Multiscale Electrophysiology File Format Version 3.0

(MEF3)

Feature Overview:

Feature	Characteristics

Format	• One directory per channel
	• Channel are segmented in time (single segment is channels are supported)
	• Extensible channel types (currently, time series & video)
	• Time series channel:
	    • 32 bit resolution (integer)
	    • Independent channel sampling frequencies
	    • Any time series data can be encoded (e.g. transforms of original data)

Time Series Compression	• Decreased data storage • Increased network transfer, read/write speeds • Variable block sizes • Channel-specific sampling rates supported reduce data volume • Adaptive lossless or lossy compression • Improved compression ratio with decreased signal variance (e.g. filtering) • Independent blocks allow parallel compression / decompression
Encryption	• AES 128-bit • HIPAA compliant • Sharing of human data does not require de-identification procedures • Dual-tiered, single-password encryption scheme allowing differential access to the same file • Unauthorized copies have no access to creator-determined file regions: technical metadata, subject-identifying metadata, specific records, time series data • Times are optionally offset, preserving true time of day, but obscuring actual recording date and time zone. • No encryption level is required
Access	• Rapid random access via indices files • Field alignment facilitates direct variable access after data read

Feature	Characteristics
Analysis	• Separate directory for each channel to facilitate parallel processing • Independence of time series blocks support asynchronous and parallel processing • Multiple precalculated fields facilitate various analyses
Real-time	• The structure of MEF files allows real-time reading and writing. • Catastrophic failure during an acquisition will leave an intact valid MEF structure

Redundancy & Damage mitigation	• 32-bit CRC checksums for detection of file, individual record, & time series block corruption
	• Time Series Channels:
	    • Block independence limits extent of data loss if damage occurs
	    • Block alignment facilitates file recovery
	    • Multiple fields duplicated in block header and indices file
	    • Entire indices file can be reconstructed from data file

Time	• Time discontinuities supported and indexed • μUTC time provides globally accurate date & time of day to microsecond resolution • μUTC time is easily converted to UTC time for use with standard Unix / Posix time functions
Events	• Stored in binary records file • User-defined event types readily accommodated by records format
Video	• Video channels are explicitly supported
Support	• Open source (Apache software license) • Freely available C, Matlab, & Java functions and software

MEF Data Hierarchy (See Figure 1)

- Each collection of recorded channels is called a “Session”. A session is a directory at the top level of the hierarchy.
- A session directory is not required, MEF channels or segments can be acquired and used independently.

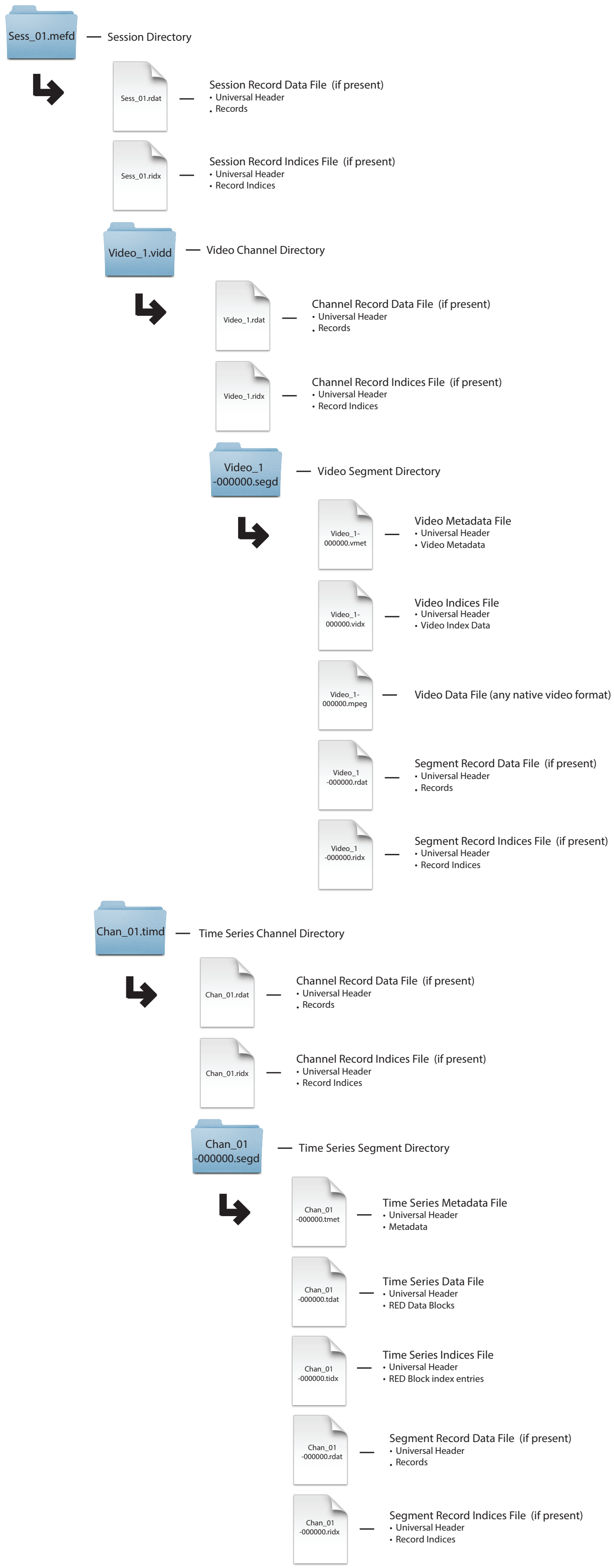
- Channel Directories: Channels are any data stream. Currently time-series and video data are supported, but other channel types may be incorporated in the future.
- All channels are divided into segments. All channels are required to have at least one segment.
- Every level of the hierarchy may have records associated with that level.
- Each Session Directory *(if present)* contains:
  - Record Data File *(if present, a session Record Indices file must be present)*
  - Record Indices File *(if present, a session Record Data file must be present)*
  - Time Series Directories containing:
    - Record Data File *(if present, a channel Record Indices file must be present)*
    - Record Indices File *(if present, a channel Record Data file must be present)*
    - Segment directories containing:
      - (Time Series) Metadata File
      - (Time Series) Data File
      - (Time Series) Indices File
      - Record Data File *(if present, a segment Record Indices file must be present)*
      - Record Indices File *(if present, a segment Record Data file must be present)*
  - Video Channel directories containing:
    - Record Data File *(if present, a channel Record Indices file must be present)*
    - Record Indices File *(if present, a channel Record Data file must be present)*
    - Segment directories containing:
      - (Video) Metadata File
      - (Video) Indices File
      - (Video) Data File (native video format file)
      - Record Data File *(if present, a segment Record Indices file must be present)*
      - Record Indices File *(if present, a segment Record Data file must be present)*

MEF Naming Conventions (See Figure 1)

- Session Directories are named according to user preference and carry the ".mefd" extension.
- Record Data Files are named as the level (session, channel, segment) name appended by ".rdat".
- Record Indices Files are named as the level name appended by ".ridx".

- Time Series Channel Directories are named as the channel name appended by ".timd".
- Video Channel Directories are named according to user preference appended by ".vidd".
- Segment Directories are named with the channel name, hyphenated with sequential fixed-width (6 digit) numbers starting from 0 (e.g. 000000, 000001, ...) appended by ".segd". (e.g. "Chan_01-000000.segd").
- Time Series Metadata Files are named as the segment name appended by ".tmet".
- Time Series Indices Files are named as the segment name appended by ".tidx".
- Time Series Data Files are named as the segment name appended by ".tdat".
- Video Metadata Files are named as the segment name appended by ".vmet".
- Video Indices Files are named with the video directory name appended by ".vidx".
- The Video Data Files are named with the Video Segment Directory name appended by their standard extensions (e.g. "Video_1-000000.mpeg"). There is one video data file per video channel segment.

Figure 1: MEF Data Hierarchy & Naming Conventions



MEF Data Type Definitions:

Type Name	Description
ui1	1 byte unsigned integer
si1	1 byte signed integer
ui4	4 byte unsigned integer
si4	4 byte signed integer
sf4	4 byte signed floating point number
ui8	8 byte unsigned integer
si8	8 byte signed integer
sf8	8 byte signed floating point number
utf8[n]	zero-terminated UTF-8 encoded string of maximum length “n” characters (not including terminal zero)
ascii[n]	zero-terminated ascii encoded string of maximum length “n” characters (not including terminal zero)

MEF Time Series Data Format

- Data are stored in compressed blocks, compressed with the RED (range encoded differences) algorithm.
- RED can encode signed integer data with 32-bit resolution, giving a full range of $-(2^{31})$ to $+(2^{31} - 1)$. [decimal -2,147,483,648 to +2,147,483,647] [hex 0x80000000 to 0x7FFFFFFF]
- -2^{31} is reserved to represent NaN (not a number). [decimal -2,147,483,648] [hex 0x80000000]
- $+(2^{31} - 1)$ is reserved to represent positive infinity. [decimal 2,147,483,647] [hex 0x7FFFFFFF]
- $-(2^{31} - 1)$ is reserved to represent negative infinity. [decimal -2,147,483,647] [hex 0x80000001]
- The unreserved range is therefore $-(2^{31} - 2)$ to $+(2^{31} - 2)$. [decimal -2,147,483,646 to +2,147,483,646] [hex 0x80000002 to 0x7FFFFFFE]
- Data blocks are indexed in the Time Series Indices File for random access.

MEF Data Alignment

- All fields in all files in the format are aligned such that their values align to a multiple of their size from the beginning of the file. This allows for data read to be cast directly into data structures and for memory mapping of files.
- This alignment also facilitates recovery in the event of file damage.
- Pad bytes are added, if necessary, to maintain alignment, at the end of RED Blocks, and Record Bodies. The value of the the pad byte is specified to be 0x7E, the ascii tilde ("~"). Specification of this value is done to facilitate reproducible CRCs and may be useful in the case of data recovery if file damage were to occur.

MEF Strings

- All strings related to naming and descriptive data use UTF-8 encoding to allow for international character sets.
- UTF-8 encoding:
  - variable length characters
  - up to 4 bytes per character
  - not endian-sensitive
  - strings are null-terminated
- Unused bytes in MEF string fields are set to zero to promote reproducibility of CRC values.

Micro-UTC (μUTC) Time

- All times in MEF are represented as μUTC times.
- A μUTC time is an si8 containing the elapsed microseconds since January 1, 1970 at 00:00:00 in the GMT (Greenwich Mean Time) time zone.
- μUTC is simply converted to UTC (Coordinated Universal Time: seconds since 1/1/1970 at 00:00:00 GMT. Referred to as "The Epoch", defined by the International Telecommunications Union) by dividing by 1,000,000.
- In MEF library functions, μUTC times that have had the recording time offset applied to them are made negative to indicate this status. Recording times prior to The Epoch (negative μUTC times) remain possible in MEF 3 by avoiding use of library functions that use negative status as an indicator of being offset (realistically a need for recording times prior to 1970 is unlikely).

Tiered Encryption

- Level 1 and Level 2 encryption can be selected in various places:
    - Sections 2 and 3 of Metadata Files
    - Individual records of Record Data Files
    - Individual RED blocks of the Time Series Data Files
- Level 2 decryption ability guarantees Level 1 decryption ability, but not the converse.
- Level 1 encryption is typically used for technical data, and Level 2 encryption for potentially subject identifying data. This way technical data can be shared with collaborators with out violating subject privacy. The encryption levels can be chosen in any way desired by the file creator, however.
- Level 2 encryption requires specification of a Level 1 password, even if Level 1 encryption is not used anywhere in the file.
- The encryption / decryption algorithm is the 128-bit Advanced Encryption Standard (AES). [http://www.csrc.nist.gov/publications/fips/fips197/fips-197.pdf], which satisfies the Health Insurance Portability and Accountability Act (HIPAA) 112-bit requirement for symmetric encryption of human data.

UTF-8 passwords

- AES-128 requires a 16 byte key. Therefore multibyte UTF-8 password characters are used internally in MEF by taking the last (most unique) byte in each character of the UTF-8 encoding.
- The password length limit is 15 (UTF-8) characters because MEF passwords are required to be null terminated strings.

Recording Time Offsets

- The Recording Time Offset is included in Section 3 of the Metadata files, and if times are not offset this field is set to zero.
- The GMT (Greenwich Meantime) offset should be set to the actual value at the recording site at the start of recording, regardless of whether Recording Time Offsets are used. This is because μUTC times are always relative to GMT, so local time calculation requires this information. The GMT offset is stored as the integer number seconds ahead (positive) or behind GMT (negative). The valid range, in MEF, is -86400 to +86400 (-24 to +24 hours).

- The format does accommodate the possibility of a change in GMT offset during the recording due to the beginning or end of Daylight Saving Time (DST), but does not accommodate more than one start and one stop of DST (i.e. recordings exceeding one year in duration). Recording time offsets are not applied, to these numbers.
- Times that have been offset are made negative to indicate this status.
- As recording time offsets are stored in section 3 of the Metadata files, in. To remove offsets, Metadata files should be read first when reading a segment.

Time Series Compression

- Compression is done by differencing the data, and then range encoding the differences. The algorithm is referred to as RED, for range-encoded differences. RED is a lossless compression.
- Data can optionally be detrended prior to applying RED compression. This operation is lossless, but is generally more useful in lossy compression routines.
- Lossy compression is permitted in time series data by scaling data prior to compression with the RED algorithm. Scaling is adaptive and may vary from block to block. The scaled values must be rounded to the nearest integer, introducing the loss. Lossy compression is not required, but can produce substantial storage savings with negligible data differences in data streams whose sample-value specificities exceed their information content. Compression can also be useful in speeding transmission and viewing of data.
- Four compression modes are currently supported:

  1. Lossless (default)
  2. Fixed Scale Factor: a user-specified scale factor is applied to the block (1.0 results in lossless compression)
  3. Fixed Compression Ratio: the scale factor is adjusted until the block compression ratio (block_bytes / input_array_size [as si4s]) is this number plus or minus a tolerance. e.g. 20% of the original si4 size with a 1% tolerance is 0.19 to 0.21. If lossless compression can achieve or exceed the desired ratio (plus the tolerance), lossless compression will be applied. This option may add noticeable processing time to compression, but once done, adds negligible time to decompression.
  4. Mean Residual Ratio: the scale factor is adjusted until the mean(abs((scaled_data - original_data) / original_data)) for the values in the block, is this number plus or minus a tolerance. e.g. 0.5% difference with a 0.1% tolerance is 0.004-0.006. This option may add noticeable processing time to compression, but once done, adds negligible time to decompression.

Protected and Discretionary File Regions

- The protected region is reserved for possible future additions to the MEF format and should not be used by end users.
- The discretionary region is reserved for end user use so that custom data can be conveniently added to the files without interfering with the specified format fields.
- Protected and discretionary regions can be found in the universal header, each section of the metadata files, RED block header, and indices files.

Encryption Level Schema

- The following table contains codes for encryption that are useful in processing as well as in file encoding.

Encryption Level Schema:

Value	Meaning
0	No encryption
1	Level 1 encrypted
-1	Level 1 encryption specified, currently decrypted
2	Level 2 encrypted
-2	Level 2 encryption specified, currently decrypted
-128	No entry

Universal Header

- Each file in the MEF structure begins with a universal header
- The only current exception is video data files whose format is determined by their specific video format (e.g. MPEG).
- The universal header is not encrypted.

- Design concepts:
  - Contain the minimum information required to read a file in the absence of any other files (e.g. indices or metadata). Appropriate interpretation of the data may still require metadata and passwords. In some file types universal header information may be duplicated in the metadata for convenience.
  - Contain the minimum information to uniquely identify a file and it’s place in a MEF hierarchy.
  - Contain the minimum information required to detect file corruption.
  - Contain no potentially subject identifying information.

