch64.

v2.29

- CRC Improvements
  - New AVX512 vclmul versions of crc16_t10dif(), crc32_ieee(), crc32_gzip_refl.
- Erasure code improvements
  - Added AVX512 ec functions with 5 and 6 outputs. Can improve performance for codes with 5 or more parity by running in batches of up to 6 at a time.

v2.28

- New next-arch versions of 64-bit CRC. All norm and reflected 64-bit polynomials are expanded to utilize vpclmulqdq.

v2.27

- New multi-threaded compression option for igzip cli tool

v2.26

- Adler32 added to external API.
- Multi-arch improvements.
- Performance test improvements.

v2.25

- Igzip performance improvements and features.
  - Performance improvements for uncompressable files. Random or uncompressable files can be up to 3x faster in level 1 or 2 compression.
  - Additional small file performance improvments.
  - New options in igzip cli: use name from header or not, test compressed file.
- Multi-arch autoconf script.
  - Autoconf should detect architecture and run base functions at minimum.

v2.24

- Igzip small file performance improvements and new features.
  - Better performance on small files.
  - New gzip/zlib header and trailer handling.
  - New gzip/zlib header parsing helper functions.
  - New user-space compression/decompression tool igzip.
- New mem unit added with first function isal_zero_detect().

v2.23

- Igzip inflate (decompression) performance improvements.
  - Implemented multi-byte decode for inflate. Decode can pack up to three symbols into the decode table making some compressed streams decompress much faster depending on the prevalence of short codes.

v2.22

- Igzip: AVX2 version of level 3 compression added.
- Erasure code examples
  - New examples for standard EC encode and decode.
  - Example of piggyback EC encode and decode.

v2.21

- Igzip improvements
  - New compression levels added. ISA-L fast deflate now has more levels to balance speed vs. target compression level. Level 0, 1 are as in previous generations. New levels 2 & 3 target higher compression roughly comparable to zlib levels 2-3. Level 3 is currently only optimized for processors with AVX512 instructions.
- New T10dif & copy function - crc16_t10dif_copy()
  - CRC and copy was added to emulate T10dif operations such as DIF insert and strip. This function stitches together CRC and memcpy operations eliminating an extra data read.
- CRC32 iscsi performance improvements
  - Fixes issue under some distributions where warm cache performance was reduced.

v2.20

- Igzip improvements
  - Optimized deflate_hash in compression functions. Improves performance of using preset dictionary.
  - Removed alignment restrictions on input structure.

v2.19

- Igzip improvements
  - Add optimized Adler-32 checksum.
  - Implement zlib compression format.
  - Add stateful dictionary support.
  - Add struct reset functions for both deflate and inflate.
- Reflected IEEE format CRC32 is released out. Function interface is named crc32_gzip_refl.
- Exact work condition of Erasure Code Reed-Solomon Matrix is determined by new added program gen_rs_↩ matrix_limits.

v2.18

- New 2-pass fully-dynamic deflate compression (level -1). ISA-L fast deflate now has two levels. Level 0 (default) is the same as previous generations. Setting to level 1 will switch to the fully-dynamic compression that will typically reach higher compression ratios.
- RAID AVX512 functions.

v2.17

- New fast decompression (inflate)
- Compression improvements (deflate)
  - Speed and compression ratio improvements.
  - Fast custom Huffman code generation.
  - New features:
    - Run-time option of gzip crc calculation and headers/trailer.
    - Choice of static header (BTYPE 01) blocks.
    - LARGE_WINDOW, 32K history, now default.
    - Stateless full flush mode.
- CRC64
  - Six new 64-bit polynomials supported. Normal and reflected versions of ECMA, ISO and Jones polynomials.

v2.16

- Units added: crc, raid, igzip (deflate compression).

v2.15

- Erasure code updates. New AVX512 versions.
- Nasm support. ISA-L ported to build with nasm or yasm assembler.

- Windows DLL support. Windows builds DLL by default.

v2.14

- Autoconf and autotools build allows easier porting to additional systems. Previous make system still available to embedded users with Makefile.unx.
- Includes update for building on Mac OS X/darwin systems. Add –target=darwin to ./configure step.

v2.13

- Erasure code improvments
  - 32-bit port of optimized gf_vect_dot_prod() functions. This makes ec_encode_data() functions much faster on 32-bit processors.
  - Avoton performance improvements. Performance on Avoton for gf_vect_dot_prod() and ec_encode_data() can improve by as much as 20%.

v2.11

- Incremental erasure code. New functions added to erasure code to handle single source update of code blocks. The function ec_encode_data_update() works with parameters similar to ec_encode_data() but are called incrementally with each source block. These versions are useful when source blocks are not all available at once.

v2.10

- Erasure code updates
  - New AVX and AVX2 support functions.
  - Changes min len requirement on gf_vect_dot_prod() to 32 from 16.
  - Tests include both source and parity recovery with ec_encode_data().
  - New encoding examples with Vandermonde or Cauchy matrix.

v2.8

- First open release of erasure code unit that is part of ISA-L.

Chapter 5

ISA-L Function Overview

ISA-L is logically broken into mostly independent units based on the source directories of the same name.

- erasure_codes
- crc
- raid
- mem
- igzip

The library can also be built with subsets of available units. For example `$ make -f Makefile.unx units=crc` will only build a library with crc functions.

5.1 ISA-L Functions

5.1.1 Erasure Code Functions

Functions pertaining to erasure codes implement a general Reed-Solomon type encoding for blocks of data to protect against erasure of whole blocks. Individual operations can be described in terms of arithmetic in the Galois finite field GF(2^8) with the particular field-defining primitive or reducing polynomial $x^8 + x^4 + x^3 + x^2 + 1$ (0x1d).

For example, the function ec_encode_data() will generate a set of parity blocks P_i from the set of k source blocks D_i and arbitrary encoding coefficients $a_{i,j}$ where each byte in P is calculated from sources as:

$$P_i = \sum_{j=1}^{k} a_{i,j} \cdot D_j$$

where addition and multiplication $\cdot$ is defined in GF(2^8). Since any arbitrary set of coefficients $a_{i,j}$ can be supplied, the same fundamental function can be used for encoding blocks or decoding from blocks in erasure.

5.1.1.1 EC Usage

Various examples are available in examples/ec and unit tests in `erasure_code` to show an encode and decode (re-hydrate) cycle or partial update operation. As seen in `ec example` the process starts with picking an encode matrix, parameters k (source blocks) and m (total parity + source blocks), and expanding the necessary coefficients.

```
// Initialize g_tbls from encode matrix
ec_init_tables(k, p, &encode_matrix[k * k], g_tbls);
```

In the example, a symmetric encode matrix is used where only the coefficients describing the parity blocks are used for encode and the upper matrix is initialized as an identity to simplify generation of the corresponding decode matrix. Next the parity for all (m - k) blocks are calculated at once.

```
// Generate EC parity blocks from sources
ec_encode_data(len, k, p, g_tbls, frag_ptrs, &frag_ptrs[k]);
```

5.1.2 RAID Functions

Functions in the RAID section calculate and operate on XOR and P+Q parity found in common RAID implementations. The mathematics of RAID are based on Galois finite-field arithmetic to find one or two parity bytes for each byte in N sources such that single or dual disk failures (one or two erasures) can be corrected. For RAID5, a block of parity is calculated by the xor across the N source arrays. Each parity byte is calculated from N sources by:

$$P = D_0 + D_1 + ... + D_{N-1}$$

where D_n are elements across each source array [0-(N-1)] and + is the bit-wise exclusive or (xor) operation. Elements in GF(2^8) are implemented as bytes.

For RAID6, two parity bytes P and Q are calculated from the source array. P is calculated as in RAID5 and Q is calculated using the generator g as:

$$Q = g^0 D_0 + g^1 D_1 + g^2 D_2 + ... + g^{N-1} D_{N-1}$$

where g is chosen as {2}, the second field element. Multiplication and the field are defined using the primitive polynomial $x^8 + x^4 + x^3 + x^2 + 1$ (0x1d).

5.1.2.1 RAID Usage

RAID function usage is similar to erasure code except no coefficient expansion step is necessary. As seen in `raid example` the xor_gen() and xor_check() functions are used to generate and check parity.

5.1.3 CRC Functions

Functions in the CRC section include fast implementations of cyclic redundancy check using specialized instructions such as PCLMULQDQ, carry-less multiplication. Generally, a CRC is the remainder in binary division of a message and a CRC polynomial in GF(2).

$$CRC(M(x)) = x^{deg(P(x))} \cdot M(x) \, mod \, P(x)$$

CRC is used in many storage applications to ensure integrity of data by appending the CRC to a message. Various standards choose the polynomial P and may vary by initial seeding value, bit reversal and inverting the result and seed.

5.1.3.1 CRC Usage

CRC functions have a simple interface such as in `crc example`.

```
crc64_checksum = crc64_ecma_refl(crc64_checksum, inbuf, avail_in);
```

Updates with new buffers are possible with subsequent calls. No extra finalize step is necessary. An example of combining independent CRC values is found in `crc combine example`.

5.1.4 Compress/Inflate Functions

Functions in the igzip unit perform fast, loss-less data compression and decompression within the `deflate`, `zlib`, and `gzip` binary standards. Functions for stream based (data pieces at a time) and stateless (data all at once) are available as well as multiple parameters to change the speed vs. compression ratio or other features. In addition, there are functions to fine tune compression by pre-computing static Huffman tables and setting for subsequent compression runs, parsing compression headers and other specific tasks to give more control.

5.1.4.1 Compress/Inflate Usage

The interface for compression and decompression functions is similar to zlib, zstd and others where a context structure keeps parameters and internal state to render from an input buffer to an output buffer. I/O buffer pointers and size are often the only required settings. ISA-L, unlike zlib and others, does not allocate new memory and must be done by the user explicitly when required (level 1 and above). This gives the user more flexibility to when dynamic memory is allocated and reused. The minimum code for starting a compression is just allocating a stream structure and initializing it. This can be done just once for multiple compression runs.

```
struct isal_zstream stream;
isal_deflate_init(&stream);
```

Using level 1 compression and above requires an additional, initial allocation for an internal intermediate buffer. Suggested sizes are defined in external headers.

```
stream.level = 1;
stream.level_buf = malloc(ISAL_DEF_LVL1_DEFAULT);
stream.level_buf_size = ISAL_DEF_LVL1_DEFAULT;
```

After init, subsequent, multiple compression runs can be performed by supplying (or re-using) I/O buffers.

```
stream.next_in = inbuf;
stream->next_out = outbuf;
stream->avail_in = inbuf_size;
stream->avail_out = outbuf_size;
isal_deflate(stream);
```

See `igzip example` for a simple example program or review the perf or check tests for more.

igzip: ISA-L also provides a user program *igzip* to compress and decompress files. Optionally igzip can be compiled with multi-threaded compression. See `man igzip` for details.

5.2 General Library Features

5.2.1 Multi-Binary Dispatchers

Multibinary support is available for all units in ISA-L. With multibinary support functions, an appropriate version is selected at first run and can be called instead of architecture-specific versions. This allows users to deploy a single binary with multiple function versions and choose at run time based on platform features. All functions also have base functions, written in portable C, which the multibinary function will call if none of the required instruction sets are enabled.

5.2.2 Threading

All ISA-L library functions are single threaded but reentrant and thread-safe making it easy for users to incorporate with any threading library. The igzip command line utility has threaded compression but not built by default. To add to an automake build do `$ make D="-DHAVE_THREADS"`.

5.2.3 Included Tests and Utilities

ISA-L source `repo` includes unit tests, performance tests and other utilities.

Examples:

- `ec example`
- `raid example`
- `crc example`
- `igzip example`

Chapter 6

ISA-L Testing

Tests are divided into check tests, unit tests and fuzz tests. Check tests, built with `make check`, should have no additional dependencies. Other unit tests built with `make test` may have additional dependencies in order to make comparisons of the output of ISA-L to other standard libraries and ensure compatibility. Fuzz tests are meant to be run with a fuzzing tool such as `AFL` or `llvm libFuzzer` fuzzing to direct the input data based on coverage. There are a number of scripts in the /tools directory to help with automating the running of tests.

6.1 Test check

`./tools/test_autorun.sh` is a helper script for kicking off check tests, that typically run for a few minutes, or extended tests that could run much longer. The command `test_autorun.sh check` build and runs all check tests with autotools and runs other short tests to ensure check tests, unit tests, examples, install, exe stack, format are correct. Each run of `test_autorun.sh` builds tests with a new random test seed that ensures that each run is unique to the seed but deterministic for debugging. Tests are also built with sanitizers and Electric Fence if available.

6.2 Extended tests

Extended tests are initiated with the command `./tools/test_autorun.sh ext`. These build and run check tests, unit tests, and other utilities that can take much longer than check tests alone. This includes special compression tools and some cross targets such as the no-arch build of base functions only and mingw build if tools are available.