Universal Header:

Field	Offset	Bytes	Type	Contents
Header CRC	0	4	ui4	• CRC of the universal header after this field • 0 indicates no entry
Body CRC	4	4	ui4	• CRC of the body of the file after the universal header • 0 indicates no entry
File Type	8	5	ascii[4] or ui4	• 4 ascii characters of file name extension, null terminated or used as ui4 value • 0 (all zeros = zero-length string) indicates no entry
MEF Version Major	13	1	ui1	• numeric value: 3, currently • 0 indicates no entry • 0xFF indicates no entry
MEF Version Minor	14	1	ui1	• numeric value: 0, currently • 0 indicates no entry • 0xFF indicates no entry
Byte Order Code	15	1	ui1	• 0 ==> big-endian • 1 ==> little-endian • 0xFF indicates no entry
Start Time	16	8	si8	• File start time in μUTC format • If recording time offset is used, it is applied here • 0x8000000000000000 indicates no entry
End Time	24	8	si8	• File end time in μUTC format • If recording time offset is used, it is applied here • 0x8000000000000000 indicates no entry

Field	Offset	Bytes	Type	Contents
Number of Entries	32	8	si8	• Number of entries in the file • See **Universal Header Number of Entries** table (below) for the specific meaning for each file type • -1 indicates no entry
Maximum Entry Size	40	8	si8	• Maximum size of an entry in the file • See **Universal Header Number of Entries** table (below) for the specific meaning for each file type • -1 indicates no entry
Segment Number	48	4	si4	• Number of the segment (if applicable) • -1 indicates no entry • -2 indicates channel level • -3 indicates session level
Channel Name	52	256	utf8[63]	• Channel name without path or extension • Zero-length string indicates no entry
Session Name	308	256	utf8[63]	• Session name without path or extension • Zero-length string indicates no entry
Anonymized Name	564	256	utf8[63]	• Anonymized subject name • Zero-length string indicates no entry
Level UUID	820	16	ui1	• 16 random bytes shared by all files in the current level • zeros indicate no entry
File UUID	836	16	ui1	• 16 random bytes unique to the current file • zeros indicate no entry

Field	Offset	Bytes	Type	Contents
Provenance UUID	852	16	ui1	• File UUID of the file from which the current file was derived • zeros indicate no entry • Identity with File UUID indicates that this is the originating file.
Level 1 Encryption Password Validation Field	868	16	ui1	• First 16 binary bytes of a SHA-256 hash of the Level 1 password • zeros indicate no entry
Level 2 Encryption Password Validation Field	884	16	ui1	• Exclusive-or of first 16 bytes of a SHA-256 hash of the Level 2 password with the unhashed Level 1 password • zeros indicate no entry
Protected Region	900	60		• Filled with zeros • Reserved for potential future use
Discretionary Region	960	64		• Filled with zeros if unused • Discretionary end-user use

Universal Header: Number of Entries

File Type	Extension(s)	*Number of Entries* Contents	*Maximum Entry Size* Contents
Record Data File	rdat	• Number of records in the file • -1 indicates no entry	• Number of **bytes** (including record header and pad bytes) in the largest record in the file • -1 indicates no entry
Record Indices File	ridx	• Number of records indices in the file (= number of records) • -1 indicates no entry	• Number of **bytes** in a record index (a constant) • -1 indicates no entry
Metadata Files	tmet vmet	1	• Number of **bytes** in a metadata file (a constant) • -1 indicates no entry
Time Series Data File	tdat	• Number of RED blocks in the file • -1 indicates no entry	• Number of **samples** in the largest RED block in the file • -1 indicates no entry
Time Series Indices File	tidx	• Number of time series indices in the file (= the number of RED blocks) • -1 indicates no entry	• Number of **bytes** in a time series index (a constant) • -1 indicates no entry
Video Indices File	vidx	• Number of video indices (= clips) in the file • -1 indicates no entry	• Maximum number of **bytes** in a clip in the video file. • -1 indicates no entry

Metadata Files

- One for each channel segment in the MEF hierarchy
- The metadata files share an identical format, but section 2 fields are specific to the channel data type.
- Currently there are 2 types of metadata files specified: time-series and video. The first three fields of section 2 are common to all section 2 types: *Channel Description, Session Description, and Recording Duration.*
- Each type of metadata file has it's own file type, which also serves as it's file name extension.

Metadata Files:

Field	Offset	Bytes	Type	Contents	Encryption
Universal Header	0	1024		See "Universal Header" description	None
Section 1					
Section 2 Encryption	1024	1	si1	see Encryption Level Schema table	None
Section 3 Encryption	1025	1	si1	see Encryption Level Schema table	None
Protected Region	1026	766		• Filled with zeros • Reserved for potential future use	None
Discretionary Region	1792	768		• Filled with zeros if unused • Discretionary end-user use	None
Section 2 (technical data)					
Metadata Section 2 Channel Type Specific Fields	2560	10752		See channel type specific tables below	As specified in Section 1
Section 3 (subject specific data)					
Recording Time Offset	13312	8	si8	• value to add to all μUTC times to adjust them to true UTC time • 0x8000000000000000 indicates no entry	As specified in Section 1
DST Start Time	13320	8	si8	• μUTC of Daylight Saving Time start, if occurred during recording • 0 indicates DST did not begin during recording • 0x8000000000000000 indicates no entry	As specified in Section 1

Field	Offset	Bytes	Type	Contents	Encryption
DST End Time	13328	8	si8	• μUTC of Daylight Saving Time end, if occurred during recording • 0 indicates DST did not end during recording • 0x8000000000000000 indicates no entry	As specified in Section 1
GMT offset (at start of recording)	13336	4	si4	• File recording time zone expressed in seconds ahead or behind GMT. Must be added to uUTCs to get local time of day. (e.g. example, 0 indicates GMT, -18000 indicates US Eastern Standard Time) • --86401 indicates no entry (24 hours and 1 second behind GMT)	As specified in Section 1
Subject Name 1	13340	128	utf8[31]	• typically subject first name • Zero-length string indicates no entry	As specified in Section 1
Subject Name 2	13468	128	utf8[31]	• typically subject last name • Zero-length string indicates no entry	As specified in Section 1
Subject ID	13596	128	utf8[31]	• subject ID • Zero-length string indicates no entry	As specified in Section 1
Recording Location	13724	512	utf8[127]	• Typically: Originating Institution, City, Country • Zero-length string indicates no entry	As specified in Section 1
Protected Region	14236	1124		• Filled with zeros • Reserved for potential future use	As specified in Section 1

Field	Offset	Bytes	Type	Contents	Encryption
Discretionary Region	15360	1024		• Filled with zeros if unused • Discretionary end-user use	As specified in Section 1

Time Series Metadata Section 2

Field	Offset	Bytes	Type	Contents	Encryption
Universal Header	0	1024		See “Universal Header” description	None
Section 1 *(see Metadata Section 1)*					
Section 2 (technical data)					
Channel Description	2560	2048	utf8[511]	• Description of recording channel • Zero-length string indicates no entry • Present in all section 2 metadata types	As specified in Section 1
Session Description	4608	2048	utf8[511]	• Description of recording session • Zero-length string indicates no entry • Present in all section 2 metadata types	As specified in Section 1
Recording Duration	6656	8	si8	• Pecording duration in microseconds • -1 indicates no entry • Present in all section 2 metadata types	As specified in Section 1
Reference Description	6664	2048	utf8[511]	• Description of recording reference channel • Zero-length string indicates no entry	As specified in Section 1
Acquisition Channel Number	8712	8	si8	• Number of the channel in the original recording • -1 indicates no entry	As specified in Section 1
Sampling Frequency	8720	8	sf8	• Sampling frequency • -1.0 indicates no entry	As specified in Section 1
Low Frequency Filter Setting	8728	8	sf8	• High-pass filter setting (Hz) • -1.0 indicates no entry	As specified in Section 1

Field	Offset	Bytes	Type	Contents	Encryption
High Frequency Filter Setting	8736	8	sf8	• Low-pass filter setting (Hz) • -1.0 indicates no entry	As specified in Section 1
Notch Filter Frequency Setting	8744	8	sf8	• Notch filter setting (Hz) • -1.0 indicates no notch filter or no entry	As specified in Section 1
AC Line Frequency	8752	8	sf8	• AC line frequency (Hz) • -1.0 indicates no entry	As specified in Section 1
Units Conversion Factor	8760	8	sf8	• Value to multiply sample values by to get native units (“Units Description” field) • 0.0 indicates no entry • Negative values indicate values are inverted (Note: negative values affect Minimum & Maximum Native Sample Value calculation)	As specified in Section 1
Units Description	8768	128	utf8[31]	• String describing units (e.g. “microvolts”) • Zero-length string indicates no entry	As specified in Section 1

Field	Offset	Bytes	Type	Contents	Encryption
Maximum Native Sample Value	8896	8	sf8	• The highest native sample value • Units Conversion Factor is applied to this number • If the Units Conversion Factor is positive, this is the maximum RED sample value times the Units Conversion Factor. If the Units Conversion Factor is negative, this is the minimum RED sample value times the Units Conversion Factor • If lossy compression is used the Scale Factor and offset are also applied to this number • NaN indicates no entry. Note that this means that the contents of this field cannot be directly compared to the NO_ENTRY value, but must be evaluated with a system function such as isnan(). This can fail in principle under different representations of NaN on different systems. • If Units Conversion Factor has no entry, it is presumed to be 1.0 for calculation of this value	As specified in Section 1

Field	Offset	Bytes	Type	Contents	Encryption
Minimum Native Sample Value	8904	8	sf8	• The lowest native sample value • Units Conversion Factor is applied to this number • If the Units Conversion Factor is positive, this is the minimum RED sample value times the Units Conversion Factor. If the Units Conversion Factor is negative, this is the maximum RED sample value times the Units Conversion Factor. • If lossy compression is used the Scale Factor and offset are also applied to this number • NaN indicates no entry. Note that this means that the contents of this field cannot be directly compared to the NO_ENTRY value, but must be evaluated with a system function such as isnan(). This can fail in principle under different representations of NaN on different systems. • If Units Conversion Factor has no entry, it is presumed to be 1.0 for calculation of this value	As specified in Section 1

Field	Offset	Bytes	Type	Contents	Encryption
Start Sample	8912	8	si8	• Number of the first sample in the RED block data relative to all samples in the channel (not the segment) • The first sample number in *first* segment is zero • -1 indicates no entry	As specified in Section 1
Number of Samples	8920	8	si8	• Total recorded samples in the segment • -1 indicates no entry	As specified in Section 1
Number of Blocks	8928	8	si8	• Total recorded RED blocks in the file • -1 indicates no entry • Duplicated in Universal Header of Time Series Indices and Data Files	As specified in Section 1
Maximum Block Bytes	8936	8	si8	• Maximum bytes, including header & pad bytes, in any RED block in the file • -1 indicates no entry	As specified in Section 1
Maximum Block Samples	8944	4	ui4	• Maximum number of samples in a RED block • 0xFFFFFFFF indicates no entry • Duplicated (as an si8) in Universal Header of Time Series Data Files	As specified in Section 1
Maximum Difference Bytes	8948	4	ui4	• Maximum bytes required for the difference data in the compressed blocks • 0xFFFFFFFF indicates no entry	As specified in Section 1

Field	Offset	Bytes	Type	Contents	Encryption
Block Interval	8952	8	si8	• Microseconds between RED blocks • -1 indicates no entry, or that the intervals vary	As specified in Section 1
Number of Discontinuities	8960	8	si8	• Number of discontinuities in the segment • First sample is a discontinuity • -1 indicates no entry	As specified in Section 1
Maximum Contiguous Blocks	8968	8	si8	• Maximum number of contiguous RED blocks between discontinuities in the segment • -1 indicates no entry	As specified in Section 1
Maximum Contiguous Block Bytes	8976	8	si8	• Maximum number of contiguous compressed bytes between discontinuities in the segment (including block headers and pad bytes) • -1 indicates no entry	As specified in Section 1
Maximum Contiguous Samples	8984	8	si8	• Maximum number of contiguous samples between discontinuities • -1 indicates no entry	As specified in Section 1
Protected Region	8992	2160		• Filled with zeros • Reserved for potential future use	As specified in Section 1
Discretionary Region	11152	2160		• Filled with zeros if unused • Discretionary end-user use	As specified in Section 1

Field	Offset	Bytes	Type	Contents	Encryption
Section 3 *(see Metadata Section 3)*					

Video Metadata Section 2

Field	Offset	Bytes	Type	Contents	Encryption
Universal Header	0	1024		See "Universal Header" description	None
Section 1 *(see Metadata Section 1)*					
Section 2 (technical video data)					
Channel Description	2560	2048	utf8[511]	• Description of the video stream • Zero-length string indicates no entry • Present in all section 2 types	As specified in Section 1
Session Description	4608	2048	utf8[511]	• Description of recording session • Zero-length string indicates no entry • Present in all section 2 types	As specified in Section 1
Recording Duration	6656	8	si8	• recording duration in microseconds • -1 indicates no entry • Present in all section 2 types	As specified in Section 1
Horizontal Resolution	6664	8	si8	• Horizontal pixels • -1 indicates no entry	As specified in Section 1
Vertical Resolution	6672	8	si8	• Vertical pixels • -1 indicates no entry	As specified in Section 1
Frame Rate	6680	8	sf8	• frames per second • -1.0 indicates no entry or variable frame rate	As specified in Section 1

Field	Offset	Bytes	Type	Contents	Encryption
Number of Clips	6688	8	si8	• Number of clips (= video indices) in the video index file • -1 indicates no entry • Duplicated in Universal Header of Video Indices Files	As specified in Section 1
Maximum Clip Bytes	6696	8	si8	• Maximum bytes in a clip in the video file • -1 indicates no entry	As specified in Section 1
Video Format	6704	128	utf8[31]	• e.g. "MPEG-4" • Zero-length string indicates no entry	As specified in Section 1
Video File CRC	6832	4	ui4	• CRC of the video file. • 0 indicates no entry	As specified in Section 1
Protected Region	6836	3236		• Filled with zeros • Reserved for potential future use	As specified in Section 1
Discretionary Region	10072	3240		• Filled with zeros if unused • Discretionary end-user use	As specified in Section 1
Section 3 *(see Metadata Section 3)*					

Records Data File

- Binary format described below
- Can be present at any level of the MEF hierarchy, but is never required.
- If a Records Data File is present, a Records Index File must also be present, and vice versa.
- Each record begins with a record header
- Example record types include:
    - Electrode & probe descriptions

- Electrode coordinates
- Electrode diagrams
- Spike records
- Seizure marks
- Event related study data
- Sleep stage / behavioral state
- Miscellaneous notes
- Acquisition system log entries
- Acquisition system configuration
- End-user defined record types

- Records can also be compressed, but the specific compression algorithm (e.g. jpeg, png, bzip) should be defined in the record body.
- The length of the body of each record must be padded to a multiple of 16 for encryption. The pad-byte value is 0xFE (ascii tilde, “~”).