6.3 Fuzz testing

`./tools/test_fuzz.sh` is a helper script for fuzzing to setup, build and run the ISA-L inflate fuzz tests on multiple fuzz tools. Fuzzing with `llvm libFuzzer` requires clang compiler tools with `-fsanitize=fuzzer` or `libFuzzer` installed. You can invoke the default fuzz tests under llvm with

```
./tools/test_fuzz.sh -e checked
```

To use `AFL`, install tools and system setup for `afl-fuzz` and run

```
./tools/test_fuzz.sh -e checked --afl 1 --llvm -1 -d 1
```

This uses internal vectors as a seed. You can also specify a sample file to use as a seed instead with `-f <file>`. One of three fuzz tests can be invoked: checked, simple, and round_trip.

Chapter 7

ISA-L Build Details

7.1 Build tools

NASM: For x86-64 builds it is highly recommended to get an up-to-date version of `nasm` that can understand the latest instruction sets. Building with an older assembler version is often possible but the library may lack some function versions for the best performance. For example, as a minimum, nasm v2.11.01 or yasm 1.2.0 can be used to build a limited functionality library but it will not include any function versions with AVX2, AVX512, or optimizations for many processors before the assembler's build. The configure or make tools can run tests to check the assembler's knowledge of new instructions and change build defines. For autoconf builds, check the output of configure for full nasm support as it includes the following lines.

```
checking for nasm... yes
checking for modern nasm... yes
checking for optional nasm AVX512 support... yes
checking for additional nasm AVX512 support... yes
```

If an appropriate nasm is not available from your distro, it is simple to build from source or download an executable from `nasm`.

```
git clone --depth=10 https://github.com/netwide-assembler/nasm
cd nasm
./autogen.sh
./configure
make
sudo make install
```

7.2 Windows Build Environment Details

The windows dynamic and static libraries can be built with the nmake tool on the windows command line when appropriate paths and tools are setup as follows.

7.2.1 Download nasm and put into path

Download and install `nasm` and add location to path.

```
set PATH=%PATH%;C:\Program Files\NASM
```

7.2.2 Setup compiler environment

Install compiler and run environment setup script.
Compilers for windows usually have a batch file to setup environment variables for the command line called `vcvarsall.bat` or `compilervars.bat` or a link to run these. For Visual Studio this may be as follows for Community edition.

```
C:\Program Files (x86)\Microsoft Visual Studio\2019\Community\VC\Auxiliary\Build\vcvarsall.bat x64
```

For the Intel compiler the path is typically as follows where yyyy, x, zzz represent the version.

```
C:\Program Files (x86)\IntelSWTools\system_studio_for_windows_yyyy.x.zzz\compilers_and_libraries_yyyy\bin\compile
```

7.2.3 Build ISA-L libs and copy to appropriate place

Run `nmake /f Makefile.nmake`
This should build isa-l.dll, isa-l.lib and isa-l_static.lib. You may want to copy the libs to a system directory in the dynamic linking path such as `C:\windows\system32` or to a project directory.
To build a simple program with a static library.

```
cl /Fe: test.exe test.c isa-l_static.lib
```

Chapter 8

Instruction Set Requirements for arch-specific functions (non-multibinary)

Global crc64_ecma_norm_by8 (uint64_t init_crc, const unsigned char *buf, uint64_t len)
SSE3, CLMUL

Global crc64_ecma_refl_by8 (uint64_t init_crc, const unsigned char *buf, uint64_t len)
SSE3, CLMUL

Global crc64_iso_norm_by8 (uint64_t init_crc, const unsigned char *buf, uint64_t len)
SSE3, CLMUL

Global crc64_iso_refl_by8 (uint64_t init_crc, const unsigned char *buf, uint64_t len)
SSE3, CLMUL

Global crc64_jones_norm_by8 (uint64_t init_crc, const unsigned char *buf, uint64_t len)
SSE3, CLMUL

Global crc64_jones_refl_by8 (uint64_t init_crc, const unsigned char *buf, uint64_t len)
SSE3, CLMUL

Global crc64_rocksoft_norm_by8 (uint64_t init_crc, const unsigned char *buf, uint64_t len)
SSE3, CLMUL

Global crc64_rocksoft_refl_by8 (uint64_t init_crc, const unsigned char *buf, uint64_t len)
SSE3, CLMUL

Chapter 9

Data Structure Index

9.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 10

File Index

10.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 11

Data Structure Documentation

11.1 BitBuf2 Struct Reference

Holds Bit Buffer information.
`#include <igzip_lib.h>`

Data Fields

- uint64_t m_bits
  bits in the bit buffer
- uint32_t m_bit_count
  number of valid bits in the bit buffer
- uint8_t * m_out_buf
  current index of buffer to write to
- uint8_t * m_out_end
  end of buffer to write to
- uint8_t * m_out_start
  start of buffer to write to

11.1.1 Detailed Description

Holds Bit Buffer information.
The documentation for this struct was generated from the following file:

- igzip_lib.h

11.2 inflate_huff_code_large Struct Reference

Large lookup table for decoding huffman codes.
`#include <igzip_lib.h>`

Data Fields

- uint32_t short_code_lookup [1<<(ISAL_DECODE_LONG_BITS)]
  Short code lookup table.
- uint16_t long_code_lookup [ISAL_HUFF_CODE_LARGE_LONG_ALIGNED]
  Long code lookup table.

11.2.1 Detailed Description

Large lookup table for decoding huffman codes.
The documentation for this struct was generated from the following file:

- igzip_lib.h

11.3 inflate_huff_code_small Struct Reference

Small lookup table for decoding huffman codes.
`#include <igzip_lib.h>`

Data Fields

- uint16_t short_code_lookup [1<<(ISAL_DECODE_SHORT_BITS)]

  Short code lookup table.
- uint16_t long_code_lookup [ISAL_HUFF_CODE_SMALL_LONG_ALIGNED]

  Long code lookup table.

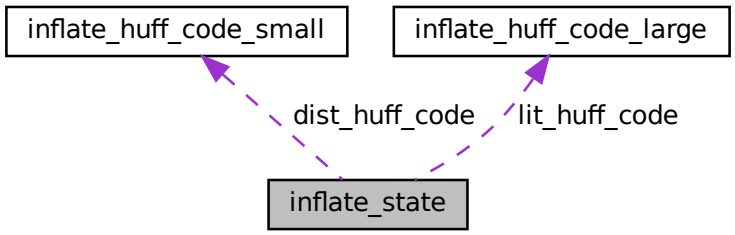
11.3.1 Detailed Description

Small lookup table for decoding huffman codes.
The documentation for this struct was generated from the following file:

- igzip_lib.h

11.4 inflate_state Struct Reference

Holds decompression state information.
`#include <igzip_lib.h>`
Collaboration diagram for inflate_state:



Data Fields

- uint8_t * next_out

  Next output Byte.
- uint32_t avail_out

Number of bytes available at next_out.

- uint32_t total_out

  Total bytes written out so far.
- uint8_t * next_in

  Next input byte.
- uint64_t read_in

  Bits buffered to handle unaligned streams.
- uint32_t avail_in

  Number of bytes available at next_in.
- int32_t read_in_length

  Bits in read_in.
- struct inflate_huff_code_large lit_huff_code

  Structure for decoding lit/len symbols.
- struct inflate_huff_code_small dist_huff_code

  Structure for decoding dist symbols.
- enum isal_block_state block_state

  Current decompression state.
- uint32_t dict_length

  Length of dictionary used.
- uint32_t bfinal

  Flag identifying final block.
- uint32_t crc_flag

  Flag identifying whether to track of crc.
- uint32_t crc

  Contains crc or adler32 of output if crc_flag is set.
- uint32_t hist_bits

  Log base 2 of maximum lookback distance.
- int32_t copy_overflow_length

  Length left to copy when outbuffer overflow occurred.
- int32_t copy_overflow_distance

  Lookback distance when outbuffer overflow occurred.
- int16_t tmp_in_size

  Number of bytes in tmp_in_buffer.
- int32_t tmp_out_valid

  Number of bytes in tmp_out_buffer.
- int32_t tmp_out_processed

  Number of bytes processed in tmp_out_buffer.
- uint8_t tmp_in_buffer [ISAL_DEF_MAX_HDR_SIZE]

  Temporary buffer containing data from the input stream.
- uint8_t tmp_out_buffer [2 *ISAL_DEF_HIST_SIZE+ISAL_LOOK_AHEAD]

  Temporary buffer containing data from the output stream.
- int32_t type0_block_len

  Length left to read of type 0 block when outbuffer overflow occurred.
- int32_t count

  Count of bytes remaining to be parsed.

11.4.1 Detailed Description

Holds decompression state information.
The documentation for this struct was generated from the following file:

- igzip_lib.h

11.5 isal_dict Struct Reference

Structure for holding processed dictionary information.
`#include <igzip_lib.h>`

11.5.1 Detailed Description

Structure for holding processed dictionary information.
The documentation for this struct was generated from the following file:

- igzip_lib.h

11.6 isal_gzip_header Struct Reference

Holds Gzip header information.
`#include <igzip_lib.h>`

Data Fields

- uint32_t text
  Optional Text hint.
- uint32_t time
  Unix modification time in gzip header.
- uint32_t xflags
  xflags in gzip header
- uint32_t os
  OS in gzip header.
- uint8_t * extra
  Extra field in gzip header.
- uint32_t extra_buf_len
  Length of extra buffer.
- uint32_t extra_len
  Actual length of gzip header extra field.
- char * name
  Name in gzip header.
- uint32_t name_buf_len
  Length of name buffer.
- char * comment
  Comments in gzip header.
- uint32_t comment_buf_len
  Length of comment buffer.
- uint32_t hcrc
  Header crc or header crc flag.
- uint32_t flags
  Internal data.

11.6.1 Detailed Description

Holds Gzip header information.
The documentation for this struct was generated from the following file:

- igzip_lib.h

11.7 isal_huff_histogram Struct Reference

Holds histogram of deflate symbols.
`#include <igzip_lib.h>`

Data Fields

- uint64_t lit_len_histogram [ISAL_DEF_LIT_LEN_SYMBOLS]

  Histogram of Literal/Len symbols seen.
- uint64_t dist_histogram [ISAL_DEF_DIST_SYMBOLS]

  Histogram of Distance Symbols seen.
- uint16_t hash_table [IGZIP_LVL0_HASH_SIZE]

  Tmp space used as a hash table.

11.7.1 Detailed Description

Holds histogram of deflate symbols.
The documentation for this struct was generated from the following file:

- igzip_lib.h

11.8 isal_hufftables Struct Reference

Holds the huffman tree used to huffman encode the input stream.
`#include <igzip_lib.h>`

Data Fields

- uint8_t deflate_hdr [ISAL_DEF_MAX_HDR_SIZE]

  deflate huffman tree header
- uint32_t deflate_hdr_count

  Number of whole bytes in deflate_huff_hdr.
- uint32_t deflate_hdr_extra_bits

  Number of bits in the partial byte in header.
- uint32_t dist_table [IGZIP_DIST_TABLE_SIZE]

  bits 4:0 are the code length, bits 31:5 are the code
- uint32_t len_table [IGZIP_LEN_TABLE_SIZE]

  bits 4:0 are the code length, bits 31:5 are the code
- uint16_t lit_table [IGZIP_LIT_TABLE_SIZE]

  literal code
- uint8_t lit_table_sizes [IGZIP_LIT_TABLE_SIZE]

  literal code length
- uint16_t dcodes [30 - IGZIP_DECODE_OFFSET]

distance code

- uint8_t dcodes_sizes [30 - IGZIP_DECODE_OFFSET]

  distance code length

11.8.1 Detailed Description

Holds the huffman tree used to huffman encode the input stream.
The documentation for this struct was generated from the following file:

- igzip_lib.h

11.9 isal_mod_hist Struct Reference

Holds modified histogram.
`#include <igzip_lib.h>`

Data Fields

- uint32_t d_hist [30]

  Distance.

11.9.1 Detailed Description

Holds modified histogram.
The documentation for this struct was generated from the following file:

- igzip_lib.h

11.10 isal_zlib_header Struct Reference

Holds Zlib header information.
`#include <igzip_lib.h>`

Data Fields

- uint32_t info

  base-2 logarithm of the LZ77 window size minus 8

- uint32_t level

  Compression level (fastest, fast, default, maximum)

- uint32_t dict_id

  Dictionary id.

- uint32_t dict_flag

  Whether to use a dictionary.

11.10.1 Detailed Description

Holds Zlib header information.
The documentation for this struct was generated from the following file:

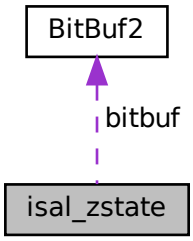
- igzip_lib.h

11.11 isal_zstate Struct Reference

Holds the internal state information for input and output compression streams.

`#include <igzip_lib.h>`

Collaboration diagram for isal_zstate:



Data Fields

- uint32_t total_in_start

  Not used, may be replaced with something else.
- uint32_t block_next

  Start of current deflate block in the input.
- uint32_t block_end

  End of current deflate block in the input.
- uint32_t dist_mask

  Distance mask used.
- enum isal_zstate_state state

  Current state in processing the data stream.
- struct BitBuf2 bitbuf

  Bit Buffer.
- uint32_t crc

  Current checksum without finalize step if any (adler)
- uint8_t has_wrap_hdr

  keeps track of wrapper header
- uint8_t has_eob_hdr

  keeps track of eob hdr (with BFINAL set)
- uint8_t has_eob

  keeps track of eob on the last deflate block
- uint8_t has_hist

  flag to track if there is match history
- uint16_t has_level_buf_init

  flag to track if user supplied memory has been initialized.
- uint32_t count

  used for partial header/trailer writes

- uint8_t tmp_out_buff [16]

  temporary array
- uint32_t tmp_out_start

  temporary variable
- uint32_t tmp_out_end

  temporary variable
- uint32_t b_bytes_valid

  number of valid bytes in buffer
- uint32_t b_bytes_processed

  number of bytes processed in buffer
- uint8_t buffer [2 ∗IGZIP_HIST_SIZE+ISAL_LOOK_AHEAD]

  Internal buffer.
- uint16_t head [IGZIP_LVL0_HASH_SIZE]

  Hash array.

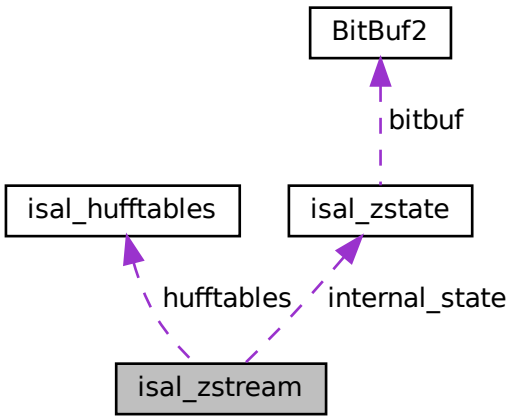
11.11.1 Detailed Description

Holds the internal state information for input and output compression streams.
The documentation for this struct was generated from the following file:

- igzip_lib.h

11.12 isal_zstream Struct Reference

Holds stream information.
`#include <igzip_lib.h>`
Collaboration diagram for isal_zstream:



Data Fields

- uint8_t * next_in

  Next input byte.
- uint32_t avail_in

  number of bytes available at next_in
- uint32_t total_in

  total number of bytes read so far
- uint8_t * next_out

  Next output byte.
- uint32_t avail_out

  number of bytes available at next_out
- uint32_t total_out

  total number of bytes written so far
- struct isal_hufftables * hufftables

  Huffman encoding used when compressing.
- uint32_t level

  Compression level to use.
- uint32_t level_buf_size

  Size of level_buf.
- uint8_t * level_buf

  User allocated buffer required for different compression levels.
- uint16_t end_of_stream

  non-zero if this is the last input buffer
- uint16_t flush

  Flush type can be NO_FLUSH, SYNC_FLUSH or FULL_FLUSH.
- uint16_t gzip_flag

  Indicate if gzip compression is to be performed.
- uint16_t hist_bits

  Log base 2 of maximum lookback distance, 0 is use default.
- struct isal_zstate internal_state

  Internal state for this stream.

11.12.1 Detailed Description

Holds stream information.
The documentation for this struct was generated from the following file:

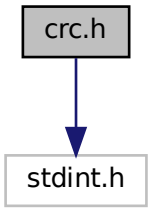
- igzip_lib.h

Chapter 12

File Documentation

12.1 crc.h File Reference

CRC functions.
`#include <stdint.h>`
Include dependency graph for crc.h:



Functions

- uint16_t crc16_t10dif (uint16_t init_crc, const unsigned char *buf, uint64_t len)

  Generate CRC from the T10 standard, runs appropriate version.
- uint16_t crc16_t10dif_copy (uint16_t init_crc, uint8_t *dst, uint8_t *src, uint64_t len)

  Generate CRC and copy T10 standard, runs appropriate version.
- uint32_t crc32_ieee (uint32_t init_crc, const unsigned char *buf, uint64_t len)

  Generate CRC from the IEEE standard, runs appropriate version.
- uint32_t crc32_gzip_refl (uint32_t init_crc, const unsigned char *buf, uint64_t len)

  Generate the customized CRC based on RFC 1952 CRC (http://www.ietf.org/rfc/rfc1952.txt) standard, runs appropriate version.
- unsigned int crc32_iscsi (unsigned char *buffer, int len, unsigned int init_crc)

  ISCSI CRC function, runs appropriate version.
- unsigned int crc32_iscsi_base (unsigned char *buffer, int len, unsigned int crc_init)

  ISCSI CRC function, baseline version.
- uint16_t crc16_t10dif_base (uint16_t seed, uint8_t *buf, uint64_t len)

Generate CRC from the T10 standard, runs baseline version.

- uint16_t crc16_t10dif_copy_base (uint16_t init_crc, uint8_t *dst, uint8_t *src, uint64_t len)

  Generate CRC and copy T10 standard, runs baseline version.

- uint32_t crc32_ieee_base (uint32_t seed, uint8_t *buf, uint64_t len)

  Generate CRC from the IEEE standard, runs baseline version.

- uint32_t crc32_gzip_refl_base (uint32_t seed, uint8_t *buf, uint64_t len)

  Generate the customized CRC based on RFC 1952 CRC (http://www.ietf.org/rfc/rfc1952.txt) standard, runs baseline version.

12.1.1 Detailed Description

CRC functions.

12.1.2 Function Documentation

12.1.2.1 crc16_t10dif()

```
uint16_t crc16_t10dif (
            uint16_t init_crc,
            const unsigned char * buf,
            uint64_t len )
```

Generate CRC from the T10 standard, runs appropriate version.
This function determines what instruction sets are enabled and selects the appropriate version at runtime.

Returns

16 bit CRC

Parameters

init_crc	initial CRC value, 16 bits
buf	buffer to calculate CRC on
len	buffer length in bytes (64-bit data)

12.1.2.2 crc16_t10dif_base()

```
uint16_t crc16_t10dif_base (
            uint16_t seed,
            uint8_t * buf,
            uint64_t len )
```

Generate CRC from the T10 standard, runs baseline version.

Returns

16 bit CRC

Parameters

seed	initial CRC value, 16 bits
buf	buffer to calculate CRC on
len	buffer length in bytes (64-bit data)

12.1.2.3 crc16_t10dif_copy()

```
uint16_t crc16_t10dif_copy (
            uint16_t init_crc,
            uint8_t * dst,
            uint8_t * src,
            uint64_t len )
```

Generate CRC and copy T10 standard, runs appropriate version.
Stitched CRC + copy function.

Returns

16 bit CRC

Parameters

init_crc	initial CRC value, 16 bits
dst	buffer destination for copy
src	buffer source to crc + copy
len	buffer length in bytes (64-bit data)

12.1.2.4 crc16_t10dif_copy_base()

```
uint16_t crc16_t10dif_copy_base (
            uint16_t init_crc,
            uint8_t * dst,
            uint8_t * src,
            uint64_t len )
```

Generate CRC and copy T10 standard, runs baseline version.

Returns

16 bit CRC

Parameters

init_crc	initial CRC value, 16 bits
dst	buffer destination for copy
src	buffer source to crc + copy
len	buffer length in bytes (64-bit data)

12.1.2.5 crc32_gzip_refl()

```
uint32_t crc32_gzip_refl (
            uint32_t init_crc,
            const unsigned char * buf,
            uint64_t len )
```

Generate the customized CRC based on RFC 1952 CRC (http://www.ietf.org/rfc/rfc1952.txt) standard, runs appropriate version.
This function determines what instruction sets are enabled and selects the appropriate version at runtime.
Note: CRC32 IEEE standard is widely used in HDLC, Ethernet, Gzip and many others. Its polynomial is 0x04C11DB7 in normal and 0xEDB88320 in reflection (or reverse). In ISA-L CRC, function crc32_ieee is actually designed for normal CRC32 IEEE version. And function crc32_gzip_refl is actually designed for reflected CRC32 IEEE. These two versions of CRC32 IEEE are not compatible with each other. Users who want to replace their not optimized crc32 ieee with ISA-L's crc32 function should be careful of that. Since many applications use CRC32 IEEE reflected version, Please have a check whether crc32_gzip_refl is right one for you instead of crc32_ieee.

Returns

32 bit CRC

Parameters

init_crc	initial CRC value, 32 bits
buf	buffer to calculate CRC on
len	buffer length in bytes (64-bit data)

12.1.2.6 crc32_gzip_refl_base()

```
uint32_t crc32_gzip_refl_base (
            uint32_t seed,
            uint8_t * buf,
            uint64_t len )
```

Generate the customized CRC based on RFC 1952 CRC (http://www.ietf.org/rfc/rfc1952.txt) standard, runs baseline version.

Returns

32 bit CRC

Parameters

seed	initial CRC value, 32 bits
buf	buffer to calculate CRC on
len	buffer length in bytes (64-bit data)

12.1.2.7 crc32_ieee()

```
uint32_t crc32_ieee (
            uint32_t init_crc,
            const unsigned char * buf,
            uint64_t len )
```

Generate CRC from the IEEE standard, runs appropriate version.
This function determines what instruction sets are enabled and selects the appropriate version at runtime. Note: CRC32 IEEE standard is widely used in HDLC, Ethernet, Gzip and many others. Its polynomial is 0x04C11DB7 in normal and 0xEDB88320 in reflection (or reverse). In ISA-L CRC, function crc32_ieee is actually designed for normal CRC32 IEEE version. And function crc32_gzip_refl is actually designed for reflected CRC32 IEEE. These two versions of CRC32

IEEE are not compatible with each other. Users who want to replace their not optimized crc32 ieee with ISA-L's crc32 function should be careful of that. Since many applications use CRC32 IEEE reflected version, Please have a check whether crc32_gzip_refl is right one for you instead of crc32_ieee.

Returns

32 bit CRC

Parameters

init_crc	initial CRC value, 32 bits
buf	buffer to calculate CRC on
len	buffer length in bytes (64-bit data)

12.1.2.8 crc32_ieee_base()

```
uint32_t crc32_ieee_base (
            uint32_t seed,
            uint8_t * buf,
            uint64_t len )
```

Generate CRC from the IEEE standard, runs baseline version.

Returns

32 bit CRC

Parameters

seed	initial CRC value, 32 bits
buf	buffer to calculate CRC on
len	buffer length in bytes (64-bit data)

12.1.2.9 crc32_iscsi()

```
unsigned int crc32_iscsi (
            unsigned char * buffer,
            int len,
            unsigned int init_crc )
```

ISCSI CRC function, runs appropriate version.
This function determines what instruction sets are enabled and selects the appropriate version at runtime.

Returns

32 bit CRC

Parameters

buffer	buffer to calculate CRC on
len	buffer length in bytes
init_crc	initial CRC value

12.1.2.10 crc32_iscsi_base()

```
unsigned int crc32_iscsi_base (
            unsigned char * buffer,
            int len,
            unsigned int crc_init )
```

ISCSI CRC function, baseline version.

Returns

32 bit CRC

Parameters

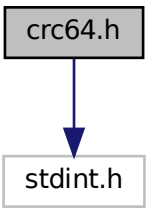
buffer	buffer to calculate CRC on
len	buffer length in bytes
crc_init	initial CRC value

12.2 crc64.h File Reference

CRC64 functions.

```
#include <stdint.h>
```

Include dependency graph for crc64.h:



Functions

- uint64_t crc64_ecma_refl (uint64_t init_crc, const unsigned char ∗buf, uint64_t len)

  Generate CRC from ECMA-182 standard in reflected format, runs appropriate version.
- uint64_t crc64_ecma_norm (uint64_t init_crc, const unsigned char ∗buf, uint64_t len)

  Generate CRC from ECMA-182 standard in normal format, runs appropriate version.
- uint64_t crc64_iso_refl (uint64_t init_crc, const unsigned char ∗buf, uint64_t len)

  Generate CRC from ISO standard in reflected format, runs appropriate version.
- uint64_t crc64_iso_norm (uint64_t init_crc, const unsigned char ∗buf, uint64_t len)

  Generate CRC from ISO standard in normal format, runs appropriate version.