Records Data File:

Field	Offset	Bytes	Contents
Universal Header	0	512	*See “Universal Header” description*
Records	512		*See “Record Header Format” description*
…			

Record Header Format:

Field	Offset	Bytes	Type	Contents	Encryption
Record CRC	0	4	ui4	• Cyclically redundant checksum for record and remainder of Record Header • 0 indicates no entry	None
Type	4	5	ascii[4] or ui4	• 4 byte integer, typically representing 4 ascii characters, designating record type, null terminated, or used as ui4 value • 0 (all zeros = zero-length string) indicates no entry	None
Record Version Major	9	1	ui1	• Record type's major version • 0xFF indicates no entry	None
Record Version Minor	10	1	ui1	• Record type's minor version • 0xFF indicates no entry	None
Encryption	11	1	si1	*see "Encryption Level Schema" table*	None
Bytes	12	4	ui4	• Record size in bytes, excluding record header, including pad bytes if any. • 0 indicates no entry	None
Time	16	8	si8	• Record time in μUTC time format. • If recording time offset is used for the session it is applied here also. • 0x8000000000000000 indicates no entry	None

Record Indices File Format

- Universal header
- Sequential record index data
- 8-byte boundary aligned

Record Indices File:

Field	Offset	Bytes	Contents
Universal Header	0	512	*See "Universal Header" description*
Record Index	512	24	*See "Record Index Format" description*
…			

Record Index Format:

Field	Offset	Bytes	Type	Contents
Type	0	5	ascii[4] or ui4	• 4 byte integer, typically representing 4 or used as ui4 value, designating record type, null terminated, or used as ui4 value • 0 (all zeros = zero-length string) indicates no entry
Major Version	5	1	ui1	• Record type's major version • 0xFF indicates no entry
Minor Version	6	1	ui1	• Record type's minor version • 0xFF indicates no entry
Encryption	7	1	si1	*see "Encryption Level Schema" table*
File Offset	8	8	si8	• Record start file offset in bytes. • -1 indicates no entry
Time	16	8	si8	• Record time in μUTC time format. • If recording time offset is used for the session it is applied here also. • 0x8000000000000000 indicates no entry

Time Series Indices File Format

- Universal header
- Sequential time series index data
- 8-byte boundary aligned

Time Series Indices File:

Field	Offset	Bytes	Contents
Universal Header	0	512	*See "Universal Header" description*
Time Series Index…	512	32	*See "Time Series Index Format" description*
...			

Time Series Index Format:

Field	Offset	Bytes	Type	Contents
File Offset	0	8	si8	• RED block file offset in bytes. • -1 indicates no entry
Start Time	8	8	si8	• μUTC time • If recording time offset is used for the session it is applied here also. • 0x8000000000000000 indicates no entry
Start Sample	16	8	si8	• Number of the first sample in the RED block data relative to all samples in the segment (not the channel). • The first sample number in *every* segment is zero. • -1 indicates no entry
Number of Samples	24	4	ui4	• Number of samples in the RED block • 0xFFFFFFFF indicates no entry
Block Bytes	28	4	ui4	• Bytes in RED block including header & pad bytes • 0xFFFFFFFF indicates no entry
Maximum Sample Value	32	4	si4	• Maximum sample value in the block • Units Conversion Factor is not applied to this number • If lossy compression is used the Scale Factor is applied to this number • If a block offset is used, it is applied to this number • RED NaN (0x80000000) indicates no entry
Minimum Sample Value	36	4	si4	• Minimum sample value in the block • Units Conversion Factor is not applied to this number • If lossy compression is used the Scale Factor is applied to this number • If a block offset is used, it is applied to this number • RED NaN (0x80000000) indicates no entry

Field	Offset	Bytes	Type	Contents
Protected Region	40	4		• Filled with zeros • Reserved for potential future use
RED Block Flags	44	1	ui1	• From RED block header • See RED Block Flags table below.
RED Block Protected Region	45	3		From RED block header
RED Block Discretionary Region	48	8		From RED block header

Video Indices File Format

- Universal header
- Sequential video index data
- 8-byte boundary aligned

Video Indices File:

Field	Offset	Bytes	Type	Contents
Universal Header	0	512		*See "Universal Header" description*
		Block Indices Data		
Video Index	512	40		*See "Video Index Format" description*
...				

Video Index Format:

Field	Offset	Bytes	Type	Contents
Start Time	0	8	si8	• μUTC time of first frame in clip. • If recording time offset is used for the session it is applied here also. • 0x8000000000000000 indicates no entry
End Time	8	8	si8	• μUTC time of last frame in clip. • If recording time offset is used for the session it is applied here also. • 0x8000000000000000 indicates no entry
Start Frame	16	4	ui4	• Number of the first frame in the clip in the video file. • Numbering starts at zero. • 0xFFFFFFFF indicates no entry
End Frame	20	4	ui4	• Number of the last frame in the clip in the video file. • Numbering starts at zero. • 0xFFFFFFFF indicates no entry
File Offset	24	8	si8	• File offset to frame, typically a keyframe, depending on format • -1 indicates no entry
Clip Bytes	32	8	si8	• Number of bytes in the clip. • -1 indicates no entry
Protected Region	40	16		• Filled with zeros • Reserved for potential future use
Discretionary Region	56	8		• Filled with zeros if unused • Discretionary end-user use

Time Series Data File Format

- Universal header
- Sequential RED blocks

- Each block is 8-byte boundary aligned

Time Series Data Encryption

- Optionally the time series data can be encrypted with either Level 1 or 2 encryption
- The encryption uses AES-128 to encrypt the first 16 (typically most significant) bytes of the statistical model in each RED compressed block.
- Encryption / decryption adds negligible time to data processing.

Time Series Data File:

Field	Offset	Bytes	Type	Contents
Universal Header	0	512		See “Universal Header” description
RED Block	512	varies		See “RED Block Format” description
...				

RED Blocks

- Data are stored in compressed independent blocks
- Raw data are differenced. Differences are encoded in a single signed byte. If there is overflow, i.e > +127 or < -127, then a keysample is introduced flagged by the reserved value -128. The 4 bytes following the keysample flag contain the full undifferenced value of the (second) data point generating the overflow difference, as an si4.
- The differenced data are statistically modeled, the model is stored in the RED block header.
- Range encoding is used to compress the differences, using the statistical model.
- Blocks are required to be 8-byte boundary aligned, and are terminally padded to an 8-byte boundary with the value 0x7E (tilde, “~”) as necessary. Pad bytes are included in the block bytes value, and in the block CRC.
- In compression, if the RED_PROCESSING_DIRECTIVE detrend_data is set, each sample will be dtretended prior to scaling and compressing. The slope and intercept will be stored in the block header. This is a lossless operation, but has more utility in lossy compression.

- In compression, if the value of the scale_factor is greater than 1.0, the (possibly offset) values will be divided by this value and rounded, prior to differencing. This is a lossy operation.
- In decompression, if the value of the scale_factor is greater than 1.0, the values of the samples will be multiplied by this value and rounded after un-differencing.
- In decompression, if the block offset is non-zero, this value will be added to each of the samples after un-differencing and possibly scaling.

RED Block Format:

Field	Offset	Bytes	Type	Contents
Block CRC	0	4	ui4	• CRC of the remainder of block • 0 indicates no entry
Flags	4	1	ui1	See RED Block Flags table below.
Protected Region	5	3		reserved for future use
Discretionary Region	8	8		discretionary end-user use
Detrend Slope	16	4	sf4	• Combined with Detrend Intercept to detrend the data in a block • This is a lossless procedure, but adds time to compression & decompression. • 0.0 in BOTH Detrend Slope and Detrend Intercept indicates no entry
Detrend Intercept	20	4	sf4	• Combined with Detrend Slope to detrend the data in a block • This is a lossless procedure, but adds time to compression & decompression. • 0.0 in BOTH Detrend Slope and Detrend Intercept indicates no entry
Scale Factor	24	4	sf4	• Values in block are divided by this value and rounded for lossy compression. • Values in block are multiplied by this value and rounded for decompression. • 1.0 indicates lossless compression • Values < 1.0 are invalid
Difference Bytes	28	4	ui4	Number of difference bytes in the encoded block
Number of Samples	32	4	ui4	Number of data samples encoded in the block
Block Bytes	36	4	ui4	Number of bytes in the compressed block including header and pad (boundary alignment) bytes.

Field	Offset	Bytes	Type	Contents
Start Time	40	8	si8	• *μ*UTC time. • If recording time offset is used for the session it is applied here also.
Statistics	48	256	ui1	• Statistical model of difference values for the block. • The first 16 bytes in the model are be encrypted if data encryption is used.
Compressed Data	304	*varies*		RED-encoded data
Pad bytes		*varies*		pad byte value (0x7E) repeated as necessary to maintain 8-byte alignment

RED Block Flags:

Field	Name	Contents
Bit 0	Discontinuity Bit	• 0 indicates no discontinuity • 1 indicates that this block began after a discontinuity in recording. • The first block in a file is always considered a discontinuity.
Bit 1	Level 1 Encrypted Block Bit	• 0 indicates the block is not currently level 1 encrypted. • 1 indicates the block is currently level 1 encrypted. • The encryption level desired is set by the “encryption” field in the RED_PROCESSING_DIRECTIVES. • This bit is mutually exclusive with “Level 2 Encrypted Block Bit” (bit 2)
Bit 2	Level 2 Encrypted Block Bit	• 0 indicates the block is not currently level 2 encrypted. • 1 indicates the block is currently level 2 encrypted. • The encryption level desired is set by the “encryption” field in the RED_PROCESSING_DIRECTIVES. • This bit is mutually exclusive with “Level 1 Encrypted Block Bit” (bit 1)
Bits 3 - 7		reserved for future use

Meflib API

The meflib API functions and headers are contained in the files “meflib.c” & “meflib.h”. While this is open source, the general idea is that user-defined code not be added to these files. User defined records are defined and coded in “mefrec.c” and “mefrec.h”. The functions required for adding a new record type are described in “MEF 3 Records Specification”

```
/************************************************************************************/
/******************************** Elemental Typedefs ********************************/
/************************************************************************************/

typedef char                si1;
typedef unsigned char       ui1;
typedef short               si2;
typedef unsigned short      ui2;
typedef int                 si4;
typedef unsigned int        ui4;
typedef long int            si8;
typedef long unsigned int   ui8;
typedef float               sf4;
typedef double              sf8;
typedef long double         sf16;  // NOTE: it often requires an explicit compiler instruction
                                   // to implement true long floating point math.
                                   // In icc and gcc the instruction is:
                                   // “-Qoption,cpp,–extended_float_type"
```

These typedefs are used throughout the library to facilitate compilation on systems with different word sizes.

The first character indicates signedness, “s” for signed, “u” for unsigned.

The second character indicates format: “i” for integer type, “f” for floating point type.

The final number indicates the number of bytes in the type, 1, 2, 4, 8, or 16

example: “si4” indicates a signed integer of 4 byte length

```
/************************************************************************************/
/************************************ MEF Booleans **********************************/
/************************************************************************************/

#define MEF_TRUE            1
#define MEF_UNKNOWN         0
#define MEF_FALSE           -1
```

A balanced ternary schema including true, unknown, & false states. This is used throughout the library, and is typically represented in an si1 type.

```
/***********************************************************************************/
/********************************** MEF Globals ********************************/
/***********************************************************************************/


// Structures
typedef struct {
        // time constants
        si8     recording_time_offset;
        ui4     recording_time_offset_mode;
        si4     GMT_offset;
        si8     DST_start_time;
        si8     DST_end_time;

        // alignment fields
        si4     universal_header_aligned;
        si4     metadata_section_1_aligned;
        si4     time_series_metadata_section_2_aligned;
        si4     video_metadata_section_2_aligned;
        si4     metadata_section_3_aligned;
        si4     all_metadata_structures_aligned;
        si4     time_series_indices_aligned;
        si4     video_indices_aligned;
        si4     RED_block_header_aligned;
        si4     record_header_aligned;
        si4     record_indices_aligned;
        si4     all_record_structures_aligned;
        si4     all_structures_aligned;

        // RED
        sf8     *RED_normal_CDF_table;

        // CRC
        ui4     *CRC_table;
        ui4     CRC_mode;

        // AES tables
        si4     *AES_sbox_table;
        si4     *AES_rcon_table;
        si4     *AES_rsbox_table;

        // SHA256 tables
        ui4     *SHA256_h0_table;
        ui4     *SHA256_k_table;

        // UTF8 tables
        ui4     *UTF8_offsets_from_UTF8_table;
        si1     *UTF8_trailing_bytes_for_UTF8_table;

        // miscellaneous
        si4     verbose;
        ui4     behavior_on_fail;
        ui4     file_creation_umask;
```

```
} MEF_GLOBALS;


/ Global Defaults
#define MEF_GLOBALS_VERBOSE_DEFAULT                        MEF_FALSE
#define MEF_GLOBALS_RECORDING_TIME_OFFSET_DEFAULT          0
#define MEF_GLOBALS_RECORDING_TIME_OFFSET_MODE_DEFAULT     (RTO_APPLY_ON_OUTPUT |
                                                           RTO_REMOVE_ON_INPUT)
#define MEF_GLOBALS_GMT_OFFSET_DEFAULT                     0
#define MEF_GLOBALS_DST_START_TIME_DEFAULT                 UUTC_NO_ENTRY
#define MEF_GLOBALS_DST_END_TIME_DEFAULT                   UUTC_NO_ENTRY
#define MEF_GLOBALS_FILE_CREATION_UMASK_DEFAULT            S_IWOTH  // defined in <sys/stat.h>
#define MEF_GLOBALS_BEHAVIOR_ON_FAIL_DEFAULT               EXIT_ON_FAIL
#define MEF_GLOBALS_CRC_MODE_DEFAULT                       CRC_CALCULATE_ON_OUTPUT
```

These values are used throughout the library in a thread-safe manner. They are initialized to the application heap via the function initialize_MEF_globals(), which is in turn called by initialize_meflib(). These two functions are described below.

The recording_time_offset and GMT_offset constants will be described with the recording time offset functions. The alignment fields will be discussed with the alignment checking functions. The CRC_mode constants and CRC_table will be described with the CRC functions. Likewise, the AES, UTF-8 and, SHA lookup tables will be discussed in their respective sections below.

```
/************************************************************************************/
/*********************** Error Checking Standard Functions ************************/
/************************************************************************************/

// Constants
#define USE_GLOBAL_BEHAVIOR   0
#define RESTORE_BEHAVIOR      1
#define EXIT_ON_FAIL          2
#define RETURN_ON_FAIL        4
#define SUPPRESS_ERROR_OUTPUT 8

// Function Prototypes

void    *e_calloc(size_t n_members, size_t size, const si1 *function, si4 line,
        ui4 behavior_on_fail);

FILE    *e_fopen(si1 *path, si1 *mode, const si1 *function, si4 line, ui4 behavior_on_fail);

size_t  e_fread(void *ptr, size_t size, size_t n_members, FILE *stream, si1 *path,
        const si1 *function, si4 line, ui4 behavior_on_fail);

si4     e_fseek(FILE *stream, size_t offset, si4 whence, si1 *path, const si1 *function, si4
        line, ui4 behavior_on_fail);

long    e_ftell(FILE *stream, const si1 *function, si4 line, ui4 behavior_on_fail);

size_t  e_fwrite(void *ptr, size_t size, size_t n_members, FILE *stream, si1 *path,
        const si1 *function, si4 line, ui4 behavior_on_fail);
```

```
void    *e_malloc(size_t n_bytes, const si1 *function, si4 line, ui4 behavior_on_fail);

void    *e_realloc(void *ptr, size_t n_bytes, const si1 *function, si4 line,
        ui4 behavior_on_fail);
```

These functions are provided for convenience. They call their corresponding standard c functions (e.g. e_calloc() calls calloc()), but have built in error messaging. The behavior_on_fail parameter defines what the function does on failure.

example:

```
ui4     behavior;
si4     *data;

behavior = (RETURN_ON_FAIL | SUPPRESS_ERROR_OUTPUT);
data = (si4 *) e_calloc((size_t) buffer_size, sizeof(si4), __FUNCTION__, __LINE__, behavior);
```

__FUNCTION__ and __LINE__ are compiler macros replaced with the function name and line of the function in which they occur; these can contain any string and number, however, for more complex failure tracking. Because of the way in which the behavior parameter is defined, on failure, this call to e_calloc() will return NULL, as would calloc(), and no error messages will be displayed. If USE_GLOBAL_BEHAVIOR is passed into this parameter, the MEF_global value of behavior_on_fail will be used. This is the most common usage in the library. At the time of this writing the default global behavior_on_fail value is EXIT_ON_FAIL, which will produce error messages and then exit the program.

```
/**************************************************************************************/
/*************************** Alignment Checking Functions *************************/
/**************************************************************************************/

// Prototypes
si4                     check_all_alignments(const si1 *function, si4 line);

si4                     check_metadata_alignment(ui1 *bytes);

si4                     check_metadata_section_1_alignment(ui1 *bytes);

si4                     check_metadata_section_3_alignment(ui1 *bytes);

si4                     check_record_header_alignment(ui1 *bytes);

si4                     check_record_indices_alignment(ui1 *bytes);

si1                     check_record_structure_alignments(ui1 *bytes);

si4                     check_RED_block_header_alignment(ui1 *bytes);

si4                     check_time_series_indices_alignment(ui1 *bytes);

si4                     check_time_series_metadata_section_2_alignment(ui1 *bytes);

si4                     check_universal_header_alignment(ui1 *bytes);

si4                     check_video_indices_alignment(ui1 *bytes);

si4                     check_video_metadata_section_2_alignment(ui1 *bytes);
```

The structures in the MEF library are designed such that they can be read in directly from disk to the structure without explicit assignment operations for each of the fields. Because compilers can rearrange fields within structures, this can fail in principle, but the fields are laid out such that this would be quite unlikely.