- uint64_t crc64_jones_refl (uint64_t init_crc, const unsigned char *buf, uint64_t len)

  Generate CRC from "Jones" coefficients in reflected format, runs appropriate version.
- uint64_t crc64_jones_norm (uint64_t init_crc, const unsigned char *buf, uint64_t len)

  Generate CRC from "Jones" coefficients in normal format, runs appropriate version.
- uint64_t crc64_rocksoft_refl (uint64_t init_crc, const unsigned char *buf, uint64_t len)

  Generate CRC from "Rocksoft" coefficients in reflected format, runs appropriate version.
- uint64_t crc64_rocksoft_norm (uint64_t init_crc, const unsigned char *buf, uint64_t len)

  Generate CRC from "Rocksoft" coefficients in normal format, runs appropriate version.
- uint64_t crc64_ecma_refl_by8 (uint64_t init_crc, const unsigned char *buf, uint64_t len)

  Generate CRC from ECMA-182 standard in reflected format.
- uint64_t crc64_ecma_norm_by8 (uint64_t init_crc, const unsigned char *buf, uint64_t len)

  Generate CRC from ECMA-182 standard in normal format.
- uint64_t crc64_ecma_refl_base (uint64_t init_crc, const unsigned char *buf, uint64_t len)

  Generate CRC from ECMA-182 standard in reflected format, runs baseline version.
- uint64_t crc64_ecma_norm_base (uint64_t init_crc, const unsigned char *buf, uint64_t len)

  Generate CRC from ECMA-182 standard in normal format, runs baseline version.
- uint64_t crc64_iso_refl_by8 (uint64_t init_crc, const unsigned char *buf, uint64_t len)

  Generate CRC from ISO standard in reflected format.
- uint64_t crc64_iso_norm_by8 (uint64_t init_crc, const unsigned char *buf, uint64_t len)

  Generate CRC from ISO standard in normal format.
- uint64_t crc64_iso_refl_base (uint64_t init_crc, const unsigned char *buf, uint64_t len)

  Generate CRC from ISO standard in reflected format, runs baseline version.
- uint64_t crc64_iso_norm_base (uint64_t init_crc, const unsigned char *buf, uint64_t len)

  Generate CRC from ISO standard in normal format, runs baseline version.
- uint64_t crc64_jones_refl_by8 (uint64_t init_crc, const unsigned char *buf, uint64_t len)

  Generate CRC from "Jones" coefficients in reflected format.
- uint64_t crc64_jones_norm_by8 (uint64_t init_crc, const unsigned char *buf, uint64_t len)

  Generate CRC from "Jones" coefficients in normal format.
- uint64_t crc64_jones_refl_base (uint64_t init_crc, const unsigned char *buf, uint64_t len)

  Generate CRC from "Jones" coefficients in reflected format, runs baseline version.
- uint64_t crc64_jones_norm_base (uint64_t init_crc, const unsigned char *buf, uint64_t len)

  Generate CRC from "Jones" coefficients in normal format, runs baseline version.
- uint64_t crc64_rocksoft_refl_by8 (uint64_t init_crc, const unsigned char *buf, uint64_t len)

  Generate CRC from "Rocksoft" coefficients in reflected format.
- uint64_t crc64_rocksoft_refl_base (uint64_t init_crc, const unsigned char *buf, uint64_t len)

  Generate CRC from "Rocksoft" coefficients in reflected format, runs baseline version.
- uint64_t crc64_rocksoft_norm_by8 (uint64_t init_crc, const unsigned char *buf, uint64_t len)

  Generate CRC from "Rocksoft" coefficients in normal format.
- uint64_t crc64_rocksoft_norm_base (uint64_t init_crc, const unsigned char *buf, uint64_t len)

  Generate CRC from "Rocksoft" coefficients in normal format, runs baseline version.

12.2.1 Detailed Description

CRC64 functions.

12.2.2 Function Documentation

12.2.2.1 crc64_ecma_norm()

```
uint64_t crc64_ecma_norm (
            uint64_t init_crc,
            const unsigned char * buf,
            uint64_t len )
```

Generate CRC from ECMA-182 standard in normal format, runs appropriate version.
This function determines what instruction sets are enabled and selects the appropriate version at runtime.

Returns

64 bit CRC

Parameters

init_crc	initial CRC value, 64 bits
buf	buffer to calculate CRC on
len	buffer length in bytes (64-bit data)

12.2.2.2 crc64_ecma_norm_base()

```
uint64_t crc64_ecma_norm_base (
            uint64_t init_crc,
            const unsigned char * buf,
            uint64_t len )
```

Generate CRC from ECMA-182 standard in normal format, runs baseline version.

Returns

64 bit CRC

Parameters

init_crc	initial CRC value, 64 bits
buf	buffer to calculate CRC on
len	buffer length in bytes (64-bit data)

12.2.2.3 crc64_ecma_norm_by8()

```
uint64_t crc64_ecma_norm_by8 (
            uint64_t init_crc,
            const unsigned char * buf,
            uint64_t len )
```

Generate CRC from ECMA-182 standard in normal format.

Requires SSE3, CLMUL

Returns

64 bit CRC

Parameters

init_crc	initial CRC value, 64 bits
buf	buffer to calculate CRC on
len	buffer length in bytes (64-bit data)

12.2.2.4 crc64_ecma_refl()

```
uint64_t crc64_ecma_refl (
            uint64_t init_crc,
            const unsigned char * buf,
            uint64_t len )
```

Generate CRC from ECMA-182 standard in reflected format, runs appropriate version.
This function determines what instruction sets are enabled and selects the appropriate version at runtime.

Returns

64 bit CRC

Parameters

init_crc	initial CRC value, 64 bits
buf	buffer to calculate CRC on
len	buffer length in bytes (64-bit data)

12.2.2.5 crc64_ecma_refl_base()

```
uint64_t crc64_ecma_refl_base (
            uint64_t init_crc,
            const unsigned char * buf,
            uint64_t len )
```

Generate CRC from ECMA-182 standard in reflected format, runs baseline version.

Returns

64 bit CRC

Parameters

init_crc	initial CRC value, 64 bits
buf	buffer to calculate CRC on
len	buffer length in bytes (64-bit data)

12.2.2.6 crc64_ecma_refl_by8()

```
uint64_t crc64_ecma_refl_by8 (
            uint64_t init_crc,
```

```
            const unsigned char * buf,
            uint64_t len )
```

Generate CRC from ECMA-182 standard in reflected format.

Requires SSE3, CLMUL

Returns

64 bit CRC

Parameters

init_crc	initial CRC value, 64 bits
buf	buffer to calculate CRC on
len	buffer length in bytes (64-bit data)

12.2.2.7 crc64_iso_norm()

```
uint64_t crc64_iso_norm (
            uint64_t init_crc,
            const unsigned char * buf,
            uint64_t len )
```

Generate CRC from ISO standard in normal format, runs appropriate version.
This function determines what instruction sets are enabled and selects the appropriate version at runtime.

Returns

64 bit CRC

Parameters

init_crc	initial CRC value, 64 bits
buf	buffer to calculate CRC on
len	buffer length in bytes (64-bit data)

12.2.2.8 crc64_iso_norm_base()

```
uint64_t crc64_iso_norm_base (
            uint64_t init_crc,
            const unsigned char * buf,
            uint64_t len )
```

Generate CRC from ISO standard in normal format, runs baseline version.

Returns

64 bit CRC

Parameters

init_crc	initial CRC value, 64 bits

Parameters

buf	buffer to calculate CRC on
len	buffer length in bytes (64-bit data)

12.2.2.9 crc64_iso_norm_by8()

```
uint64_t crc64_iso_norm_by8 (
            uint64_t init_crc,
            const unsigned char * buf,
            uint64_t len )
```

Generate CRC from ISO standard in normal format.

Requires SSE3, CLMUL

Returns

64 bit CRC

Parameters

init_crc	initial CRC value, 64 bits
buf	buffer to calculate CRC on
len	buffer length in bytes (64-bit data)

12.2.2.10 crc64_iso_refl()

```
uint64_t crc64_iso_refl (
            uint64_t init_crc,
            const unsigned char * buf,
            uint64_t len )
```

Generate CRC from ISO standard in reflected format, runs appropriate version.
This function determines what instruction sets are enabled and selects the appropriate version at runtime.

Returns

64 bit CRC

Parameters

init_crc	initial CRC value, 64 bits
buf	buffer to calculate CRC on
len	buffer length in bytes (64-bit data)

12.2.2.11 crc64_iso_refl_base()

```
uint64_t crc64_iso_refl_base (
```

```
            uint64_t init_crc,
            const unsigned char * buf,
            uint64_t len )
```

Generate CRC from ISO standard in reflected format, runs baseline version.

Returns

64 bit CRC

Parameters

init_crc	initial CRC value, 64 bits
buf	buffer to calculate CRC on
len	buffer length in bytes (64-bit data)

12.2.2.12 crc64_iso_refl_by8()

```
uint64_t crc64_iso_refl_by8 (
            uint64_t init_crc,
            const unsigned char * buf,
            uint64_t len )
```

Generate CRC from ISO standard in reflected format.

Requires SSE3, CLMUL

Returns

64 bit CRC

Parameters

init_crc	initial CRC value, 64 bits
buf	buffer to calculate CRC on
len	buffer length in bytes (64-bit data)

12.2.2.13 crc64_jones_norm()

```
uint64_t crc64_jones_norm (
            uint64_t init_crc,
            const unsigned char * buf,
            uint64_t len )
```

Generate CRC from "Jones" coefficients in normal format, runs appropriate version.
This function determines what instruction sets are enabled and selects the appropriate version at runtime.

Returns

64 bit CRC

Parameters

init_crc	initial CRC value, 64 bits
buf	buffer to calculate CRC on
len	buffer length in bytes (64-bit data)

12.2.2.14 crc64_jones_norm_base()

```
uint64_t crc64_jones_norm_base (
            uint64_t init_crc,
            const unsigned char * buf,
            uint64_t len )
```

Generate CRC from "Jones" coefficients in normal format, runs baseline version.

Returns

64 bit CRC

Parameters

init_crc	initial CRC value, 64 bits
buf	buffer to calculate CRC on
len	buffer length in bytes (64-bit data)

12.2.2.15 crc64_jones_norm_by8()

```
uint64_t crc64_jones_norm_by8 (
            uint64_t init_crc,
            const unsigned char * buf,
            uint64_t len )
```

Generate CRC from "Jones" coefficients in normal format.

Requires SSE3, CLMUL

Returns

64 bit CRC

Parameters

init_crc	initial CRC value, 64 bits
buf	buffer to calculate CRC on
len	buffer length in bytes (64-bit data)

12.2.2.16 crc64_jones_refl()

```
uint64_t crc64_jones_refl (
```

```
            uint64_t init_crc,
            const unsigned char * buf,
            uint64_t len )
```

Generate CRC from "Jones" coefficients in reflected format, runs appropriate version.
This function determines what instruction sets are enabled and selects the appropriate version at runtime.

Returns

64 bit CRC

Parameters

init_crc	initial CRC value, 64 bits
buf	buffer to calculate CRC on
len	buffer length in bytes (64-bit data)

12.2.2.17 crc64_jones_refl_base()

```
uint64_t crc64_jones_refl_base (
            uint64_t init_crc,
            const unsigned char * buf,
            uint64_t len )
```

Generate CRC from "Jones" coefficients in reflected format, runs baseline version.

Returns

64 bit CRC

Parameters

init_crc	initial CRC value, 64 bits
buf	buffer to calculate CRC on
len	buffer length in bytes (64-bit data)

12.2.2.18 crc64_jones_refl_by8()

```
uint64_t crc64_jones_refl_by8 (
            uint64_t init_crc,
            const unsigned char * buf,
            uint64_t len )
```

Generate CRC from "Jones" coefficients in reflected format.

Requires SSE3, CLMUL

Returns

64 bit CRC

Parameters

init_crc	initial CRC value, 64 bits
buf	buffer to calculate CRC on
len	buffer length in bytes (64-bit data)

12.2.2.19 crc64_rocksoft_norm()

```
uint64_t crc64_rocksoft_norm (
            uint64_t init_crc,
            const unsigned char * buf,
            uint64_t len )
```

Generate CRC from "Rocksoft" coefficients in normal format, runs appropriate version.
This function determines what instruction sets are enabled and selects the appropriate version at runtime.

Returns

64 bit CRC

Parameters

init_crc	initial CRC value, 64 bits
buf	buffer to calculate CRC on
len	buffer length in bytes (64-bit data)

12.2.2.20 crc64_rocksoft_norm_base()

```
uint64_t crc64_rocksoft_norm_base (
            uint64_t init_crc,
            const unsigned char * buf,
            uint64_t len )
```

Generate CRC from "Rocksoft" coefficients in normal format, runs baseline version.

Returns

64 bit CRC

Parameters

init_crc	initial CRC value, 64 bits
buf	buffer to calculate CRC on
len	buffer length in bytes (64-bit data)

12.2.2.21 crc64_rocksoft_norm_by8()

```
uint64_t crc64_rocksoft_norm_by8 (
            uint64_t init_crc,
```

```
            const unsigned char * buf,
            uint64_t len )
```

Generate CRC from "Rocksoft" coefficients in normal format.

Requires SSE3, CLMUL

Returns

64 bit CRC

Parameters

init_crc	initial CRC value, 64 bits
buf	buffer to calculate CRC on
len	buffer length in bytes (64-bit data)

12.2.2.22 crc64_rocksoft_refl()

```
uint64_t crc64_rocksoft_refl (
            uint64_t init_crc,
            const unsigned char * buf,
            uint64_t len )
```

Generate CRC from "Rocksoft" coefficients in reflected format, runs appropriate version.
This function determines what instruction sets are enabled and selects the appropriate version at runtime.