For example, on a 64 bit CPU structures are generally laid out on 8 byte boundaries. If they are not inherently 8 byte aligned, the compiler will often pad the structure. Explicitly padding the structure to create 8 byte alignment will alleviate this problem. Likewise an 8 byte data type should fall on a natural 8 byte boundary within the structure, if it does not the compiler may try to rearrange or pad the structure. In practice designing a structure such that the compiler will leave it intact is usually quite easy. In the case of alignment failure, the library would need to be updated to perform explicit assignment.

The alignment checking functions simply compare compiler generated offsets to expected offsets from the layout on disk. If all the field offsets match, the functions return MEF_TRUE, if they do not they return MEF_FALSE. Prior to checking, the global alignment flags are each set to MEF_UNKNOWN. In addition to a return value, each of these functions also sets its corresponding MEF_GLOBAL field to MEF_TRUE or MEF_FALSE.

The function check_all_alignments() calls all of the other alignment checking functions and returns MEF_TRUE if all of those functions return MEF_TRUE. This function also takes a function and line argument similar to the error checking functions. This function is called from initialize_meflib(), and so need not be called explicitly if initialize_meflib() is called.

If a buffer (the “bytes” field) is passed the function will not allocate any memory for the testing. If NULL is passed in the “bytes” field the function will allocate memory for the testing and then free it once the check is complete.

example 1 (adapted from check_all_alignments()):

```
…
bytes = (ui1 *) e_malloc(METADATA_FILE_BYTES, __FUNCTION__, __LINE__, USE_GLOBAL_BEHAVIOR);
// METADATA is largest fixed file structure, so this will be enough memory to check all
// the library structures

// check all structures
return_value = MEF_TRUE;
if ((check_universal_header_alignment(bytes)) == MEF_FALSE)
        return_value = MEF_FALSE;
if ((check_metadata_alignment(bytes)) == MEF_FALSE)
        return_value = MEF_FALSE;
if ((check_RED_block_header_alignment(bytes)) == MEF_FALSE)
        return_value = MEF_FALSE;
if ((check_time_series_indices_alignment(bytes)) == MEF_FALSE)
        return_value = MEF_FALSE;
```

```
if ((check_video_indices_alignment(bytes)) == MEF_FALSE)
        return_value = MEF_FALSE;
if ((check_record_indices_alignment(bytes)) == MEF_FALSE)
        return_value = MEF_FALSE;
if ((check_record_header_alignment(bytes)) == MEF_FALSE)
        return_value = MEF_FALSE;
if ((check_record_structure_alignments(bytes)) == MEF_FALSE)
        return_value = MEF_FALSE;

free(bytes);

return(return_value);
```

example 2 (the most common use):

```
return_value = check_all_alignments(__FUNCTION__, __LINE__);
```

```
/************************************************************************************/
/*************************** General Purpose MEF Functions *************************/
/************************************************************************************/
```

As a group, these functions facilitate working with various aspects of the MEF format. Each will be described separately below.

FUNCTION: all_zeros()

```
// Prototype
si1     all_zeros(ui1 *bytes, si4 field_length);
```

all_zeros() returns MEF_TRUE if field pointed to by "bytes" contains all zeros, MEF_FALSE if not. The expected length of the field is passed in "field_length". It is useful in checking fields whose "no entry" value is defined to be all zeros.

example (from show_universal_header()):

```
if (all_zeros(uh->level_1_password_validation_field, PASSWORD_VALIDATION_FIELD_BYTES) ==
MEF_TRUE)
        printf("Level 1 Password Validation_Field: no entry\n");
```

FUNCTION: allocate_file_processing_struct()

```
// Prototype
FILE_PROCESSING_STRUCT *allocate_file_processing_struct(si8 raw_data_bytes, ui4 file_type_code,
        FILE_PROCESSING_DIRECTIVES *directives, FILE_PROCESSING_STRUCT *proto_fps, si8
        bytes_to_copy);
```

This function allocates a FILE_PROCESSING_STRUCT and returns a pointer to it.

```
// Structures
typedef struct {
	si1				full_file_name[MEF_FULL_FILE_NAME_BYTES];  // full path
										   // including extension
	FILE				*fp;
	si4				fd;  // file descriptor
	si8				file_length;
	ui4				file_type_code;
	UNIVERSAL_HEADER		*universal_header;
	FILE_PROCESSING_DIRECTIVES	directives;
	PASSWORD_DATA			*password_data;  // this will often be the same for all
							 // files
	METADATA			metadata;
	TIME_SERIES_INDEX		*time_series_indices;
	VIDEO_INDEX			*video_indices;
	ui1				*records;
	RECORD_INDEX			*record_indices;
	ui1				*RED_blocks;
	si8				raw_data_bytes;
	ui1				*raw_data;
} FILE_PROCESSING_STRUCT;

typedef struct {
	si1				close_file;
	si1				free_password_data;  // when freeing FPS
	si8				io_bytes;  // bytes to read or write
	ui4				lock_mode;
	ui4				open_mode;
} FILE_PROCESSING_DIRECTIVES;

// Constants

// File Processing constants
#define FPS_FILE_LENGTH_UNKNOWN			-1
#define FPS_FULL_FILE				-1
#define FPS_NO_LOCK_TYPE			~(F_RDLCK | F_WRLCK | F_UNLCK)
#define FPS_NO_LOCK_MODE			0
#define FPS_READ_LOCK_ON_READ_OPEN		1
#define FPS_WRITE_LOCK_ON_READ_OPEN		2
#define FPS_WRITE_LOCK_ON_WRITE_OPEN		4
#define FPS_WRITE_LOCK_ON_READ_WRITE_OPEN	8
#define FPS_READ_LOCK_ON_READ			16
#define FPS_WRITE_LOCK_ON_WRITE			32
#define FPS_NO_OPEN_MODE			0
#define FPS_R_OPEN_MODE				1
#define FPS_R_PLUS_OPEN_MODE			2
#define FPS_W_OPEN_MODE				4
#define FPS_W_PLUS_OPEN_MODE			8
#define FPS_A_OPEN_MODE				16
#define FPS_A_PLUS_OPEN_MODE			32
#define FPS_GENERIC_READ_OPEN_MODE		(FPS_R_OPEN_MODE | FPS_R_PLUS_OPEN_MODE |
						FPS_W_PLUS_OPEN_MODE | FPS_A_PLUS_OPEN_MODE)
#define FPS_GENERIC_WRITE_OPEN_MODE		(FPS_R_PLUS_OPEN_MODE | FPS_W_OPEN_MODE |
						FPS_W_PLUS_OPEN_MODE | FPS_A_OPEN_MODE |
						FPS_A_PLUS_OPEN_MODE)
```

```
// File Processing Directives defaults
#define FPS_DIRECTIVE_CLOSE_FILE_DEFAULT               MEF_TRUE
#define FPS_DIRECTIVE_FREE_PASSWORD_DATA_DEFAULT       MEF_FALSE
#define FPS_DIRECTIVE_LOCK_MODE_DEFAULT                (FPS_READ_LOCK_ON_READ_OPEN |
                                                       FPS_WRITE_LOCK_ON_WRITE_OPEN |
                                                       FPS_WRITE_LOCK_ON_READ_WRITE_OPEN)
#define FPS_DIRECTIVE_OPEN_MODE_DEFAULT                FPS_NO_OPEN_MODE
#define FPS_DIRECTIVE_IO_BYTES_DEFAULT                 FPS_FULL_FILE  // bytes to read or
                                                                      // write
```

The FILE_PROCESSING_STRUCT (FPS) is the fundamental file handling unit of the MEF library. The raw_data field contains the data as it is arranged in the MEF structures, and on disk. The universal_header pointer within the FPS will be assigned the value of the start of the raw_data array. Depending on file type, one of the other pointers within the structure will be assigned to the raw_data array after the universal header region.

The passed parameter raw_data_bytes determines the amount of memory allocated to the raw_data field. If this parameter is greater than or equal to UNIVERSAL_HEADER_BYTES, the universal_header pointer is assigned to the raw_data field.

The FILE_PROCESSING_STRUCT's file_length field is set to FPS_FILE_LENGTH_UNKNOWN upon allocation. This value is updated to reflect the current length of the file on disk (in bytes) during read and write operations.

If a prototype FILE_PROCESSING_STRUCT is passed in proto_fps, its directives, password data, and raw data are copied to the new FILE_PROCESSING_STRUCT (unless bytes_to_copy is greater than raw_data_bytes). The amount of raw_data copied is specified in the bytes_to_copy field. If proto_fps is NULL, no copying is performed. If copying is performed, the universal header's CRC will be not be calculated, and may be inaccurate. This is updated in write_MEF_file() before write out, and so is not usually an issue. It could be explicitly calculated with calculate_CRC().

If a pointer to a FILE_PROCESSING_DIRECTIVES structure is passed, These values are copied into the new FPS's directives. These supersede any directives passed in the prototype FPS's directives. If this pointer is NULL and the prototype FPS pointer is NULL, the directives are set to their default values.

The FILE_PROCESSING_DIRECTIVES are used by the reading and writing functions. Specifically, **close_file** tells reading & writing functions to to close the file when they are finished. **free_password_data** tells functions freeing a FILE_PROCESSING_STRUCT to free this also. This is often undesirable as the pointer to a single PASSWORD_DATA structure is often shared between many FILE_PROCESSING_STRUCTs. At this writing the default value of **the free_password_data directive** is MEF_FALSE**. io_bytes** tells reading & writing functions how much of the file to read or write. By default this is the whole file, but this is an impractical choice for very large files that should be processed piecemeal such as the time series data files, or some record data files. **lock_mode** specifies *advisory locking* on the file. All the MEF library functions observe the advisory locking mechanism, to facilitate parallel processing of files. Note that, as this is *advisory* only, external functions may choose to ignore these locks. **open_mode** specifies how a file should be opened, and corresponds to standard Unix / Posix opening modes. This parameter interacts with the **lock_mode** parameter. **read_time_series_data** specifies that time series segment data should be read when reading in a segment data file. At this writing, the default value of this directive is MEF_FALSE. Likewise **read_records_data** species that all the records data should be read in when reading a records data file. At this writing, the default value of this directive is MEF_FALSE also. Records and time series data files can be very large and so reading the whole file is often undesirable, hence the default value of MEF_FALSE for these directives. These directives are used by the functions read_MEF_session(), read_MEF_channel(), and read_MEF_segment(). They are *not* used by read_MEF_file() which uses the io_bytes parameter to determine how much of a file to read.

The file_type_code specifies which of the FILE_PROCESSING_STRUCT pointers will be assigned to the raw_data after the universal header. The file_type_string field of the universal header is also set by the file_type_code. If the file_type_code is zero, these assignments are not made.

The raw_data_bytes parameter specifies how much memory to allocate to the raw_data array. This value is copied into the corresponding member of the new FPS.

example 1: allocate an empty FILE_PROCESSING_STRUCT

```
fps = allocate_file_processing_struct(0, 0, NULL, NULL, 0);
```

example 2: allocate an empty FILE_PROCESSING_STRUCT with space for just a universal header

```
fps = allocate_file_processing_struct(UNIVERSAL_HEADER_BYTES, 0, NULL, NULL, 0);
```

example 3: allocate a metadata FILE_PROCESSING_STRUCT and copy its universal header from the prototype FPS, “other_fps”

```
fps = allocate_file_processing_struct(METADATA_FILE_BYTES, TIME_SERIES_METADATA_FILE_TYPE_CODE,
NULL, other_fps, UNIVERSAL_HEADER_BYTES);
```

example 4: allocate a metadata FILE_PROCESSING_STRUCT and copy all of the data, including the universal header from “other_metadata_fps”.

```
fps = allocate_file_processing_struct(METADATA_FILE_BYTES, TIME_SERIES_METADATA_FILE_TYPE_CODE,
NULL, other_metadata_fps, METADATA_FILE_BYTES);
```

FUNCTION: apply_recording_time_offset()

```
// Prototype
```

```
void	apply_recording_time_offset(si8 *time);
```

The global recording time offset is applied to the passed µUTC time. If the value is negative, it is presumed to already have had the recording time offset applied, and nothing is done. The converse function is remove_recording_time_offset() described below.

FUNCTION: check_password()

```
// Prototype
si4	check_password(si1 *password, const si1 *function, si4 line);
```

Checks that the password pointer is not NULL, and that the password length is less than or equal to PASSWORD_BYTES. Returns 0 on success, 1 on failure. This function does not validate the password against the password validation fields. Process_password_data() does this. In fact, process_password_data() is the only library function to call check_password().

example (from process_password_data()):

```
if (check_password(unspecified_password, __FUNCTION__, __LINE__) == 0)
	// password is not NULL, and is of valid length
```

FUNCTION: count_directories()

```
// Prototype
si4	count_directories(si1 *enclosing_directory, si1 *extension);
```

Returns the number of directories with the specified extension in an enclosing directory. This function can be useful for knowing how many channels or segments currently exist in a parallel processing situation.

FUNCTION: cpu_endianness()

```
// Constants
#define MEF_BIG_ENDIAN		0
#define MEF_LITTLE_ENDIAN	1

// Prototype
ui1	cpu_endianness();
```

Returns MEF_BIG_ENDIAN on big-endian machines and MEF_LITTLE_ENDIAN on little-endian machines. The current library only supports little-endian MEF files, but the specification supports both. If there is a future demand for big-endian MEF, the library can be updated.

example (from initialize_meflib()):

```
if (cpu_endianness() != MEF_LITTLE_ENDIAN)
        fprintf(stderr, "Error: Library only coded for little-endian machines currently\n");
```

FUNCTION: decrypt_metadata()

```
// Prototype
si4     decrypt_metadata(FILE_PROCESSING_STRUCT *fps);
```

Decrypts sections 2 and 3 of metadata file (passed in fps) if they are currently encrypted and if access level is sufficient. It marks decrypted sections as decrypted (negative of encryption level) in section 1 of the metadata.

It returns zero on success.

FUNCTION: decrypt_records()

```
// Prototype
si4     decrypt_records(FILE_PROCESSING_STRUCT *fps);

// Constant
#define UNKNOWN_NUMBER_OF_ENTRIES     -1
```

Decrypts records if they are currently encrypted and the access level is sufficient as specified in the record header. Marks decrypted records as decrypted (negative of encryption level) in record header. If the number of records is known (stored in the universal header **number_of_entries** field), this value is used, if that is set to UNKNOWN_NUMBER_OF_ENTRIES the function will work, as long as the FILE_PROCESSING_STRUCT's raw_data_bytes field reflects an integral number of records.

The function also applies or removes the recording time offset to the times in the record headers according to the value of the the MEF_global recording_time_offset_mode.

It returns zero on success.

FUNCTION: encrypt_metadata()

```
// Prototype
si4     encrypt_metadata(FILE_PROCESSING_STRUCT *fps);
```

Encrypts sections 2 and 3 of metadata file (passed in fps), if they are currently decrypted, to the encryption level specified in section 1 of the metadata. It marks encrypted sections as encrypted (positive of encryption level) in section 1 of the metadata.

It returns zero on success.

FUNCTION: encrypt_records()

```
// Constant
#define UNKNOWN_NUMBER_OF_ENTRIES     -1

// Prototype
si4     encrypt_records(FILE_PROCESSING_STRUCT *fps);
```

Encrypts records if currently decrypted to the level specified in the record header. It marks encrypted records as encrypted (positive of encryption level) in the record header. If the number of records is known (stored in the universal header **number_of_entries** field), this value is used, if this is set to UNKNOWN_NUMBER_OF_ENTRIES the function will work as long as the FILE_PROCESSING_STRUCT's raw_data_bytes field reflects an integral number of records.

The function also applies or removes the recording time offset to the times in the record headers according to the value of the the MEF_global recording_time_offset_mode.

It returns zero on success.

FUNCTION: extract_path_parts()

```
// Prototype
extract_path_parts(si1 *full_file_name, si1 *path, si1 *name, si1 *extension);
```

Non-destructively copies the path (**full_file_name** string up to enclosing directory) into **path** (if not NULL), the name (last component in full_file_name) into **name** (if not NULL), and the extension (last component in full_file_name after a ".") into **extension** (if not NULL). Pass NULL for any components that are not needed. Terminal forward slashes (“/“) are removed. the path is prepended with the current working directory if the **full_file_name** does not begin from root. The function returns zero on success.

example:

```
SESSION        session;
si1            *passed_session_directory = “/Data/Session_1.mefd”

extract_path_and_name(passed_session_directory, session_path, session_name, session_extension);
```

On return, session_path contains “/Data”, session._name contains “Session_1”, and extension contains “mefd”. if only the name was required the following call would suffice:

```
extract_path_and_name(passed_session_directory, NULL, session_name, NULL);
```

FUNCTION: extract_terminal_password_bytes()

```
// Prototype
si4    extract_terminal_password_bytes(si1 *password, si1 *password_bytes);
```

UTF-8 passwords can contain up to 4 bytes per character. In UTF-8 encoding, the most unique byte in each character is the terminal byte. This function extracts those bytes from the UTF-8 password (passed in **password**) to password_bytes, which is used to generate the encryption key for the AES algorithms. Unused bytes are zeroed. This function is called by process_password_data().

FUNCTION: fill_empty_password_bytes()

```
// Prototype
void   fill_empty_password_bytes(si1 *password_bytes);
```

Zero-value bytes at end of the password_bytes array are replaced with **replicable** pseudo-random values generated by the included MEF library function random_byte(). This function is not currently used in the library, but can be used to strengthen weak passwords, although as MEF is open source, the determined hacker could overcome this measure.

(inspired by the password “x” :)

FUNCTION: find_discontinuity_indices()

```
// Prototype
si8	*find_discontinuity_indices(TIME_SERIES_INDEX *tsi, si8 num_disconts, si8
	number_of_blocks);
```

Allocates and returns an array of indices into a TIME_SERIES_INDEX array where discontinuities occur. This can be useful in processing data where crossing discontinuity boundaries is not desirable. It is the calling function's responsibility to free this array.

FUNCTION: find_discontinuity_samples()

```
// Prototype
si8	*find_discontinuity_samples(TIME_SERIES_INDEX *tsi, si8 num_disconts, si8
	number_of_blocks, si1 add_tail);
```

Allocates and returns an array of sample numbers within a segment where discontinuities occur. This can be useful in processing data where crossing discontinuity boundaries is not desirable. It is the calling function's responsibility to free this array. If add_tail is set to MEF_TRUE, the final entry in the array will be the total number of samples in the segment. This can be useful for developing clean loops needing to know the number of samples in a contiguous segment.

FUNCTION: force_behavior()

```
// Constants
#define RESTORE_BEHAVIOR	-1

// Prototype
void	force_behavior(si4 behavior);
```

Changes MEF_globals value of behavior_on_fail and stores original value for restoration in a subsequent call.

THIS ROUTINE IS NOT THREAD SAFE: USE ONLY IN SINGLE THREADED APPLICATIONS.

example: force RETURN_ON_FAIL for a function call, and then restore original value

```
force_behavior(RETURN_ON_FAIL);
function_whose_failure_can_be_handled();
force_behavior(RESTORE_BEHAVIOR);
```

FUNCTION: fps_close()

```
// Prototype
void    fps_close(FILE_PROCESSING_STRUCT *fps);
```

Closes the file associated with the FPS's FILE pointer and sets it to NULL. It also sets the FPS's file descriptor to -1 (closed file).

FUNCTION: fps_lock()

```
// Constants
#define FPS_NO_LOCK_TYPE                          ~(F_RDLCK | F_WRLCK | F_UNLCK)
                                                  // from <fcntl.h>
#define FPS_NO_LOCK_MODE                          0
#define FPS_READ_LOCK_ON_READ_OPEN                1
#define FPS_WRITE_LOCK_ON_READ_OPEN               2
#define FPS_WRITE_LOCK_ON_WRITE_OPEN              4
#define FPS_WRITE_LOCK_ON_READ_WRITE_OPEN         8
#define FPS_READ_LOCK_ON_READ                     16
#define FPS_WRITE_LOCK_ON_WRITE                   32

// Prototype
si4     fps_lock(FILE_PROCESSING_STRUCT *fps, si4 lock_type, const si1 *function, si4 line,
        ui4 behavior_on_fail);
```

Sets an *advisory lock* on the file specified by the FPS directive's lock_mode. The lock is set in blocking mode (i.e. it waits until a lock can be obtained). **lock_type** specifies either a read or write lock. The function & line arguments are provided to know where the function was called from in the case of failure.

FUNCTION: fps_open()

```
// Constants
#define FPS_NO_OPEN_MODE                          0
#define FPS_R_OPEN_MODE                           1
#define FPS_R_PLUS_OPEN_MODE                      2
#define FPS_W_OPEN_MODE                           4
#define FPS_W_PLUS_OPEN_MODE                      8
#define FPS_A_OPEN_MODE                           16
#define FPS_A_PLUS_OPEN_MODE                      32
#define FPS_GENERIC_READ_OPEN_MODE                (FPS_R_OPEN_MODE |
                                                  FPS_R_PLUS_OPEN_MODE |
                                                  FPS_W_PLUS_OPEN_MODE |
                                                  FPS_A_PLUS_OPEN_MODE)
#define FPS_GENERIC_WRITE_OPEN_MODE               (FPS_R_PLUS_OPEN_MODE |
                                                  FPS_W_OPEN_MODE |
                                                  FPS_W_PLUS_OPEN_MODE |
```

```
                                                  FPS_A_OPEN_MODE |
                                                  FPS_A_PLUS_OPEN_MODE)

// Prototype
si4     fps_open(FILE_PROCESSING_STRUCT *fps, const si1 *function, si4 line,
        ui4 behavior_on_fail);
```

Opens the file specified by the FPS according to the FPS directive open_mode. If the mode permits file creation, the file will be created. If higher level directories are needed to open the file in the specified location, they too are created. Once open, the file is optionally locked according to the FPS directive's lock_mode. The file descriptor and file length are also updated.

FUNCTION: fps_read()

```
// Prototype
si4     fps_read(FILE_PROCESSING_STRUCT *fps, const si1 *function, si4 line,
        ui4 behavior_on_fail);
```

Reads bytes specified by the FPS directive's io_bytes (or more commonly the full file if this is set to FPS_FULL_FILE). If lock_on_read is specified in the FPS directive's lock_mode, the file will be locked prior to the read and unlocked after the read.

FUNCTION: fps_unlock()

```
// Prototype
si4     fps_unlock(FILE_PROCESSING_STRUCT *fps, const si1 *function, si4 line,
        ui4 behavior_on_fail);
```

Releases the *advisory lock* on the file specified by the FPS. The function & line arguments are provided to know where the function was called from in the case of failure.

FUNCTION: fps_write()

```
// Prototype
si4     fps_write(FILE_PROCESSING_STRUCT *fps, const si1 *function, si4 line,
        ui4 behavior_on_fail);
```

Writes bytes specified by the FPS directive's io_bytes (or more commonly the full file if this is set to FPS_FULL_FILE). If lock_on_write is the specified in the FPS directive's lock_mode, the file will be locked prior to the write and unlocked after the write. The file descriptor and file length are also updated.

FUNCTION: free_channel()

```
// Prototype
void    free_channel(CHANNEL *channel, si4 free_channel_structure);
```

Frees all the memory pointed to by a CHANNEL structure including all memory associated with SEGMENT structures within it. if free_channel_structure is set to MEF_TRUE, the passed CHANNEL structure will itself be freed also.

FUNCTION: free_file_processing_struct()

```
// Prototype
void    free_file_processing_struct(FILE_PROCESSING_STRUCT *fps);
```

Frees a FILE_PROCESSING_STRUCT's raw_data buffer if not NULL, and then frees the FILE_PROCESSING_STRUCT. It also closes the FILE pointer, if it is open and the close_file directive is set to MEF_TRUE. If the free_password_data directive is set to MEF_TRUE, the FILE_PROCESSING_STRUCT's password_data will be freed.

FUNCTION: free_segment()

```
// Prototype
void    free_segment(SEGMENT *segment, si4 free_segment_structure);
```

Frees all the memory pointed to by a SEGMENT structure. if free_segment_structure is set to MEF_TRUE, the passed SEGMENT structure will itself be freed also.

FUNCTION: free_session()

```
// Prototype
void    free_session(SESSION *session, si4 free_session_structure);
```

Frees all the memory pointed to by a SESSION structure including all memory associated with CHANNEL structures within it, and the SEGMENT structures within them. If free_session_structure is set to MEF_TRUE, the passed SESSION structure will itself be freed also.

FUNCTION: generate_file_list()

```
// Prototype
si1     **generate_file_list(si1 **file_list, si4 *num_files, si1 *enclosing_directory, si1
        *extension)
```

Creates a list of files in the enclosing_directory with the specified extension. If file_list is not NULL, it is presumed to be allocated, otherwise it will be allocated and it is the calling function's responsibility to free it. The function can also be used to generate a list of directories with a specified extension. The number of files or directories in the list is returned in num_files.

FUNCTION: generate_hex_string()

```
// Prototype
si1     *generate_hex_string(ui1 *bytes, si4 num_bytes, si1 *string);
```

Creates a hexadecimal string from "num_bytes" of the bytes in "bytes" into the string pointed to by "string". If string is NULL, it will be allocated. The length of the string required is: (num_bytes + 1) * 3. This is conveniently generated by the macro HEX_STRING_BYTES().

example 1:

```
ui1     hex_str[HEX_STRING_BYTES(ENCRYPTION_KEY_BYTES)];

generate_hex_string(pwd->level_1_encryption_key, ENCRYPTION_KEY_BYTES, hex_str);
printf("Level 1 Encryption Key: %s\n", hex_str);
```

example 2:

```
ui1     *hex_str;

hex_str = generate_hex_string(pwd->level_1_encryption_key, ENCRYPTION_KEY_BYTES, NULL);
printf("Level 1 Encryption Key: %s\n", hex_str);
free(hex_str);
```

FUNCTION: generate_recording_time_offset()

```
// Constants
#define USE_SYSTEM_TIME       -1
#define MAXIMUM_GMT_OFFSET    86400
#define MINIMUM_GMT_OFFSET    -86400

// Prototype
si8    generate_recording_time_offset(si8 recording_start_time_uutc, si4 GMT_offset);
```

The function calculates the recording time offset from the passed recording_start_time_uutc and GMT_offset. The result is stored in the MEF_globals variables recording_time_offset and GMT_offset, respectively. If recording_start_time_uutc equals USE_SYSTEM_TIME, the recording time offset and GMT will be obtained from the system settings, and recording is assumed to start at the time of the function call.

The GMT offset is the number of seconds (not hours) the recording time zone is offset from GMT at the time of recording start. Its range is MINIMUM_GMT_OFFSET to MAXIMUM_GMT_OFFSET. If GMT_offset is outside this range, it's value will be set to zero, and an error will be generated.

The function returns the recording time offset.

example:

```
#define CST_OFFSET_HOURS      -6

generate_recording_time_offset(recording_start_time, CST_OFFSET_HOURS * 3600);
```

FUNCTION: generate_segment_name()

```
// Prototype
si1    *generate_segment_name(FILE_PROCESSING_STRUCT *fps, si1 *segment_name);
```

A simple convenience function to generate the segment name from the channel name and segment number in the FPS's universal header. The result is stored in **segment_name** if it is not NULL. The result is allocated and returned otherwise. If allocated, the calling function is responsible for freeing it.

FUNCTION: generate_UUID()

```
// Prototype
ui1    *generate_UUID(ui1 *uuid);
```

Assigns 16 random bytes to the passed uuid buffer. The possibility of 16 zero bytes is excluded as this is the NO_ENTRY value for UUIDs. The result is stored in **uuid** if it is not NULL. The result is allocated and returned otherwise. If allocated, the calling function is responsible for freeing it.

example:

```
generate_UUID(universal_header->level_UUID);
```

FUNCTION: initialize_file_processing_directives()

```
// File Processing Directives defaults
#define FPS_DIRECTIVE_CLOSE_FILE_DEFAULT                    MEF_TRUE
#define FPS_DIRECTIVE_FREE_PASSWORD_DATA_DEFAULT            MEF_FALSE
#define FPS_DIRECTIVE_LOCK_MODE_DEFAULT                     (FPS_READ_LOCK_ON_READ_OPEN |
                                                            FPS_WRITE_LOCK_ON_WRITE_OPEN |
                                                            FPS_WRITE_LOCK_ON_READ_WRITE_OPEN)
#define FPS_DIRECTIVE_OPEN_MODE_DEFAULT                     FPS_NO_OPEN_MODE
#define FPS_DIRECTIVE_IO_BYTES_DEFAULT                      FPS_FULL_FILE
#define FPS_DIRECTIVE_UPDATE_DEPENDENT_FILES_DEFAULT        MEF_FALSE

// Prototype
FILE_PROCESSING_DIRECTIVES *initialize_file_processing_directives(
        FILE_PROCESSING_DIRECTIVES *directives);
```

If NULL is passed a FILE_PROCESSING_DIRECTIVES structure is allocated and it's pointer returned. In either case, the fields of the structure are set to their default values.

FUNCTION: initialize_MEF_globals()

```
// Prototype
void    initialize_MEF_globals();
```

The MEF_GLOBALS are allocated to the global heap and initialized to their default values. A global pointer to the MEF_GLOBALS structure is set whose name is "MEF_globals". These globals are used by many functions in the library. It includes boolean fields stating whether structure alignment has been confirmed, lookup tables for CRC calculation, UTF8 printing, AES encryption, and SHA hash functions, the session recording time offset and GMT offset, and a verbose flag which if set will cause many library functions to show the output of their processing.

If the global pointer MEF_globals is NULL, a MEF_GLOBALS structure will be allocated on the application heap and the MEF_globals pointer set to its address. If the MEF_globals pointer is not NULL the function will simply reset all the global values to their defaults. This function is called by initialize_meflib(), and so is rarely called explicitly.

example:

```
extern MEF_GLOBALS     *MEF_globals;

initialize_MEF_globals();
```

FUNCTION: initialize_meflib()

```
// Prototype
si4     initialize_meflib();
```

Initializes MEF_globals to default values (if the MEF_globals pointer is NULL, which it is at the launch of the library), checks CPU endianness, checks MEF structure alignments, seeds the random number generator with the current time, sets the file creation umask, and a loads the CRC, UTF8, AES, and SHA lookup tables into the global heap (not stack). Returns MEF_TRUE if all structures are aligned, MEF_FALSE if not. The function currently exits if the cpu endianness is not little endian. This can be changed if there is a demand for big endian processing going forward.

example 1:

```
if (initialize_meflib() == MEF_FALSE) {
        fprintf(stderr, “error initializing meflib => exiting\n”)
        exit(1);
}
MEF_globals->verbose = MEF_TRUE;      // globals initialized by initialize_meflib(),
                                      // default verbose setting is MEF_FALSE
```

example 2:

```
initialize_MEF_globals();             // globals will not be initialized by initialize_meflib()
MEF_globals->verbose = MEF_TRUE;      // default verbose setting is MEF_FALSE
initialize_meflib();
```

This example initializes MEF_globals to their default values. It then sets verbose to MEF_TRUE. Because MEF_globals is not NULL, initialize_meflib() will not call initialize_MEF_globals(), allowing verbose output of initialization routines, and preserving any other non-default global setting changes that were made.