Returns

64 bit CRC

Parameters

init_crc	initial CRC value, 64 bits
buf	buffer to calculate CRC on
len	buffer length in bytes (64-bit data)

12.2.2.23 crc64_rocksoft_refl_base()

```
uint64_t crc64_rocksoft_refl_base (
            uint64_t init_crc,
            const unsigned char * buf,
            uint64_t len )
```

Generate CRC from "Rocksoft" coefficients in reflected format, runs baseline version.

Returns

64 bit CRC

Parameters

init_crc	initial CRC value, 64 bits

Parameters

buf	buffer to calculate CRC on
len	buffer length in bytes (64-bit data)

12.2.2.24 crc64_rocksoft_refl_by8()

```
uint64_t crc64_rocksoft_refl_by8 (
            uint64_t init_crc,
            const unsigned char * buf,
            uint64_t len )
```

Generate CRC from "Rocksoft" coefficients in reflected format.

Requires SSE3, CLMUL

Returns

64 bit CRC

Parameters

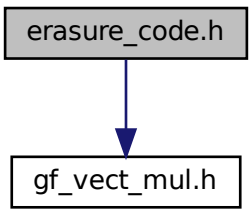
init_crc	initial CRC value, 64 bits
buf	buffer to calculate CRC on
len	buffer length in bytes (64-bit data)

12.3 erasure_code.h File Reference

Interface to functions supporting erasure code encode and decode.

```
#include "gf_vect_mul.h"
```

Include dependency graph for erasure_code.h:



Functions

- void ec_init_tables (int k, int rows, unsigned char ∗a, unsigned char ∗gftbls)

Initialize tables for fast Erasure Code encode and decode.

- void ec_init_tables_base (int k, int rows, unsigned char *a, unsigned char *gftbls)

  Initialize tables for fast Erasure Code encode and decode, runs baseline version.
- void ec_encode_data (int len, int k, int rows, unsigned char *gftbls, unsigned char **data, unsigned char **coding)

  Generate or decode erasure codes on blocks of data, runs appropriate version.
- void ec_encode_data_base (int len, int srcs, int dests, unsigned char *v, unsigned char **src, unsigned char **dest)

  Generate or decode erasure codes on blocks of data, runs baseline version.
- void ec_encode_data_update (int len, int k, int rows, int vec_i, unsigned char *g_tbls, unsigned char *data, unsigned char **coding)

  Generate update for encode or decode of erasure codes from single source, runs appropriate version.
- void ec_encode_data_update_base (int len, int k, int rows, int vec_i, unsigned char *v, unsigned char *data, unsigned char **dest)

  Generate update for encode or decode of erasure codes from single source.
- void gf_vect_dot_prod_base (int len, int vlen, unsigned char *gftbls, unsigned char **src, unsigned char *dest)

  GF(2^8) vector dot product, runs baseline version.
- void gf_vect_dot_prod (int len, int vlen, unsigned char *gftbls, unsigned char **src, unsigned char *dest)

  GF(2^8) vector dot product, runs appropriate version.
- void gf_vect_mad (int len, int vec, int vec_i, unsigned char *gftbls, unsigned char *src, unsigned char *dest)

  GF(2^8) vector multiply accumulate, runs appropriate version.
- void gf_vect_mad_base (int len, int vec, int vec_i, unsigned char *v, unsigned char *src, unsigned char *dest)

  GF(2^8) vector multiply accumulate, baseline version.
- unsigned char gf_mul (unsigned char a, unsigned char b)

  Single element GF(2^8) multiply.
- unsigned char gf_inv (unsigned char a)

  Single element GF(2^8) inverse.
- void gf_gen_rs_matrix (unsigned char *a, int m, int k)

  Generate a matrix of coefficients to be used for encoding.
- void gf_gen_cauchy1_matrix (unsigned char *a, int m, int k)

  Generate a Cauchy matrix of coefficients to be used for encoding.
- int gf_invert_matrix (unsigned char *in, unsigned char *out, const int n)

  Invert a matrix in GF(2^8)

12.3.1 Detailed Description

Interface to functions supporting erasure code encode and decode.
This file defines the interface to optimized functions used in erasure codes. Encode and decode of erasures in GF(2^8) are made by calculating the dot product of the symbols (bytes in GF(2^8)) across a set of buffers and a set of coefficients. Values for the coefficients are determined by the type of erasure code. Using a general dot product means that any sequence of coefficients may be used including erasure codes based on random coefficients. Multiple versions of dot product are supplied to calculate 1-6 output vectors in one pass. Base GF multiply and divide functions can be sped up by defining GF_LARGE_TABLES at the expense of memory size.

12.3.2 Function Documentation

12.3.2.1 ec_encode_data()

```
void ec_encode_data (
            int len,
            int k,
            int rows,
            unsigned char * gftbls,
            unsigned char ** data,
            unsigned char ** coding )
```

Generate or decode erasure codes on blocks of data, runs appropriate version.
Given a list of source data blocks, generate one or multiple blocks of encoded data as specified by a matrix of GF(2^8) coefficients. When given a suitable set of coefficients, this function will perform the fast generation or decoding of Reed-Solomon type erasure codes.
This function determines what instruction sets are enabled and selects the appropriate version at runtime.

Parameters

len	Length of each block of data (vector) of source or dest data.
k	The number of vector sources or rows in the generator matrix for coding.
rows	The number of output vectors to concurrently encode/decode.
gftbls	Pointer to array of input tables generated from coding coefficients in ec_init_tables(). Must be of size 32∗k∗rows
data	Array of pointers to source input buffers.
coding	Array of pointers to coded output buffers.

Returns

none

12.3.2.2 ec_encode_data_base()

```
void ec_encode_data_base (
            int len,
            int srcs,
            int dests,
            unsigned char * v,
            unsigned char ** src,
            unsigned char ** dest )
```

Generate or decode erasure codes on blocks of data, runs baseline version.
Baseline version of ec_encode_data() with same parameters.

12.3.2.3 ec_encode_data_update()

```
void ec_encode_data_update (
            int len,
            int k,
            int rows,
            int vec_i,
            unsigned char * g_tbls,
            unsigned char * data,
            unsigned char ** coding )
```

Generate update for encode or decode of erasure codes from single source, runs appropriate version.

Given one source data block, update one or multiple blocks of encoded data as specified by a matrix of GF(2^8) coefficients. When given a suitable set of coefficients, this function will perform the fast generation or decoding of Reed-Solomon type erasure codes from one input source at a time.
This function determines what instruction sets are enabled and selects the appropriate version at runtime.

Parameters

len	Length of each block of data (vector) of source or dest data.
k	The number of vector sources or rows in the generator matrix for coding.
rows	The number of output vectors to concurrently encode/decode.
vec_i	The vector index corresponding to the single input source.
g_tbls	Pointer to array of input tables generated from coding coefficients in ec_init_tables(). Must be of size 32∗k∗rows
data	Pointer to single input source used to update output parity.
coding	Array of pointers to coded output buffers.

Returns

none

12.3.2.4 ec_encode_data_update_base()

```
void ec_encode_data_update_base (
            int len,
            int k,
            int rows,
            int vec_i,
            unsigned char * v,
            unsigned char * data,
            unsigned char ** dest )
```

Generate update for encode or decode of erasure codes from single source.
Baseline version of ec_encode_data_update().

12.3.2.5 ec_init_tables()

```
void ec_init_tables (
            int k,
            int rows,
            unsigned char * a,
            unsigned char * gftbls )
```

Initialize tables for fast Erasure Code encode and decode.
Generates the expanded tables needed for fast encode or decode for erasure codes on blocks of data. 32bytes is generated for each input coefficient.

Parameters

k	The number of vector sources or rows in the generator matrix for coding.
rows	The number of output vectors to concurrently encode/decode.
a	Pointer to sets of arrays of input coefficients used to encode or decode data.
gftbls	Pointer to start of space for concatenated output tables generated from input coefficients. Must be of size 32∗k∗rows.

Returns

none

12.3.2.6 ec_init_tables_base()

```
void ec_init_tables_base (
            int k,
            int rows,
            unsigned char * a,
            unsigned char * gftbls )
```

Initialize tables for fast Erasure Code encode and decode, runs baseline version.
Baseline version of ec_encode_data() with same parameters.

12.3.2.7 gf_gen_cauchy1_matrix()

```
void gf_gen_cauchy1_matrix (
            unsigned char * a,
            int m,
            int k )
```

Generate a Cauchy matrix of coefficients to be used for encoding.
Cauchy matrix example of encoding coefficients where high portion of matrix is identity matrix I and lower portion is constructed as 1/(i + j) | i != j, i:{0,k-1} j:{k,m-1}. Any sub-matrix of a Cauchy matrix should be invertable.

Parameters

a	[m x k] array to hold coefficients
m	number of rows in matrix corresponding to srcs + parity.
k	number of columns in matrix corresponding to srcs.

Returns

none

12.3.2.8 gf_gen_rs_matrix()

```
void gf_gen_rs_matrix (
            unsigned char * a,
            int m,
            int k )
```

Generate a matrix of coefficients to be used for encoding.
Vandermonde matrix example of encoding coefficients where high portion of matrix is identity matrix I and lower portion is constructed as 2^{i*(j-k+1)} i:{0,k-1} j:{k,m-1}. Commonly used method for choosing coefficients in erasure encoding but does not guarantee invertable for every sub matrix. For large pairs of m and k it is possible to find cases where the decode matrix chosen from sources and parity is not invertable. Users may want to adjust for certain pairs m and k. If m and k satisfy one of the following inequalities, no adjustment is required:

- k <= 3
- k = 4, m <= 25
- k = 5, m <= 10

- k <= 21, m-k = 4
- m - k <= 3.

Parameters

a	[m x k] array to hold coefficients
m	number of rows in matrix corresponding to srcs + parity.
k	number of columns in matrix corresponding to srcs.

Returns

none

12.3.2.9 gf_inv()

```
unsigned char gf_inv (
            unsigned char a )
```

Single element GF(2^8) inverse.

Parameters

a	Input element

Returns

Field element b such that a x b = {1}

12.3.2.10 gf_invert_matrix()

```
int gf_invert_matrix (
            unsigned char * in,
            unsigned char * out,
            const int n )
```

Invert a matrix in GF(2^8)
Attempts to construct an n x n inverse of the input matrix. Returns non-zero if singular. Will always destroy input matrix in process.

Parameters

in	input matrix, destroyed by invert process
out	output matrix such that [in] x [out] = [I] - identity matrix
n	size of matrix [nxn]

Returns

0 successful, other fail on singular input matrix

12.3.2.11 gf_mul()

```
unsigned char gf_mul (
            unsigned char a,
            unsigned char b )
```

Single element GF(2^8) multiply.

Parameters

a	Multiplicand a
b	Multiplicand b

Returns

Product of a and b in GF(2^8)

12.3.2.12 gf_vect_dot_prod()

```
void gf_vect_dot_prod (
            int len,
            int vlen,
            unsigned char * gftbls,
            unsigned char ** src,
            unsigned char * dest )
```

GF(2^8) vector dot product, runs appropriate version.
Does a GF(2^8) dot product across each byte of the input array and a constant set of coefficients to produce each byte of the output. Can be used for erasure coding encode and decode. Function requires pre-calculation of a 32∗vlen byte constant array based on the input coefficients.
This function determines what instruction sets are enabled and selects the appropriate version at runtime.

Parameters

len	Length of each vector in bytes. Must be >= 32.
vlen	Number of vector sources.
gftbls	Pointer to 32∗vlen byte array of pre-calculated constants based on the array of input coefficients.
src	Array of pointers to source inputs.
dest	Pointer to destination data array.

Returns

none

12.3.2.13 gf_vect_dot_prod_base()

```
void gf_vect_dot_prod_base (
            int len,
            int vlen,
            unsigned char * gftbls,
            unsigned char ** src,
            unsigned char * dest )
```

GF(2^8) vector dot product, runs baseline version.
Does a GF(2^8) dot product across each byte of the input array and a constant set of coefficients to produce each byte of the output. Can be used for erasure coding encode and decode. Function requires pre-calculation of a 32∗vlen byte constant array based on the input coefficients.

Parameters

len	Length of each vector in bytes. Must be >= 16.
vlen	Number of vector sources.
gftbls	Pointer to 32∗vlen byte array of pre-calculated constants based on the array of input coefficients. Only elements 32∗CONST∗j + 1 of this array are used, where j = (0, 1, 2...) and CONST is the number of elements in the array of input coefficients. The elements used correspond to the original input coefficients.
src	Array of pointers to source inputs.
dest	Pointer to destination data array.