FUNCTION: initialize_metadata()

```
// Prototype
METADATA *initialize_metadata(METADATA *md);
```

The function sets all fields in a METADATA structure to their NO_ENTRY values. No encryption is performed. Section 2 fields are set according to the FPS/s file_type_code.

example:

```
(void) initialize_metadata(metadata_fps);
```

FUNCTION: initialize_universal_header()

```
// Prototype
si4     initialize_universal_header(FILE_PROCESSING_STRUCT *fps, si1 generate_level_UUID, si1
        generate_file_UUID, si1 originating_file);

// Universal Header Structure
typedef struct {
        ui4     header_CRC;
        ui4     body_CRC;
        si1     file_type_string[TYPE_BYTES];
        ui1     mef_version_major;
        ui1     mef_version_minor;
        ui1     byte_order_code;
        si8     start_time;
        si8     end_time;
        si8     number_of_entries;
        si8     maximum_entry_size;
        si4     segment_number;
        si1     channel_name[MEF_BASE_FILE_NAME_BYTES]; // utf8[63], base name only, no extension
        si1     session_name[MEF_BASE_FILE_NAME_BYTES]; // utf8[63], base name only, no extension
        si1     anonymized_name[UNIVERSAL_HEADER_ANONYMIZED_NAME_BYTES]; // utf8[63]
        ui1     level_UUID[UUID_BYTES];
        ui1     file_UUID[UUID_BYTES];
        ui1     provenance_UUID[UUID_BYTES];
        ui1     level_1_password_validation_field[PASSWORD_VALIDATION_FIELD_BYTES];
        ui1     level_2_password_validation_field[PASSWORD_VALIDATION_FIELD_BYTES];
        ui1     protected_region[UNIVERSAL_HEADER_PROTECTED_REGION_BYTES];
        ui1     discretionary_region[UNIVERSAL_HEADER_DISCRETIONARY_REGION_BYTES];
} UNIVERSAL_HEADER;

// Constants
#define NO_UUID       0
```

The function sets universal header fields to default values. It will generate the appropriate UUIDs if **generate_level_UUID** or **generate_file_UUID** are set to MEF_TRUE. If **originating_file** is set to MEF_TRUE, the provenance_UUID will be set to the value of the file_UUID. It fills in the current library's MEF version and endianness.

example:

```
initialize_universal_header(generic_uh, MEF_TRUE, MEF_FALSE, MEF_FALSE);
```

Initializes a generic universal header with a level UUID, but no file or provenance UUIDs.

FUNCTION: local_date_time_string()

```
// Prototype
void	local_date_time_string(si8 uutc_time, si1 *time_str);
```

Returns a string with local date and time from a UUTC time. **32 bytes are required** for this string. If NULL is passed for the string, it will be allocated and the pointer to the string returned, it is the calling function's responsibility to free this memory.

If the recording time offset is applied and the value is set in MEF_globals, this will be added to the UUTC time. The GMT offset is also applied to obtain the local time. Note that if a recording time offset is applied, but the reader has no access to these values, the global values of both will be zero (set in initialize_MEF_globals()). And so the time returned will be the true local time of day at the time of recording, but with recording beginning on Jan 1, 1970 in GMT.

If recording time offset is not applied, its global value is zero, but **the GMT offset is still required to know the local time of day**. Only if the global GMT offset is set correctly can the function will return the correct local time of day.

example:

```
si1	time_str[32]; // all 32 bytes are required

local_date_time_string(universal_header->start_time, time_str);
```

FUNCTION: MEF_pad()

```
// Prototype
si8	MEF_pad(ui1 *buffer, si8 content_len, ui4 alignment);
```

Fills buffer beyond content_len (in bytes) with PAD_BYTE_VALUE, to next boundary determined by alignment. Returns (content_len + pad_bytes).

FUNCTION: MEF_snprintf()

```
// Prototype
void	MEF_snprintf(si1 *target, si4 target_field_bytes, si1 *format, …);
```

A version of snprintf() that zeros the unused bytes in the target field. Called as standard sprintf() with the extra parameter target_field_bytes that specifies the length of the field being written to. MEF strings are zeroed in unused bytes to facilitate identical CRCs in files with identical information content.

example:

```
MEF_snprintf(full_file_name, MEF_FULL_FILE_NAME_BYTES, "%s/%s.%s", session->path, session->name,
SESSION_DIRECTORY_TYPE_STRING);
```

Prints full path to session directory into full_file_name field. Zeros unused bytes.

FUNCTION: MEF_sprintf()

```
// Prototype
void    MEF_sprintf(si1 *target, si4 target_field_bytes, si1 *format, …);
```

A version of sprintf() that returns the number of characters copied including the terminating zero. Useful in calculating pad-bytes needed in record data fields. See MEF_pad();

FUNCTION: MEF_strcat()

```
// Prototype
si4     MEF_strcat(si1 *target_string, si1 *source_string);
```

A version of strcat() that returns the number of characters in the concatenated string including the terminating zero. Useful in calculating pad-bytes needed in record data fields. See MEF_pad();

FUNCTION: MEF_strcpy()

```
// Prototype
si4     MEF_strcpy(si1 *target_string, si1 *source_string);
```

A version of strcpy() that returns the number of characters copied including the terminating zero. Useful in calculating pad-bytes needed in record data fields. See MEF_pad();

FUNCTION: MEF_strncat()

```
// Prototype
void   MEF_strncat(si1 *target_string, si1 *source_string, si4 target_field_bytes);
```

A version of strcat() that zeros the unused bytes in the target field. Called as standard strncat(). MEF strings are zeroed in unused bytes to facilitate identical CRCs in files with identical information content.

FUNCTION: MEF_strncpy()

```
// Prototype
void   MEF_strncpy(si1 *target_string, si1 *source_string, si4 target_field_bytes);
```

A version of strncpy() that zeros the unused bytes in the target field. Called as standard strncpy(). MEF strings are zeroed in unused bytes to facilitate identical CRCs in files with identical information content.

example:

```
MEF_strncpy(metadata_fps->universal_header->channel_name, channel_name,
MEF_BASE_FILE_NAME_BYTES);
```

Copy channel_name into universal header channel_name field, zeroing unused bytes.

FUNCTION: numerical_fixed_width_string()

```
// Prototype
si1    *numerical_fixed_width_string(si1 *string, si4 string_bytes, si4 number);
```

Writes into **string**, string_bytes total digits, including prepended zeroes, the value of number. String must be able to accommodate (string_bytes + 1) bytes. If string is NULL, it will be allocated and the pointer to it will be returned. The calling function is responsible for freeing this memory.

example:

```
si4    seg_number = 2;
si1    seg_number_string[FILE_NUMBERING_DIGITS + 1];

(void) numerical_fixed_width_string(seg_number_string, FILE_NUMBERING_DIGITS, seg_number);
MEF_sprintf(segment.name, MEF_SEGMENT_BASE_FILE_NAME_BYTES, "%s-%s", channel.name,
seg_number_string);
```

This will print the segment name into the segment.name field. The segment name is defined to be the channel name followed by a hyphen and a 6 digit (zero-prepended) version of it's segment number (in this case, 2).

FUNCTION: offset_time_series_index_times()

```
// Prototype
si4     offset_time_series_index_times(FILE_PROCESSING_STRUCT *fps, si4 action);

// Constants
#define RTO_INPUT_ACTION      1
#define RTO_OUTPUT_ACTION     2
```

This function applies the recording time offset to the array of time series indices according to the global recording_time_offset_mode and whether the operation is being done as input or output. This is specified in the passed **action** parameter, with either the constant RTO_INPUT_ACTION or RTO_OUTPUT_ACTION. This function is called by read_MEF_file() and write_MEF_file() and usually needn't be called explicitly.

FUNCTION: offset_video_index_times()

```
// Prototype
si4     offset_video_index_times(FILE_PROCESSING_STRUCT *fps, si4 action);

// Constants
#define RTO_INPUT_ACTION      1
#define RTO_OUTPUT_ACTION     2
```

This function applies the recording time offset to the array of video indices according to the global recording_time_offset_mode and whether the operation is being done as input or output. This is specified in the passed **action** parameter, with either the constant RTO_INPUT_ACTION or RTO_OUTPUT_ACTION. This function is called by read_MEF_file() and write_MEF_file() and usually needn't be called explicitly.

FUNCTION: process_password_data()

```
// Prototype
PASSWORD_DATA  *process_password_data(si1 *unspecified_password, si1 *level_1_password, si1
        *level_2_password, UNIVERSAL_HEADER *universal_header);

// Structures
typedef struct {
        ui1     level_1_encryption_key[ENCRYPTION_KEY_BYTES];
        ui1     level_2_encryption_key[ENCRYPTION_KEY_BYTES];
```

```
        ui1     access_level;
} PASSWORD_DATA;
```

Allocates a PASSWORD_DATA structure and fills it.

If an unspecified_password is passed, the function will determine whether the password is a level 1 or level 2 password and set the access_level of the PASSWORD_DATA structure accordingly via the password_validation_fields in the passed universal header. Appropriate decryption keys are generated and put into the PASSWORD_DATA structure. This is generally used for reading MEF files.

If a level_1_password or level_2_password is passed, the password validation fields will be generated into the passed universal_header structure. The access_level of the PASSWORD_DATA structure will be set according to whether a level_1 or level_2 password was passed. Appropriate encryption keys are generated and put into the PASSWORD_DATA structure. This is generally used for writing new MEF files. *Note that for level 2 access, a level 1 password must be passed, even if level 1 encryption is never used in the new MEF files.*

example 1:

```
fps->password_data = process_password_data(password, NULL, NULL, fps->universal_header);
```

Processes an unspecified password for reading by validating against the password validation fields in the universal_header. Depending on the access level, a level 1 or both a level 1 and level 2 decryption keys are generated into their appropriate fields in the PASSWORD_DATA structure. A PASSWORD_DATA structure pointer is returned.

example 2:

```
fps->password_data = process_password_data(NULL, level_1_password, level_2_password,
universal_header);
```

In writing a new MEF file, a level 1 and level 2 password are passed, and their password validation fields are written into the universal header. Both level 1 and level 2 encryption keys are generated into their appropriate fields in the PASSWORD_DATA structure.

FUNCTION: proportion_filt()

```
// Prototype
void    proportion_filt(sf8 *x, sf8 *px, si8 len, sf8 prop, si4 span);
```

Performs a sliding widow proportion filter from input array x to output array px of length len. if px is NULL it will be allocated, and responibility for freeing this memory falls to the calling function. The span is the window width in points and will be made odd if it is not.

The prop parameter varies between 0.0 and 1.0. A value of 0.0 in a local minimum filter, 0.5 is a median filter, and 1.0 is a local maximum filter. All other values in the range are valid, so a value of 0.75 would give a filter of the local 75th percentile value in the window.

FUNCTION: random_byte()

```
// Prototype
ui1    random_byte(ui4 *m_w, ui4 *m_z);
```

Returns a pseudorandom byte. Used by fill_empty_password_bytes(). Pseudorandom number generator code is contained within the function (i.e. system random number generator is not used) so that values are replicable across systems. This function is available, but not currently used in any of the other library functions.

FUNCTION: read_MEF_channel()

```
// Prototype
CHANNEL *read_MEF_channel(CHANNEL *channel, si1 *chan_path, si4 channel_type, si1 *password,
       PASSWORD_DATA *password_data, si1 read_time_series_data, si1 read_record_data)

// Channel Types
#define UNKNOWN_CHANNEL_TYPE         -1
#define TIME_SERIES_CHANNEL_TYPE     1
#define VIDEO_CHANNEL_TYPE           2

// Structures
typedef struct {
       si4                     channel_type;
       METADATA                metadata;
       FILE_PROCESSING_STRUCT *record_data_fps;
       FILE_PROCESSING_STRUCT *record_indices_fps;
       si8                     number_of_segments;
       SEGMENT                 *segments;
       si1                     path[MEF_FULL_FILE_NAME_BYTES];  // full path to enclosing
                                                                // directory
       si1                     name[MEF_BASE_FILE_NAME_BYTES];  // just base name, no extension
       si1                     extension[TYPE_BYTES];  // channel directory extension
       si1                     session_name[MEF_BASE_FILE_NAME_BYTES];  // base name, no extension
       ui1                     level_UUID[UUID_BYTES];
       si1                     anonymized_name[UNIVERSAL_HEADER_ANONYMIZED_NAME_BYTES];
       si8                     maximum_number_of_records;
       si8                     maximum_record_bytes;
       si8                     earliest_start_time;
       si8                     latest_end_time;
} CHANNEL;
```

This function will read the channel pointed to by chan_path (full path to the channel directory) and fill in the the fields in the CHANNEL structure. If a channel structure is not passed (NULL passed), one will be allocated. Either a password, or PASSWORD_DATA structure should be passed to read encrypted fields. In the case that no data is encrypted or only unencrypted data is needed, NULL can be passed for both fields. If the read_time_series_data flag is set to MEF_TRUE in the passed directives, each time series segment's data will be read into its SEGMENT structure's time_series_data_fps raw_data field after the segment data's universal header; otherwise only the segment data's universal header will be read into this field. If directives are NULL, default directives will be used.

The session_name and level_UUID fields are filled in, but are redundant with the universal header information in the record data and record indices files, if present. These fields are included in this structure because they are useful in functions that write new channel files.

If read_record_data is set to MEF_TRUE, the all record data will be read into the appropriate structure's record_data_fps raw_data field after the record data's universal header and the file will be closed; otherwise only the record data's universal header will be read into this field and the file will be left open with the file pointer pointing to the next byte after the universal header.

The metadata structure is the same as those contained in a segment FPS, but is not part of an FPS. It contains summary information of the segment metadata files. Fields whose values vary across segments and whose value cannot be expressed as a maximum, etc. are filled with their NO_ENTRY values.

The passed channel_type parameter is used to determine the metadata type to expect. If UNKNOWN_CHANNEL_TYPE is passed, the channel type is determined from the channel path name by calling channel_type_from_path(). The channel_type is stored in the CHANNEL structure.

The CHANNEL structure also keeps track of other metadata derived from universal headers and processing.

```
Other CHANNEL structure fields
number_of_segments          Number of segments in the channel
segments                    Pointer to an array of SEGMENT structures
path                        Full path to enclosing channel directory
name                        Base channel name, no extension
extension                   Channel directory extension
anonymized_name             This value, zeros if varies across segments
maximum_number_of_records   Maximum of this value across segments
maximum_record_bytes        Maximum of this value across segments
earliest_start_time         Minimum of the absolute value of this across segments
latest_end_time             Maximum of the absolute value of this across segments
```

The function returns a pointer to the CHANNEL structure.

example:

```
(void) read_MEF_channel(&session->channels[i], full_file_name, TIME_SERIES_CHANNEL_TYPE, NULL,
       password_data, MEF_FALSE, MEF_FALSE);
```

This call reads the channel specified by full_file_name into a preallocated CHANNEL structure. A PASSWORD_DATA data structure is passed, so a password is not required. The read_time_series_data and read_recod_data flags are set to MEF_FALSE, so only the universal headers will be read in from these files.