Returns

none

12.3.2.14 gf_vect_mad()

```
void gf_vect_mad (
             int len,
             int vec,
             int vec_i,
             unsigned char * gftbls,
             unsigned char * src,
             unsigned char * dest )
```

GF(2^8) vector multiply accumulate, runs appropriate version.
Does a GF(2^8) multiply across each byte of input source with expanded constant and add to destination array. Can be used for erasure coding encode and decode update when only one source is available at a time. Function requires pre-calculation of a 32∗vec byte constant array based on the input coefficients.
This function determines what instruction sets are enabled and selects the appropriate version at runtime.

Parameters

len	Length of each vector in bytes. Must be >= 64.
vec	The number of vector sources or rows in the generator matrix for coding.
vec_i	The vector index corresponding to the single input source.
gftbls	Pointer to array of input tables generated from coding coefficients in ec_init_tables(). Must be of size 32∗vec.
src	Array of pointers to source inputs.
dest	Pointer to destination data array.

Returns

none

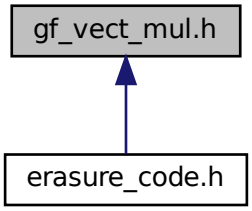
12.3.2.15 gf_vect_mad_base()

```
void gf_vect_mad_base (
            int len,
            int vec,
            int vec_i,
            unsigned char * v,
            unsigned char * src,
            unsigned char * dest )
```

GF(2^8) vector multiply accumulate, baseline version.
Baseline version of gf_vect_mad() with same parameters.

12.4 gf_vect_mul.h File Reference

Interface to functions for vector (block) multiplication in GF(2^8).
This graph shows which files directly or indirectly include this file:



Functions

- int gf_vect_mul (int len, unsigned char *gftbl, void *src, void *dest)
  GF(2^8) vector multiply by constant, runs appropriate version.
- void gf_vect_mul_init (unsigned char c, unsigned char *gftbl)
  Initialize 32-byte constant array for GF(2^8) vector multiply.
- int gf_vect_mul_base (int len, unsigned char *a, unsigned char *src, unsigned char *dest)
  GF(2^8) vector multiply by constant, runs baseline version.

12.4.1 Detailed Description

Interface to functions for vector (block) multiplication in GF(2^8).
This file defines the interface to routines used in fast RAID rebuild and erasure codes.

12.4.2 Function Documentation

12.4.2.1 gf_vect_mul()

```
int gf_vect_mul (
             int len,
             unsigned char * gftbl,
             void * src,
             void * dest )
```

GF(2^8) vector multiply by constant, runs appropriate version.
Does a GF(2^8) vector multiply b = Ca where a and b are arrays and C is a single field element in GF(2^8). Can be used for RAID6 rebuild and partial write functions. Function requires pre-calculation of a 32-element constant array based on constant C. gftbl(C) = {C{00}, C{01}, C{02}, ... , C{0f} }, {C{00}, C{10}, C{20}, ... , C{f0} }. Len and src must be aligned to 32B.
This function determines what instruction sets are enabled and selects the appropriate version at runtime.

Parameters

len	Length of vector in bytes. Must be aligned to 32B.
gftbl	Pointer to 32-byte array of pre-calculated constants based on C.
src	Pointer to src data array. Must be aligned to 32B.
dest	Pointer to destination data array. Must be aligned to 32B.

Returns

0 pass, other fail

12.4.2.2 gf_vect_mul_base()

```
int gf_vect_mul_base (
             int len,
             unsigned char * a,
             unsigned char * src,
             unsigned char * dest )
```

GF(2^8) vector multiply by constant, runs baseline version.
Does a GF(2^8) vector multiply b = Ca where a and b are arrays and C is a single field element in GF(2^8). Can be used for RAID6 rebuild and partial write functions. Function requires pre-calculation of a 32-element constant array based on constant C. gftbl(C) = {C{00}, C{01}, C{02}, ... , C{0f} }, {C{00}, C{10}, C{20}, ... , C{f0} }. Len and src must be aligned to 32B.

Parameters

len	Length of vector in bytes. Must be aligned to 32B.
a	Pointer to 32-byte array of pre-calculated constants based on C. only use 2nd element is used.
src	Pointer to src data array. Must be aligned to 32B.
dest	Pointer to destination data array. Must be aligned to 32B.

Returns

0 pass, other fail

12.4.2.3 gf_vect_mul_init()

```
void gf_vect_mul_init (
            unsigned char c,
            unsigned char * gftbl )
```

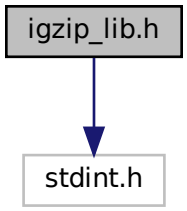
Initialize 32-byte constant array for GF(2^8) vector multiply.
Calculates array {C{00}, C{01}, C{02}, ... , C{0f} }, {C{00}, C{10}, C{20}, ... , C{f0} } as required by other fast vector multiply functions.

Parameters

c	Constant input.
gftbl	Table output.

12.5 igzip_lib.h File Reference

This file defines the igzip compression and decompression interface, a high performance deflate compression interface for storage applications.
`#include <stdint.h>`
Include dependency graph for igzip_lib.h:



Data Structures

- struct isal_huff_histogram

  Holds histogram of deflate symbols.
- struct isal_mod_hist

  Holds modified histogram.
- struct BitBuf2

  Holds Bit Buffer information.
- struct isal_zlib_header

  Holds Zlib header information.

- struct isal_gzip_header

  Holds Gzip header information.
- struct isal_zstate

  Holds the internal state information for input and output compression streams.
- struct isal_hufftables

  Holds the huffman tree used to huffman encode the input stream.
- struct isal_zstream

  Holds stream information.
- struct inflate_huff_code_large

  Large lookup table for decoding huffman codes.
- struct inflate_huff_code_small

  Small lookup table for decoding huffman codes.
- struct inflate_state

  Holds decompression state information.
- struct isal_dict

  Structure for holding processed dictionary information.

Enumerations

- enum isal_zstate_state {
  ZSTATE_NEW_HDR , ZSTATE_HDR , ZSTATE_CREATE_HDR , ZSTATE_BODY ,
  ZSTATE_FLUSH_READ_BUFFER , **ZSTATE_FLUSH_ICF_BUFFER** , **ZSTATE_TYPE0_HDR** , ZSTATE_TYPE0_BODY
  ,
  ZSTATE_SYNC_FLUSH , ZSTATE_FLUSH_WRITE_BUFFER , ZSTATE_TRL , ZSTATE_END ,
  ZSTATE_TMP_NEW_HDR , ZSTATE_TMP_HDR , ZSTATE_TMP_CREATE_HDR , ZSTATE_TMP_BODY ,
  ZSTATE_TMP_FLUSH_READ_BUFFER , **ZSTATE_TMP_FLUSH_ICF_BUFFER** , **ZSTATE_TMP_TYPE0_HDR**
  , ZSTATE_TMP_TYPE0_BODY ,
  ZSTATE_TMP_SYNC_FLUSH , ZSTATE_TMP_FLUSH_WRITE_BUFFER , ZSTATE_TMP_TRL , ZSTATE_TMP_END
  }

  Compression State please note ZSTATE_TRL only applies for GZIP compression.

Functions

- void isal_update_histogram (uint8_t *in_stream, int length, struct isal_huff_histogram *histogram)

  Updates histograms to include the symbols found in the input stream. Since this function only updates the histograms, it can be called on multiple streams to get a histogram better representing the desired data set. When first using histogram it must be initialized by zeroing the structure.
- int isal_create_hufftables (struct isal_hufftables *hufftables, struct isal_huff_histogram *histogram)

  Creates a custom huffman code for the given histograms in which every literal and repeat length is assigned a code and all possible lookback distances are assigned a code.
- int isal_create_hufftables_subset (struct isal_hufftables *hufftables, struct isal_huff_histogram *histogram)

  Creates a custom huffman code for the given histograms like isal_create_hufftables() except literals with 0 frequency in the histogram are not assigned a code.
- void isal_deflate_init (struct isal_zstream *stream)

  Initialize compression stream data structure.
- void isal_deflate_reset (struct isal_zstream *stream)

  Reinitialize compression stream data structure. Performs the same action as isal_deflate_init, but does not change user supplied input such as the level, flush type, compression wrapper (like gzip), hufftables, and end_of_stream_flag.
- void isal_gzip_header_init (struct isal_gzip_header *gz_hdr)

  Set gzip header default values.

- void isal_zlib_header_init (struct isal_zlib_header *z_hdr)

  Set zlib header default values.
- uint32_t isal_write_gzip_header (struct isal_zstream *stream, struct isal_gzip_header *gz_hdr)

  Write gzip header to output stream.
- uint32_t isal_write_zlib_header (struct isal_zstream *stream, struct isal_zlib_header *z_hdr)

  Write zlib header to output stream.
- int isal_deflate_set_hufftables (struct isal_zstream *stream, struct isal_hufftables *hufftables, int type)

  Set stream to use a new Huffman code.
- void isal_deflate_stateless_init (struct isal_zstream *stream)

  Initialize compression stream data structure.
- int isal_deflate_set_dict (struct isal_zstream *stream, uint8_t *dict, uint32_t dict_len)

  Set compression dictionary to use.
- int isal_deflate_process_dict (struct isal_zstream *stream, struct isal_dict *dict_str, uint8_t *dict, uint32_t dict_↩ len)

  Process dictionary to reuse later.
- int isal_deflate_reset_dict (struct isal_zstream *stream, struct isal_dict *dict_str)

  Reset compression dictionary to use.
- int isal_deflate (struct isal_zstream *stream)

  Fast data (deflate) compression for storage applications.
- int isal_deflate_stateless (struct isal_zstream *stream)

  Fast data (deflate) stateless compression for storage applications.
- void isal_inflate_init (struct inflate_state *state)

  Initialize decompression state data structure.
- void isal_inflate_reset (struct inflate_state *state)

  Reinitialize decompression state data structure.
- int isal_inflate_set_dict (struct inflate_state *state, uint8_t *dict, uint32_t dict_len)

  Set decompression dictionary to use.
- int isal_read_gzip_header (struct inflate_state *state, struct isal_gzip_header *gz_hdr)

  Read and return gzip header information.
- int isal_read_zlib_header (struct inflate_state *state, struct isal_zlib_header *zlib_hdr)

  Read and return zlib header information.
- int isal_inflate (struct inflate_state *state)

  Fast data (deflate) decompression for storage applications.
- int isal_inflate_stateless (struct inflate_state *state)

  Fast data (deflate) stateless decompression for storage applications.
- uint32_t isal_adler32 (uint32_t init, const unsigned char *buf, uint64_t len)

  Calculate Adler-32 checksum, runs appropriate version.

12.5.1 Detailed Description

This file defines the igzip compression and decompression interface, a high performance deflate compression interface for storage applications.
Deflate is a widely used compression standard that can be used standalone, it also forms the basis of gzip and zlib compression formats. Igzip supports the following flush features:

- No Flush: The default method where no special flush is performed.
- Sync flush: whereby isal_deflate() finishes the current deflate block at the end of each input buffer. The deflate block is byte aligned by appending an empty stored block.

- Full flush: whereby isal_deflate() finishes and aligns the deflate block as in sync flush but also ensures that subsequent block's history does not look back beyond this point and new blocks are fully independent.

Igzip also supports compression levels from ISAL_DEF_MIN_LEVEL to ISAL_DEF_MAX_LEVEL.
Igzip contains some behavior configurable at compile time. These configurable options are:

- IGZIP_HIST_SIZE - Defines the window size. The default value is 32K (note K represents 1024), but 8K is also supported. Powers of 2 which are at most 32K may also work.
- LONGER_HUFFTABLES - Defines whether to use a larger hufftables structure which may increase performance with smaller IGZIP_HIST_SIZE values. By default this option is not defined. This define sets IGZIP_HIST_SIZE to be 8 if IGZIP_HIST_SIZE $>$ 8K.

  As an example, to compile gzip with an 8K window size, in a terminal run

  ```
  gmake D="-D IGZIP_HIST_SIZE=8*1024"
  ```

  on Linux and FreeBSD, or with

  ```
  nmake -f Makefile.nmake D="-D
  *   IGZIP_HIST_SIZE=8*1024"
  ```

  on Windows.

12.5.2 Enumeration Type Documentation

12.5.2.1 isal_zstate_state

enum isal_zstate_state
Compression State please note ZSTATE_TRL only applies for GZIP compression.

Enumerator

ZSTATE_NEW_HDR	Header to be written.
ZSTATE_HDR	Header state.
ZSTATE_CREATE_HDR	Header to be created.
ZSTATE_BODY	Body state.
ZSTATE_FLUSH_READ_BUFFER	Flush buffer.
ZSTATE_TYPE0_BODY	Type0 block header to be written. Type0 block body to be written
ZSTATE_SYNC_FLUSH	Write sync flush block.
ZSTATE_FLUSH_WRITE_BUFFER	Flush bitbuf.
ZSTATE_TRL	Trailer state.
ZSTATE_END	End state.
ZSTATE_TMP_NEW_HDR	Temporary Header to be written.
ZSTATE_TMP_HDR	Temporary Header state.
ZSTATE_TMP_CREATE_HDR	Temporary Header to be created state.
ZSTATE_TMP_BODY	Temporary Body state.
ZSTATE_TMP_FLUSH_READ_BUFFER	Flush buffer.
ZSTATE_TMP_TYPE0_BODY	Temporary Type0 block header to be written. Temporary Type0 block body to be written
ZSTATE_TMP_SYNC_FLUSH	Write sync flush block.
ZSTATE_TMP_FLUSH_WRITE_BUFFER	Flush bitbuf.
ZSTATE_TMP_TRL	Temporary Trailer state.
ZSTATE_TMP_END	Temporary End state.