FUNCTION: read_MEF_file()

```
// Prototype
FILE_PROCESSING_STRUCT *read_MEF_file(FILE_PROCESSING_STRUCT *fps, si1 *file_name, si1 *password,
       PASSWORD_DATA *password_data, FILE_PROCESSING_DIRECTIVES *directives, ui4
       behavior_on_fail)

// Structures
typedef struct {
       si1                          full_file_name[MEF_FULL_FILE_NAME_BYTES];  // full path
       FILE                         *fp;    // FILE pointer
       si4                          fd;     // FILE descriptor
       si8                          file_length;
       ui4                          file_type_code;
       UNIVERSAL_HEADER             *universal_header;
       FILE_PROCESSING_DIRECTIVES   directives;
       PASSWORD_DATA                *password_data;
       METADATA                     metadata;
       TIME_SERIES_INDEX            *time_series_indices;
       VIDEO_INDEX                  *video_indices;
       ui1                          *records;
       RECORD_INDEX                 *record_indices;
       ui1                          *RED_blocks;
       si8                          raw_data_bytes;
       ui1                          *raw_data;
} FILE_PROCESSING_STRUCT;

typedef struct {
       si1                          close_file;
       si1                          free_password_data;  // when freeing FPS
       si8                          io_bytes;  // bytes to read or write
       ui4                          lock_mode;
       ui4                          open_mode;
} FILE_PROCESSING_DIRECTIVES;

// Constants
#define FPS_FULL_FILE         -1
```

The function reads any MEF file type, identified by its full path in file_name, into a FILE_PROCESSING_STRUCT (FPS). If NULL is passed for the FPS one will be allocated. If an FPS is allocated, and the passed directives are not NULL, they will be used. If the FPS's full_file_name field is NULL the passed file_name will be copied into this field. The file will be opened if it is not already open. If the close_file directive is set to MEF_FALSE, the file will be left open, otherwise it will be closed after reading. If the io_bytes parameter is set to FPS_FULL_FILE the whole file will be read, otherwise only io_bytes bytes will be read.

The data are read into the raw_data field of the FPS. The FPS's universal_header pointer is set to point to the beginning of the raw data. The appropriate file type's structure pointer in the FPS is set to point to the raw data after the universal header.

If password_data is NULL, the function will process the passed password as an unspecified password and generate password_data. Otherwise password_data will be assigned to that field in the FPS.

Read_MEF_file() validates file CRCs according to the global CRC_mode. It the decrypt encrypted data to the access level allowed by the password data. It then offsets times according to the global recording_time_offset_mode.

The function returns a pointer to a FILE_PROCESSING_STRUCT or NULL if unsuccessful.

example 1:

```
segment->time_series_data_fps = read_MEF_file(NULL, full_file_name, NULL, password_data, NULL,
USE_GLOBAL_BEHAVIOR);
```

Reads the time series data file pointed to by full_file_name. Read_MEF_file() allocates and returns a pointer to the FPS. A PASSWORD_DATA structure is supplied, so password is not processed, and need not be passed. All data are read in as the io_bytes default is FPS_FULL_FILE. The file is closed after reading as the close_file directive's default is MEF_TRUE.

example 2:

```
si1     *password = “password”;

segment->time_series_data_fps = allocate_file_processing_struct(0,
        TIME_SERIES_DATA_FILE_TYPE_CODE, NULL, 0);
segment->time_series_data_fps->directives.io_bytes = UNIVERSAL_HEADER_BYTES
segment->time_series_data_fps->directives.close_file = MEF_FALSE; // default is MEF_TRUE
(void) read_MEF_file(segment->segment_data_fps, full_file_name, password, NULL, 0);
```

Reads the time series data file pointed to by full_file_name. Read_MEF_file() does not allocate the FPS since one is passed. Preallocation was done to change the default values of the directives; in this case to read just the universal header and leave the file open with the file pointer pointing to the next byte after the universal header. A PASSWORD_DATA structure was not supplied, so password is processed as an unspecified password, and a PASSWORD_DATA structure is created for the FPS. If passwords are preserved across MEF files - the returned PASSWORD_DATA can be passed in future calls to read_MEF_file().

FUNCTION: read_MEF_segment()

```
// Prototype
SEGMENT *read_MEF_segment(SEGMENT *segment, si1 *seg_path, si4 channel_type, si1 *password,
        PASSWORD_DATA *password_data, si1 read_time_series_data, si1 read_record_data)

// Structure
typedef struct {
        si4                             channel_type;
        FILE_PROCESSING_STRUCT          *metadata_fps;
        FILE_PROCESSING_STRUCT          *time_series_data_fps;
        FILE_PROCESSING_STRUCT          *time_series_indices_fps;
        FILE_PROCESSING_STRUCT          *video_indices_fps;
        FILE_PROCESSING_STRUCT          *record_data_fps;
        FILE_PROCESSING_STRUCT          *record_indices_fps;
        si1                             name[MEF_SEGMENT_BASE_FILE_NAME_BYTES];  // base name
        si1                             path[MEF_FULL_FILE_NAME_BYTES];  // full path to enclosing
                                                                         //directory (channel dir)
        si1                             channel_name[MEF_BASE_FILE_NAME_BYTES];  // base name
        si1                             session_name[MEF_BASE_FILE_NAME_BYTES];  // base name
        ui1                             level_UUID[UUID_BYTES];
} SEGMENT;
```

This function will read the segment pointed to by seg_path (full path to the segment directory) and fill in the the fields in the SEGMENT structure. If a segment structure is not passed (NULL passed), one will be allocated. Either an unspecified password, or PASSWORD_DATA structure should be passed to read encrypted fields. In the case that no data is encrypted or only unencrypted data is needed, NULL can be passed for both fields.

The passed channel_type parameter is used to determine the metadata type to expect. If UNKNOWN_CHANNEL_TYPE is passed, the channel type is determined from the channel path name by calling channel_type_from_path(). The channel_type is stored in the CHANNEL structure.

The channel_name, session_name, and level_UUID fields are filled in, but are redundant with the universal header information in each of the files. These fields are included in this structure because they are useful in functions that write new segment files.

If read_time_series_data is set to MEF_TRUE (and it is a time series segment), the time series data will be read into the SEGMENT structure's data_fps raw_data field after the segment data's universal header and the file will be closed; otherwise only the segment data's universal header will be read into this field and the file will be left open with the file pointer pointing to the next byte after the universal header.

If read_record_data is set to MEF_TRUE, the segment's records data will be read into the SEGMENT structure's records_data_fps raw_data field after the records data's universal header and the file will be closed; otherwise only the records data's universal header will be read into this field and the file will be left open with the file pointer pointing to the next byte after the universal header.

The function returns a pointer to the SEGMENT structure.

example:

```
SEGMENT                      *segment;

segment = read_MEF_segment(NULL, full_file_name, TIME_SERIES_CHANNEL_TYPE, NULL, pwd,
        MEF_TRUE, MEF_TRUE);
```

This call will read the all the files of the segment pointed to by full_file_name and allocate and populate a SEGMENT structure. The passed password_data is assigned in the FILE_PROCESSING_STRUCTs. The time series data file is opened, read in full, and closed. Likewise for the segment record data file, if present. This is an uncommon use for large data files as reading all of the data into memory is frequently impractical.

FUNCTION: read_MEF_session()

```
// Prototype
SESSION *read_MEF_session(SESSION *session, si1 *sess_path, si1 *password, PASSWORD_DATA
        *password_data, si1 read_time_series_data, si1 read_record_data);

typedef struct {
        METADATA                time_series_metadata;
        si4                     number_of_time_series_channels;
        CHANNEL                 *time_series_channels;
        METADATA                video_metadata;
        si4                     number_of_video_channels;
        CHANNEL                 *video_channels;
        FILE_PROCESSING_STRUCT *record_data_fps;
        FILE_PROCESSING_STRUCT *record_indices_fps;
        si1                     name[MEF_BASE_FILE_NAME_BYTES];  // just base name, no extension
        si1                     path[MEF_FULL_FILE_NAME_BYTES];  // path to enclosing directory
        si1                     anonymized_name[UNIVERSAL_HEADER_ANONYMIZED_NAME_BYTES];
        ui1                     level_UUID[UUID_BYTES];
        si8                     maximum_number_of_records;
        si8                     maximum_record_bytes;
        si8                     earliest_start_time;
        si8                     latest_end_time;
} SESSION;
```

This function will read all the files associated with the session pointed to by sess_path (full path to the session directory) and fill in the the fields in the SESSION structure. If a SESSION structure is not passed (NULL passed), one will be allocated. Either an unspecified password, or PASSWORD_DATA structure should be passed to read encrypted fields. In the case that no data is encrypted or only unencrypted data is needed, NULL can be passed for both fields.

The level_UUID field is filled in, but is redundant with the universal header information in the record data and indices files, if present. This field is included in this structure because it is useful in functions that write new session files.

If the directive's read_time_series_data flag is set to MEF_TRUE, the segment data will be read into the SEGMENT structure's data_fps raw_data field after the segment data's universal header and the file will be closed; otherwise only the segment data's universal header will be read into this field and the file will be left open with the file pointer pointing to the next byte after the universal header.

If the read_record_data directive is set to MEF_TRUE, the all records data files will be read into the appropriate structure's records_data_fps raw_data field after the records data's universal header and the file will be closed; otherwise only the records data's universal header will be read into this field and the file will be left open with the file pointer pointing to the next byte after the universal header.

The metadata structures are the same as those contained in CHANNEL structures; they are not part of an FPS. It contains summary information of the channel metadata files. Fields whose values vary across channels and whose value cannot be expressed as a maximum, etc. are filled with their NO_ENTRY values.

The SESSION structure also keeps track of other metadata derived from universal headers and processing.

```
Other CHANNEL structure fields
number_of_time_series_channels      Number of time series channels in the session
time_series_channels                Pointer to an array of CHANNEL structures
number_of_video_channels            Number of video channels in the session
video_channels                      Pointer to an array of CHANNEL structures
path                                Full path to enclosing session directory
name                                Base session name, no extension
extension                           Session directory extension
anonymized_name                     This value, zeros if varies across channels
maximum_number_of_records           Maximum of this value across channels
maximum_record_bytes                Maximum of this value across channels
earliest_start_time                 Minimum of the absolute value of this across channels
latest_end_time                     Maximum of the absolute value of this across channels
```

example:

```
SESSION        *session;

session = read_MEF_session(NULL, session_directory, password, NULL, MEF_FALSE, MEF_FALSE);
```

This call will allocate a SESSION structure and read all files associated with a MEF session and fill in the fields of at the SESSION structure and all of its substructures. It will not read the segment data, or record data unless these flags are set in the passed FILE_PROCESSING_DIRECTIVES. The universal headers of those files will be read, and the files will be left open. Their file pointers will be left at the beginning of the data after the universal header. All other files will be read completely into their FILE_PROCESSING_STRUCTs and closed.

FUNCTION: reallocate_file_processing_struct()

```
// Prototype
si4    reallocate_file_processing_struct(FILE_PROCESSING_STRUCT *fps, si8 raw_data_bytes);
```

This function reallocates the raw_data array in a FILE_PROCESSING_STRUCT. The array is increased (or decreased) to the passed raw_data_bytes value. Existing data are preserved, extra bytes are zeroed. The raw_data_bytes field of the FPS is updated and appropriate pointers in the FPS are updated.

FUNCTION: remove_line_noise()

```
// Prototype
si4    remove_line_noise(si4 *data, si8 n_samps, sf8 sampling_frequency, sf8 line_frequency, sf8
*template)
```

AC line noise is removed from the input data array via template subtraction. If *template is not NULL the subtracted template will be returned in that array. If NULL is passed for *template, it will be allocated and freed. The template does not adapt so the function is best used on small chunks of data, such as individual RED blocks prior to compression.

The function remove_line_noise_adaptive() does adaptive filtering and does not return a template. The noise suppression with that function is generally better, but it is slower and does not return a template. The advantage of returning a template is that the template can be stored for each block of data so that if needed the unmodified data can be restored. There is a record type LNPT (line noise template) that was designed for this purpose, and would be stored in segment-level record files.

FUNCTION: remove_line_noise_adaptive()

```
// Prototype
void   remove_line_noise(si4 *data, si8 n_samps, sf8 sampling_frequency, sf8 line_frequency, si4
n_cycles)
```

AC line noise is removed from the input data array via template subtraction. The template adapts at a rate specified by n_cycles.

FUNCTION: remove_recording_time_offset()

```
// Prototype
void    remove_recording_time_offset(si8 *time);
```

The global recording time offset is removed from the passed μUTC time. If the value is positive, it is presumed not to have had the recording time offset applied, and nothing is done. The converse function is apply_recording_time_offset(), described above.

FUNCTION: show_file_processing_struct()

```
// Prototype
void    show_file_processing_struct(FILE_PROCESSING_STRUCT *fps)

// Structures
typedef struct {
        si1                             full_file_name[MEF_FULL_FILE_NAME_BYTES];  // full path
        FILE                            *fp;
        si4                             fd;
        si8                             file_length;
        ui4                             file_type_code;
        UNIVERSAL_HEADER                *universal_header;
        FILE_PROCESSING_DIRECTIVES      directives;
        PASSWORD_DATA                   *password_data;
        METADATA                        metadata;
        TIME_SERIES_INDEX               *time_series_indices;
        VIDEO_INDEX                     *video_indices;
        ui1                             *records;
        RECORD_INDEX                    *record_indices;
        ui1                             *RED_blocks;
        si8                             raw_data_bytes;
        ui1                             *raw_data;
} FILE_PROCESSING_STRUCT;
```

Displays all the elements of a FILE_PROCESSING_STRUCT structure.

FUNCTION: show_metadata()

```
// Prototype
void    show_metadata(FILE_PROCESSING_STRUCT *fps)

// Structures
typedef struct {
        METADATA_SECTION_1                      *section_1;
        TIME_SERIES_METADATA_SECTION_2          *time_series_section_2;
        VIDEO_METADATA_SECTION_2                *video_section_2;
        METADATA_SECTION_3                      *section_3;
} METADATA;
```

Displays all the elements of a METADATA structure of the type specified by the passed FILE_PROCESSING_STRUCT.

FUNCTION: show_password_data()

```
// Prototype
void    show_password_data(FILE_PROCESSING_STRUCT *fps);

// Structures
typedef struct {
        ui1     level_1_encryption_key[ENCRYPTION_KEY_BYTES];
        ui1     level_2_encryption_key[ENCRYPTION_KEY_BYTES];
        ui1     access_level;
} PASSWORD_DATA;
```

Displays all the elements of a PASSWORD_DATA structure.

FUNCTION: show_record()

```
// Prototype
void    show_record(RECORD_HEADER *record_header, ui4 record_number, PASSWORD_DATA *pwd);
```

This function displays the contents of the record pointed to by record_header. If the record needs to be decrypted and the access level is sufficient, the record will be decrypted. Show_record() resides in the mefrec.c file.

FUNCTION: show_records()

```
// Constant
#define UNKNOWN_NUMBER_OF_ENTRIES     -1

// Prototype
void    show_records(FILE_PROCESSING_STRUCT *fps);
```

This function displays the contents of the records data file. If the record needs to be decrypted and the access level is sufficient, the record will be decrypted. Show_records() calls show_record() for each record. Show_record() resides in the mefrec.c file. If the number_of_records is known, this number will be used. Otherwise (i.e. number_of_records == UNKNOWN_NUMBER_OF_ENTRIES) the function will still work, but could fail in the case of an incomplete terminal record.

FUNCTION: show_universal_header()

```
// Prototype
void    show_universal_header(FILE_PROCESSING_STRUCT *fps);

// Structure
typedef struct {
        ui4     file_CRC;
        si1     file_type_string[TYPE_BYTES];
        ui1     mef_version_major;
        ui1     mef_version_minor;
        ui1     byte_order_code;
        ui1     level_1_password_validation_field[PASSWORD_VALIDATION_FIELD_BYTES];
        ui1     level_2_password_validation_field[PASSWORD_VALIDATION_FIELD_BYTES];
        ui1     session_UUID[UUID_BYTES];
        ui1     channel_UUID[UUID_BYTES];
        ui1     segment_UUID[UUID_BYTES];
        ui1     protected_region[UNIVERSAL_HEADER_PROTECTED_REGION_BYTES];
        ui1     discretionary_region[UNIVERSAL_HEADER_DISCRETIONARY_REGION_BYTES];
} UNIVERSAL_HEADER;
```

This function displays the contents of a FILE_PROCESSING_STRUCT's universal_header field.