12.5.3 Function Documentation

12.5.3.1 isal_adler32()

```
uint32_t isal_adler32 (
            uint32_t init,
            const unsigned char * buf,
            uint64_t len )
```

Calculate Adler-32 checksum, runs appropriate version.
This function determines what instruction sets are enabled and selects the appropriate version at runtime.

Parameters

init	initial Adler-32 value
buf	buffer to calculate checksum on
len	buffer length in bytes

Returns

32-bit Adler-32 checksum

12.5.3.2 isal_create_hufftables()

```
int isal_create_hufftables (
            struct isal_hufftables * hufftables,
            struct isal_huff_histogram * histogram )
```

Creates a custom huffman code for the given histograms in which every literal and repeat length is assigned a code and all possible lookback distances are assigned a code.

Parameters

hufftables	the output structure containing the huffman code
histogram	histogram containing frequency of literal symbols, repeat lengths and lookback distances

Returns

Returns a non zero value if an invalid huffman code was created.

12.5.3.3 isal_create_hufftables_subset()

```
int isal_create_hufftables_subset (
            struct isal_hufftables * hufftables,
            struct isal_huff_histogram * histogram )
```

Creates a custom huffman code for the given histograms like isal_create_hufftables() except literals with 0 frequency in the histogram are not assigned a code.

Parameters

hufftables	the output structure containing the huffman code
histogram	histogram containing frequency of literal symbols, repeat lengths and lookback distances

Returns

Returns a non zero value if an invalid huffman code was created.

12.5.3.4 isal_deflate()

```
int isal_deflate (
            struct isal_zstream * stream )
```

Fast data (deflate) compression for storage applications.
The call to isal_deflate() will take data from the input buffer (updating next_in, avail_in and write a compressed stream to the output buffer (updating next_out and avail_out). The function returns when either the input buffer is empty or the output buffer is full.
On entry to isal_deflate(), next_in points to an input buffer and avail_in indicates the length of that buffer. Similarly next_out points to an empty output buffer and avail_out indicates the size of that buffer.
The fields total_in and total_out start at 0 and are updated by isal_deflate(). These reflect the total number of bytes read or written so far.
When the last input buffer is passed in, signaled by setting the end_of_stream, the routine will complete compression at the end of the input buffer, as long as the output buffer is big enough.
The compression level can be set by setting level to any value between ISAL_DEF_MIN_LEVEL and ISAL_DEF_↩ MAX_LEVEL. When the compression level is ISAL_DEF_MIN_LEVEL, hufftables can be set to a table trained for the the specific data type being compressed to achieve better compression. When a higher compression level is desired, a larger generic memory buffer needs to be supplied by setting level_buf and level_buf_size to represent the chunk of memory. For level x, the suggest size for this buffer this buffer is ISAL_DEFL_LVLx_DEFAULT. The defines ISAL_↩ DEFL_LVLx_MIN, ISAL_DEFL_LVLx_SMALL, ISAL_DEFL_LVLx_MEDIUM, ISAL_DEFL_LVLx_LARGE, and ISAL_↩ DEFL_LVLx_EXTRA_LARGE are also provided as other suggested sizes.
The equivalent of the zlib FLUSH_SYNC operation is currently supported. Flush types can be NO_FLUSH, SYNC_↩ FLUSH or FULL_FLUSH. Default flush type is NO_FLUSH. A SYNC_ OR FULL_ flush will byte align the deflate block by appending an empty stored block once all input has been compressed, including the buffered input. Checking that the out_buffer is not empty or that internal_state.state = ZSTATE_NEW_HDR is sufficient to guarantee all input has been flushed. Additionally FULL_FLUSH will ensure look back history does not include previous blocks so new blocks are fully independent. Switching between flush types is supported.
If a compression dictionary is required, the dictionary can be set calling isal_deflate_set_dictionary before calling isal↩ _deflate.
If the gzip_flag is set to IGZIP_GZIP, a generic gzip header and the gzip trailer are written around the deflate compressed data. If gzip_flag is set to IGZIP_GZIP_NO_HDR, then only the gzip trailer is written. A full-featured header is supported by the isal_write_{gzip,zlib}_header() functions.

Parameters

stream	Structure holding state information on the compression streams.

Returns

COMP_OK (if everything is ok), INVALID_FLUSH (if an invalid FLUSH is selected), ISAL_INVALID_LEVEL (if an invalid compression level is selected), ISAL_INVALID_LEVEL_BUF (if the level buffer is not large enough).

12.5.3.5 isal_deflate_init()

```
void isal_deflate_init (
            struct isal_zstream * stream )
```

Initialize compression stream data structure.

Parameters

stream	Structure holding state information on the compression streams.

Returns

none

12.5.3.6 isal_deflate_process_dict()

```
int isal_deflate_process_dict (
            struct isal_zstream * stream,
            struct isal_dict * dict_str,
            uint8_t * dict,
            uint32_t dict_len )
```

Process dictionary to reuse later.

Processes a dictionary so that the generated output can be reused to reset a new deflate stream more quickly than isal_deflate_set_dict() alone. This function is paired with isal_deflate_reset_dict() when using the same dictionary on multiple deflate objects. The stream.level must be set prior to calling this function to process the dictionary correctly. If the dictionary is longer than IGZIP_HIST_SIZE, only the last IGZIP_HIST_SIZE bytes will be used.

Parameters

stream	Structure holding state information on the compression streams.
dict_str	Structure to hold processed dictionary info to reuse later.
dict	Array containing dictionary to use.
dict_len	Length of dict.

Returns

COMP_OK, ISAL_INVALID_STATE (dictionary could not be processed)

12.5.3.7 isal_deflate_reset()

```
void isal_deflate_reset (
            struct isal_zstream * stream )
```

Reinitialize compression stream data structure. Performs the same action as isal_deflate_init, but does not change user supplied input such as the level, flush type, compression wrapper (like gzip), hufftables, and end_of_stream_flag.

Parameters

stream	Structure holding state information on the compression streams.

Returns

none

12.5.3.8 isal_deflate_reset_dict()

```
int isal_deflate_reset_dict (
            struct isal_zstream * stream,
            struct isal_dict * dict_str )
```

Reset compression dictionary to use.
Similar to isal_deflate_set_dict() but on pre-processed dictionary data. Pairing with isal_deflate_process_dict() can reduce the processing time on subsequent compression with dictionary especially on small files.
Like isal_deflate_set_dict(), this function is to be called after isal_deflate_init, or after completing a SYNC_FLUSH or FULL_FLUSH and before the next call do isal_deflate. Changing compression level between dictionary process and reset will cause return of ISAL_INVALID_STATE.

Parameters

stream	Structure holding state information on the compression streams.
dict_str	Structure with pre-processed dictionary info.

Returns

COMP_OK, ISAL_INVALID_STATE or other (dictionary could not be reset)

12.5.3.9 isal_deflate_set_dict()

```
int isal_deflate_set_dict (
            struct isal_zstream * stream,
            uint8_t * dict,
            uint32_t dict_len )
```

Set compression dictionary to use.
This function is to be called after isal_deflate_init, or after completing a SYNC_FLUSH or FULL_FLUSH and before the next call do isal_deflate. If the dictionary is longer than IGZIP_HIST_SIZE, only the last IGZIP_HIST_SIZE bytes will be used.

Parameters

stream	Structure holding state information on the compression streams.
dict	Array containing dictionary to use.
dict_len	Length of dict.

Returns

COMP_OK, ISAL_INVALID_STATE (dictionary could not be set)

12.5.3.10 isal_deflate_set_hufftables()

```
int isal_deflate_set_hufftables (
            struct isal_zstream * stream,
```

```
            struct isal_hufftables * hufftables,
            int type )
```

Set stream to use a new Huffman code.

Sets the Huffman code to be used in compression before compression start or after the successful completion of a SYNC_FLUSH or FULL_FLUSH. If type has value IGZIP_HUFFTABLE_DEFAULT, the stream is set to use the default Huffman code. If type has value IGZIP_HUFFTABLE_STATIC, the stream is set to use the deflate standard static Huffman code, or if type has value IGZIP_HUFFTABLE_CUSTOM, the stream is set to sue the isal_hufftables structure input to isal_deflate_set_hufftables.

Parameters

stream	Structure holding state information on the compression stream.
hufftables	new huffman code to use if type is set to IGZIP_HUFFTABLE_CUSTOM.
type	Flag specifying what hufftable to use.

Returns

Returns INVALID_OPERATION if the stream was unmodified. This may be due to the stream being in a state where changing the huffman code is not allowed or an invalid input is provided.

12.5.3.11 isal_deflate_stateless()

```
int isal_deflate_stateless (
            struct isal_zstream * stream )
```

Fast data (deflate) stateless compression for storage applications.

Stateless (one shot) compression routine with a similar interface to isal_deflate() but operates on entire input buffer at one time. Parameter avail_out must be large enough to fit the entire compressed output. Max expansion is limited to the input size plus the header size of a stored/raw block.

When the compression level is set to 1, unlike in isal_deflate(), level_buf may be optionally set depending on what what performance is desired.

For stateless the flush types NO_FLUSH and FULL_FLUSH are supported. FULL_FLUSH will byte align the output deflate block so additional blocks can be easily appended.

If the gzip_flag is set to IGZIP_GZIP, a generic gzip header and the gzip trailer are written around the deflate compressed data. If gzip_flag is set to IGZIP_GZIP_NO_HDR, then only the gzip trailer is written.

Parameters

stream	Structure holding state information on the compression streams.

Returns

COMP_OK (if everything is ok), INVALID_FLUSH (if an invalid FLUSH is selected), ISAL_INVALID_LEVEL (if an invalid compression level is selected), ISAL_INVALID_LEVEL_BUF (if the level buffer is not large enough), STATELESS_OVERFLOW (if output buffer will not fit output).

12.5.3.12 isal_deflate_stateless_init()

```
void isal_deflate_stateless_init (
            struct isal_zstream * stream )
```

Initialize compression stream data structure.

Parameters

stream	Structure holding state information on the compression streams.

Returns

none

12.5.3.13 isal_gzip_header_init()

```
void isal_gzip_header_init (
            struct isal_gzip_header * gz_hdr )
```

Set gzip header default values.

Parameters

gz_hdr	Gzip header to initialize.

12.5.3.14 isal_inflate()

```
int isal_inflate (
            struct inflate_state * state )
```

Fast data (deflate) decompression for storage applications.
On entry to isal_inflate(), next_in points to an input buffer and avail_in indicates the length of that buffer. Similarly next_out points to an empty output buffer and avail_out indicates the size of that buffer.
The field total_out starts at 0 and is updated by isal_inflate(). This reflects the total number of bytes written so far.
The call to isal_inflate() will take data from the input buffer (updating next_in, avail_in and write a decompressed stream to the output buffer (updating next_out and avail_out). The function returns when the input buffer is empty, the output buffer is full, invalid data is found, or in the case of zlib formatted data if a dictionary is specified. The current state of the decompression on exit can be read from state->block-state.
If the crc_flag is set to ISAL_GZIP_NO_HDR the gzip crc of the output is stored in state->crc. Alternatively, if the crc_↩ flag is set to ISAL_ZLIB_NO_HDR the adler32 of the output is stored in state->crc (checksum may not be updated until decompression is complete). When the crc_flag is set to ISAL_GZIP_NO_HDR_VER or ISAL_ZLIB_NO_HDR_VER, the behavior is the same, except the checksum is verified with the checksum after immediately following the deflate data. If the crc_flag is set to ISAL_GZIP or ISAL_ZLIB, the gzip/zlib header is parsed, state->crc is set to the appropriate checksum, and the checksum is verified. If the crc_flag is set to ISAL_DEFLATE (default), then the data is treated as a raw deflate block.
The element state->hist_bits has values from 0 to 15, where values of 1 to 15 are the log base 2 size of the matching window and 0 is the default with maximum history size.
If a dictionary is required, a call to isal_inflate_set_dict will set the dictionary.

Parameters

state	Structure holding state information on the compression streams.

Returns

ISAL_DECOMP_OK (if everything is ok), ISAL_INVALID_BLOCK, ISAL_NEED_DICT, ISAL_INVALID_↩ SYMBOL, ISAL_INVALID_LOOKBACK, ISAL_INVALID_WRAPPER, ISAL_UNSUPPORTED_METHOD, ISAL_↩ INCORRECT_CHECKSUM.

12.5.3.15 isal_inflate_init()

```
void isal_inflate_init (
            struct inflate_state * state )
```

Initialize decompression state data structure.

Parameters

state	Structure holding state information on the compression streams.

Returns

none

12.5.3.16 isal_inflate_reset()

```
void isal_inflate_reset (
            struct inflate_state * state )
```

Reinitialize decompression state data structure.

Parameters

state	Structure holding state information on the compression streams.

Returns

none

12.5.3.17 isal_inflate_set_dict()

```
int isal_inflate_set_dict (
            struct inflate_state * state,
            uint8_t * dict,
            uint32_t dict_len )
```

Set decompression dictionary to use.

This function is to be called after isal_inflate_init. If the dictionary is longer than IGZIP_HIST_SIZE, only the last IGZIP↩ _HIST_SIZE bytes will be used.

Parameters

state	Structure holding state information on the decompression stream.
dict	Array containing dictionary to use.
dict_len	Length of dict.

Returns

COMP_OK, ISAL_INVALID_STATE (dictionary could not be set)

12.5.3.18 isal_inflate_stateless()

```
int isal_inflate_stateless (
            struct inflate_state * state )
```

Fast data (deflate) stateless decompression for storage applications.
Stateless (one shot) decompression routine with a similar interface to isal_inflate() but operates on entire input buffer at one time. Parameter avail_out must be large enough to fit the entire decompressed output. Dictionaries are not supported.

Parameters

state	Structure holding state information on the compression streams.

Returns

ISAL_DECOMP_OK (if everything is ok), ISAL_END_INPUT (if all input was decompressed), ISAL_NEED_↩ DICT, ISAL_OUT_OVERFLOW (if output buffer ran out of space), ISAL_INVALID_BLOCK, ISAL_INVALID_↩ SYMBOL, ISAL_INVALID_LOOKBACK, ISAL_INVALID_WRAPPER, ISAL_UNSUPPORTED_METHOD, ISAL↩ _INCORRECT_CHECKSUM.

12.5.3.19 isal_read_gzip_header()

```
int isal_read_gzip_header (
            struct inflate_state * state,
            struct isal_gzip_header * gz_hdr )
```

Read and return gzip header information.
On entry state must be initialized and next_in pointing to a gzip compressed buffer. The buffers gz_hdr->extra, gz_↩ hdr->name, gz_hdr->comments and the buffer lengths must be set to record the corresponding field, or set to NULL to disregard that gzip header information. If one of these buffers overflows, the user can reallocate a larger buffer and call this function again to continue reading the header information.

Parameters

state	Structure holding state information on the decompression stream.
gz_hdr	Structure to return data encoded in the gzip header

Returns

ISAL_DECOMP_OK (header was successfully parsed) ISAL_END_INPUT (all input was parsed), ISAL_NAME↩ _OVERFLOW (gz_hdr->name overflowed while parsing), ISAL_COMMENT_OVERFLOW (gz_hdr->comment overflowed while parsing), ISAL_EXTRA_OVERFLOW (gz_hdr->extra overflowed while parsing), ISAL_↩ INVALID_WRAPPER (invalid gzip header found), ISAL_UNSUPPORTED_METHOD (deflate is not the compression method), ISAL_INCORRECT_CHECKSUM (gzip header checksum was incorrect)

12.5.3.20 isal_read_zlib_header()

```
int isal_read_zlib_header (
            struct inflate_state * state,
            struct isal_zlib_header * zlib_hdr )
```

Read and return zlib header information.
On entry state must be initialized and next_in pointing to a zlib compressed buffer.

Parameters

state	Structure holding state information on the decompression stream.
zlib_hdr	Structure to return data encoded in the zlib header

Returns

ISAL_DECOMP_OK (header was successfully parsed), ISAL_END_INPUT (all input was parsed), ISAL_↩ UNSUPPORTED_METHOD (deflate is not the compression method), ISAL_INCORRECT_CHECKSUM (zlib header checksum was incorrect)

12.5.3.21 isal_update_histogram()

```
void isal_update_histogram (
            uint8_t * in_stream,
            int length,
            struct isal_huff_histogram * histogram )
```

Updates histograms to include the symbols found in the input stream. Since this function only updates the histograms, it can be called on multiple streams to get a histogram better representing the desired data set. When first using histogram it must be initialized by zeroing the structure.

Parameters

in_stream	Input stream of data.
length	The length of start_stream.
histogram	The returned histogram of lit/len/dist symbols.

12.5.3.22 isal_write_gzip_header()

```
uint32_t isal_write_gzip_header (
            struct isal_zstream * stream,
            struct isal_gzip_header * gz_hdr )
```

Write gzip header to output stream.

Writes the gzip header to the output stream. On entry this function assumes that the output buffer has been initialized, so stream->next_out, stream->avail_out and stream->total_out have been set. If the output buffer contains insufficient space, stream is not modified.

Parameters

stream	Structure holding state information on the compression stream.
gz_hdr	Structure holding the gzip header information to encode.

Returns

Returns 0 if the header is successfully written, otherwise returns the minimum size required to successfully write the gzip header to the output buffer.

12.5.3.23 isal_write_zlib_header()

```
uint32_t isal_write_zlib_header (
            struct isal_zstream * stream,
            struct isal_zlib_header * z_hdr )
```

Write zlib header to output stream.

Writes the zlib header to the output stream. On entry this function assumes that the output buffer has been initialized, so stream->next_out, stream->avail_out and stream->total_out have been set. If the output buffer contains insufficient space, stream is not modified.

Parameters

stream	Structure holding state information on the compression stream.
z_hdr	Structure holding the zlib header information to encode.

Returns

Returns 0 if the header is successfully written, otherwise returns the minimum size required to successfully write the zlib header to the output buffer.

12.5.3.24 isal_zlib_header_init()

```
void isal_zlib_header_init (
            struct isal_zlib_header * z_hdr )
```

Set zlib header default values.

Parameters

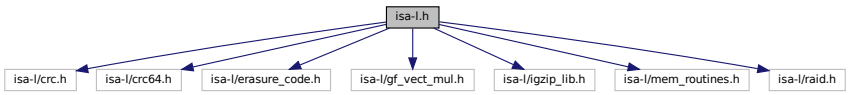
z_hdr	zlib header to initialize.

12.6 isa-l.h File Reference

Include for ISA-L library.

```
#include <isa-l/crc.h>
#include <isa-l/crc64.h>
#include <isa-l/erasure_code.h>
#include <isa-l/gf_vect_mul.h>
#include <isa-l/igzip_lib.h>
#include <isa-l/mem_routines.h>
#include <isa-l/raid.h>
```

Include dependency graph for isa-l.h:

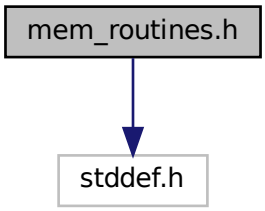


12.6.1 Detailed Description

Include for ISA-L library.

12.7 mem_routines.h File Reference

Interface to storage mem operations.
`#include <stddef.h>`
Include dependency graph for mem_routines.h:



Functions

- int isal_zero_detect (void *mem, size_t len)

  Detect if a memory region is all zero.

12.7.1 Detailed Description

Interface to storage mem operations.
Defines the interface for vector versions of common memory functions.

12.7.2 Function Documentation

12.7.2.1 isal_zero_detect()

```
int isal_zero_detect (
            void * mem,
            size_t len )
```

Detect if a memory region is all zero.
Zero detect function with optimizations for large blocks > 128 bytes

Parameters

mem	Pointer to memory region to test
len	Length of region in bytes

Returns

0 - region is all zeros other - region has non zero bytes

12.8 raid.h File Reference

Interface to RAID functions - XOR and P+Q calculation.

Functions

- int xor_gen (int vects, int len, void **array)

  Generate XOR parity vector from N sources, runs appropriate version.
- int xor_check (int vects, int len, void **array)

  Checks that array has XOR parity sum of 0 across all vectors, runs appropriate version.
- int pq_gen (int vects, int len, void **array)

  Generate P+Q parity vectors from N sources, runs appropriate version.
- int pq_check (int vects, int len, void **array)

  Checks that array of N sources, P and Q are consistent across all vectors, runs appropriate version.
- int pq_gen_base (int vects, int len, void **array)

  Generate P+Q parity vectors from N sources, runs baseline version.
- int xor_gen_base (int vects, int len, void **array)

  Generate XOR parity vector from N sources, runs baseline version.
- int xor_check_base (int vects, int len, void **array)

  Checks that array has XOR parity sum of 0 across all vectors, runs baseline version.
- int pq_check_base (int vects, int len, void **array)

  Checks that array of N sources, P and Q are consistent across all vectors, runs baseline version.

12.8.1 Detailed Description

Interface to RAID functions - XOR and P+Q calculation.
This file defines the interface to optimized XOR calculation (RAID5) or P+Q dual parity (RAID6). Operations are carried out on an array of pointers to sources and output arrays.

12.8.2 Function Documentation

12.8.2.1 pq_check()

```
int pq_check (
            int vects,
            int len,
            void ** array )
```

Checks that array of N sources, P and Q are consistent across all vectors, runs appropriate version.
This function determines what instruction sets are enabled and selects the appropriate version at runtime.

Parameters

vects	Number of vectors in array including P&Q. Must be > 3.
len	Length of each vector in bytes. Must be 16B aligned.
array	Array of pointers to source and P, Q. P and Q parity are assumed to be the last two pointers in the array. All pointers must be aligned to 16B.

Returns

0 pass, other fail

12.8.2.2 pq_check_base()

```
int pq_check_base (
            int vects,
            int len,
            void ** array )
```

Checks that array of N sources, P and Q are consistent across all vectors, runs baseline version.

Parameters

vects	Number of vectors in array including P&Q. Must be > 3.
len	Length of each vector in bytes. Must be 16B aligned.
array	Array of pointers to source and P, Q. P and Q parity are assumed to be the last two pointers in the array. All pointers must be aligned to 16B.

Returns

0 pass, other fail

12.8.2.3 pq_gen()

```
int pq_gen (
            int vects,
            int len,
            void ** array )
```

Generate P+Q parity vectors from N sources, runs appropriate version.
This function determines what instruction sets are enabled and selects the appropriate version at runtime.

Parameters

vects	Number of source+dest vectors in array. Must be > 3.
len	Length of each vector in bytes. Must be 32B aligned.
array	Array of pointers to source and dest. For P+Q the dest is the last two pointers. ie array[vects-2], array[vects-1]. P and Q parity vectors are written to these last two pointers. Src and dest pointers must be aligned to 32B.

Returns

0 pass, other fail

12.8.2.4 pq_gen_base()

```
int pq_gen_base (
            int vects,
```

```
            int len,
            void ** array )
```

Generate P+Q parity vectors from N sources, runs baseline version.

Parameters

vects	Number of source+dest vectors in array. Must be $>$ 3.
len	Length of each vector in bytes. Must be 16B aligned.
array	Array of pointers to source and dest. For P+Q the dest is the last two pointers. ie array[vects-2], array[vects-1]. P and Q parity vectors are written to these last two pointers. Src and dest pointers must be aligned to 16B.

Returns

0 pass, other fail

12.8.2.5 xor_check()

```
int xor_check (
            int vects,
            int len,
            void ** array )
```

Checks that array has XOR parity sum of 0 across all vectors, runs appropriate version.
This function determines what instruction sets are enabled and selects the appropriate version at runtime.

Parameters

vects	Number of vectors in array. Must be $>$ 1.
len	Length of each vector in bytes.
array	Array of pointers to vectors. Src and dest pointers must be aligned to 16B.

Returns

0 pass, other fail

12.8.2.6 xor_check_base()

```
int xor_check_base (
            int vects,
            int len,
            void ** array )
```

Checks that array has XOR parity sum of 0 across all vectors, runs baseline version.

Parameters

vects	Number of vectors in array. Must be $>$ 1.
len	Length of each vector in bytes.
array	Array of pointers to vectors. Src and dest pointers must be aligned to 16B.

Returns

0 pass, other fail

12.8.2.7 xor_gen()

```
int xor_gen (
            int vects,
            int len,
            void ** array )
```

Generate XOR parity vector from N sources, runs appropriate version.
This function determines what instruction sets are enabled and selects the appropriate version at runtime.

Parameters

vects	Number of source+dest vectors in array. Must be $>$ 2.
len	Length of each vector in bytes.
array	Array of pointers to source and dest. For XOR the dest is the last pointer. ie array[vects-1]. Src and dest pointers must be aligned to 32B.

Returns

0 pass, other fail

12.8.2.8 xor_gen_base()

```
int xor_gen_base (
            int vects,
            int len,
            void ** array )
```

Generate XOR parity vector from N sources, runs baseline version.

Parameters

vects	Number of source+dest vectors in array. Must be $>$ 2.
len	Length of each vector in bytes.
array	Array of pointers to source and dest. For XOR the dest is the last pointer. ie array[vects-1]. Src and dest pointers must be aligned to 32B.

Returns

0 pass, other fail

Index