FUNCTION: write_MEF_file()

```
// Prototype
si4     write_MEF_file(FILE_PROCESSING_STRUCT *fps);
```

The function will write out the file contained in the FILE_PROCESSING_STRUCT. If the file is not yet open, it will be opened. If the file requires encryption it will be encrypted. Times will be offset according to the global recording_time_offset_mode. The file CRCs will be calculated according to the global CRC_mode and the entered into the universal header.

If the io_bytes directive is set to FPS_FULL_FILE, the whole file will be written, otherwise only this number of bytes will be written. If the close_file directive is set to MEF_FALSE, the file will be left open.

```
/**************************************************************************************/
/********************************** FILTER Functions *********************************/
/**************************************************************************************/


// Constants
#define FILT_LOWPASS_TYPE           1
#define FILT_BANDPASS_TYPE          2
#define FILT_HIGHPASS_TYPE          3
#define FILT_BANDSTOP_TYPE          4
#define FILT_TYPE_DEFAULT           FILT_LOWPASS_TYPE
#define FILT_ORDER_DEFAULT          5
#define FILT_MAX_ORDER              10
```

```
#define FILT_BAD_FILTER              -1

// Typedefs & Structures
typedef struct {
        si4     order;
        si4     poles;
        si4     type;
        sf8     sampling_frequency;
        si8     data_length;
        sf8     cutoffs[2];
        sf8     *numerators;
        sf8     *denominators;
        sf8     *initial_conditions;
        si4     *orig_data;
        si4     *filt_data;
        sf8     *sf8_filt_data;
        sf8     *sf8_buffer;
} FILT_PROCESSING_STRUCT;

typedef struct {
        sf16    real;
        sf16    imag;
} FILT_LONG_COMPLEX;

// Prototypes
void                    FILT_balance(sf16 **a, si4 poles);
si4                     FILT_butter(FILT_PROCESSING_STRUCT *filtps);
void                    FILT_complex_divl(FILT_LONG_COMPLEX *a, FILT_LONG_COMPLEX *b,
                        FILT_LONG_COMPLEX *quotient);
void                    FILT_complex_expl(FILT_LONG_COMPLEX *exponent, FILT_LONG_COMPLEX *ans);
void                    FILT_complex_multl(FILT_LONG_COMPLEX *a, FILT_LONG_COMPLEX *b,
                        FILT_LONG_COMPLEX *product);
void                    FILT_elmhes(sf16 **a, si4 poles);
void                    FILT_filtfilt(FILT_PROCESSING_STRUCT *filtps);
void                    FILT_free_processing_struct(FILT_PROCESSING_STRUCT *filtps,
                        si1 free_orig_data, si1 free_filt_data);
FILT_PROCESSING_STRUCT *FILT_initialize_processing_struct(si4 order, si4 type, sf8 samp_freq,
                        si8 data_len, si1 alloc_orig_data, si1 alloc_filt_data,
                        sf8 cutoff_1, ...);
void                    FILT_generate_initial_conditions(FILT_PROCESSING_STRUCT *filtps);
void                    FILT_hqr(sf16 **a, si4 poles, FILT_LONG_COMPLEX *eigs);
void                    FILT_invert_matrix(sf16 **a, sf16 **inv_a, si4 order);
void                    FILT_mat_multl(void *a, void *b, void *product, si4 outer_dim1,
                        si4 inner_dim, si4 outer_dim2);
void                    FILT_unsymmeig(sf16 **a, si4 poles, FILT_LONG_COMPLEX *eigs);
```

The functions in the FILTER section of the library facilitate creation of Butterworth infinite impulse response (IIR) filters and perform zero-phase digital filtering using them. Many or the functions are purely internal to the filtering process, so only the gateway functions will be described here.

FUNCTION: FILT_butter()

```
// Prototype
si4     FILT_butter(FILT_PROCESSING_STRUCT *filtps);
```

This function calculates coefficients for a Butterworth filter of the specified type and returns the poles of the filter (which may be double of the order depending on filter type) in the numerator and denominator fields of the FILT_PROCESSING_STRUCT. These arrays are allocated in FILT_butter().

FUNCTION: FILT_filtfilt()

```
// Prototype
void    FILT_filtfilt(FILT_PROCESSING_STRUCT *filtps)
```

This call non-destructively applies the specified filter to the orig_data (si4) array, and returns the filtered data in the sf8_filt_data array. If the initial_conditions or sf8_buffer arrays are NULL, they will be allocated and freed after use. The initial_conditions will be calculated if they are not passed.

example:

```
FILT_PROCESSING_STRUCT        *filtps;
RED_PROCESSING_STRUCT         *rps;

// set up filter
filtps->order = 5;
filtps->type = FILT_BANDPASS_TYPE;
filtps->sampling_frequency = 32000.0;
filtps->cutoffs[0] = 100.0;
filtps->cutoffs[1] = 200.0;
FILT_butter(filtps);
FILT_generate_initial_conditions(filtps);

// apply the filter
filtps->orig_data = rps->original_data;
filtps->data_length = rps->block_header->number_of_samples;
FILT_filtfilt(filtps);
```

This code snippet applies a bandpass Butterworth filter (100 - 200 Hz band) to the integer (si4) data in the FILT_PROCESSING_STRUCT's orig_data array, returning the results as floating point data (sf8) in the FILT_PROCESSING_STRUCT's sf8_filt_data array. See RED_apply_filter() for a function that does this and fills in the integer data (si4) in the FILT_PROCESSING_STRUCT's filt_data array.

FUNCTION: FILT_free_processing_struct()

```
// Prototype
void    FILT_free_processing_struct(FILT_PROCESSING_STRUCT *filtps, si1 free_orig_data,
```

```
	si1 free_filt_data);
```

This call frees all allocated members of the passed FILT_PROCESSING_STRUCT. Given that the “orig_data” and “filt_data” members of a FILT_PROCESSING_STRUCT are often sub-portions of larger external arrays, these must be freed explicitly with the passed “free_orig_data” and “free_filt_data” flags;

example:

```
FILT_PROCESSING_STRUCT		*filtps;

FILT_free_processing_struct(FILT_PROCESSING_STRUCT *filtps, MEF_TRUE, MEF_FALSE);
```

Frees all allocated arrays in the FILT_PROCESSING_STRUCT as well as the orig_data array. The filt_data array will not be freed.

FUNCTION: FILT_initialize_processing_struct()

```
// Prototype
FILT_PROCESSING_STRUCT *FILT_initialize_processing_struct(si4 order, si4 type, sf8 samp_freq,
                        si8 data_len, si1 alloc_orig_data, si1 alloc_filt_data,
                        sf8 cutoff_1, ...);
```

This function allocates and returns a FILT_PROCESSING_STRUCT pointer. It calculates coefficients for a Butterworth filter of the specified type and returns the poles of the filter (which may be double of the order depending on filter type) in the numerator and denominator fields of the FILT_PROCESSING_STRUCT. It will also calculate the initial conditions and allocate the orig_data and filt_data arrays if those flags are set.

FUNCTION: FILT_generate_initial_conditions()

```
// Prototype
void	FILT_generate_initial_conditions(FILT_PROCESSING_STRUCT *filtps);
```

This function calculates and returns the initial conditions for a Butterworth filter of the specified type.

```
/*************************************************************************************/
/*********************************** RED Functions *******************************/
/*************************************************************************************/


// Structures
typedef struct {
```

```
	ui4	block_CRC;
	ui1	flags;
	ui1	protected_region[RED_BLOCK_PROTECTED_REGION_BYTES];
	ui1	discretionary_region[RED_BLOCK_DISCRETIONARY_REGION_BYTES];
	sf4	detrend_slope;
	sf4	detrend_intercept;
	sf4	scale_factor;
	ui4	difference_bytes;
	ui4	number_of_samples;
	ui4	block_bytes;
	si8	start_time;
	ui1	statistics[RED_BLOCK_STATISTICS_BYTES];
} RED_BLOCK_HEADER;


typedef struct {
	si1	encryption_level;  // encryption level for data blocks, passed in compression,
	                          // returned in decompression
	si1	discontinuity;  // set if block is first after a discontinuity, passed in
	                       // compression, returned in decompression
	si1	detrend_data;  // set if block is to be detrended (somewhat useful in lossless,
	                      // more useful in lossy compression)
	si1	return_lossy_data;  // if set, lossy data returned in decompressed_data during
	                           // lossy compression
	si1	reset_discontinuity;  // if discontinuity directive == MEF_TRUE, reset to
	                             // MEF_FALSE after compressing the block
	si1	require_normality;  // in lossy compression, lossless compression will be
	                           // performed in blocks whose samples are not approximately
	                           // normally distributed
	sf8	normal_correlation;  // if require_normality is set, the correlation of the sample
	                            // distribution with a normal distribution must be >= this
	                            // number (range -1.0 to 1.0)
} RED_PROCESSING_DIRECTIVES;


typedef struct {
	ui1	mode;  // compression mode
	sf8	goal_compression_ratio;  // goal value passed
	sf8	actual_compression_ratio;  // actual value returned in RED_FIXED_COMPRESSION_RATIO
	                                  // mode
	sf8	goal_mean_residual_ratio;  // goal value passed
	sf8	actual_mean_residual_ratio;  // actual value returned in RED_MEAN_RESIDUAL_RATIO
	                                    // mode
	sf8	goal_tolerance;  // tolerance for lossy compression mode goal, value of <= 0.0
	                        // uses default values, which are returned
	si4	maximum_rounds_per_block;  // maximum loops to attain goal compression
} RED_COMPRESSION_PARAMETERS;


typedef struct {
	ui4				counts[RED_BLOCK_STATISTICS_BYTES + 1];  // used by
	                                                  // RED_encode() & RED_decode()
	PASSWORD_DATA			*password_data;  // passed in compression & decompression
	RED_COMPRESSION_PARAMETERS	compression;
	RED_PROCESSING_DIRECTIVES	directives;
	si1				*difference_buffer;  // passed in both compression &
	                                                // decompression
	ui1				*compressed_data;  // passed in decompression, returned in
```

```
                                                         // compression, should not be updated
        RED_BLOCK_HEADER              *block_header; // points to beginning of current block
                                                     // within compressed_data array, updatable
        si4                           *decompressed_data;  // returned in decompression or if
                                                     // lossy data requested, used in some
                                                     // compression modes, should not be updated
        si4                           *decompressed_ptr;  // points to beginning of current block
                                                     // within decompressed_data array, updatable
        si4                           *original_data;  // passed in compression, should not be
                                                       // updated
        si4                           *original_ptr;  // points to beginning of current block
                                                      // within original_data array, updatable
        si4                           *detrended_buffer;  // used if needed in compression, size
                                                          // of decompressed block
        si4                           *scaled_buffer;  // used if needed in compression, size of
                                                       // decompressed block
} RED_PROCESSING_STRUCT;

// Macros
#define RED_MAX_DIFFERENCE_BYTES(x)          (x * 5)  // full si4 plus 1 keysample flag byte per
                                                      // sample
#define RED_MAX_COMPRESSED_BYTES(x, y)       ((RED_MAX_DIFFERENCE_BYTES(x) +
                                             RED_BLOCK_HEADER_BYTES + 7) * y)  // no compression
                                             // plus header plus maximum pad bytes, for y blocks
```

FUNCTION: RED_allocate_processing_struct()

```
// Prototype
RED_PROCESSING_STRUCT  *RED_allocate_processing_struct(si8 original_data_size, si8
        compressed_data_size, si8 decompressed_data_size, si8 difference_buffer_size, si8
        detrended_buffer_size, si8 scaled_buffer_size, PASSWORD_DATA *password_data);
```

Allocates a RED_PROCESSING_STRUCT (RPS). Within the RPS the various buffers are allocated. The PASSWORD_DATA structure is assigned. The directives are set to their defaults. The compression parameters are set to their defaults.

example:

```
rps = RED_allocate_processing_struct(max_samps, RED_MAX_COMPRESSED_BYTES(max_samps, 1), 0,
RED_MAX_DIFFERENCE_BYTES(max_samps), 0, 0, pwd);
```

Create an RPS large enough to compress a block of size max_samps. Lossless compression is the default, so no decompressed, offset, or scaled data buffers are requested.

FUNCTION: RED_calculate_mean_residual_ratio()

```
// Prototype
sf8     RED_calculate_mean_residual_ratio(si4 *original_data, si4 *lossy_data, ui4 n_samps);
```

Calculates and returns the mean residual ratio between the original_data and lossy_data buffers. Used in the MEAN_RESIDUAL_RATIO compression mode.

FUNCTION: RED_check_RPS_allocation()

```
// Prototype
si1    RED_check_RPS_allocation(RED_PROCESSING_STRUCT *rps);
```

Checks that the appropriate buffers are allocated in an RPS for the type of operation being performed. The operation is determined by the values of the members of the RPS's compression and directives structures. It returns MEF_TRUE if the appropriate buffers are allocated and MEF_FALSE if not unless the behavior_on_fail global is set to exit. Deficient allocations are printed to stderr, as are unnecessarily allocated buffers. This function may used if the programmer is uncertain which buffers to allocate for specific compression & decompression requirements. It is not called by any of the other functions in the library and must be called independently.

example:

```
RED_PROCESSING_STRUCT  *rps;
si1                    allocate_decompressed_data_buffer;

rps = RED_allocate_processing_struct(max_samps, RED_MAX_COMPRESSED_BYTES(max_samps, 1), 0,\
               RED_MAX_DIFFERENCE_BYTES(max_samps), 0, 0, password_data);

rps->compression.mode = RED_FIXED_COMPRESSION_RATIO;

force_behavior(RETURN_ON_FAIL);
RED_check_RPS_allocation(rps);
force_behavior(RESTORE_BEHAVIOR);
```

FUNCTION: RED_decode()

```
// Prototype
void   RED_decode(RED_PROCESSING_STRUCT *rps);
```

Decompress data passed in RPS from block_header pointer to RPS decompressed_ptr field. If CRC validation is requested in the directives, the block CRC will be checked, if the block does not have a valid CRC, it will not be decompressed and the function will return zero. If the block is encrypted and the access level is sufficient, the block will be decrypted before decompression. Encryption status is returned in the encryption directive. Scaling and detrending are performed as necessary. The block discontinuity status is returned in the discontinuity directive.

FUNCTION: RED_detrend()

```
// Prototype
ui4     RED_detrend(RED_PROCESSING_STRUCT *rps, si4 *input_buffer, si4 *output_buffer);
```

Detrends data from input_buffer to output_buffer. The detrended slope and intercept values entered into RPS's block_header. If the input_buffer == output_buffer detrending is done in place.

FUNCTION: RED_encode()

```
// Prototype
void    RED_encode(RED_PROCESSING_STRUCT *rps);
```

Compress data from original_ptr to block_header pointer (compressed data array). This is the main entry point into the library's compression routines.

FUNCTION: RED_encode_exec()

```
// Prototype
void    RED_encode_exec(RED_PROCESSING_STRUCT *rps, si4 *input_buffer, si1 input_is_detrended);
```

This is generally called by RED_encode() or RED_encode_lossy(), but can be called directly. It RED compresses from input_buffer to to block_header pointer (compressed data array). If the data is already detrended, it will not be done again. Encryption is done here according to the encryption directive, and the block_header flags are set appropriately. The discontinuity flag is set according to the discontinuity directive. The block CRC is calculated and filled in.

FUNCTION: RED_encode_lossy()

```
// Prototype
void    RED_encode_lossy(RED_PROCESSING_STRUCT *rps);

// Constants
#define RED_LOSSLESS_COMPRESSION                      0 // lossless (default)
#define RED_FIXED_SCALE_FACTOR                        1 // apply this scale factor to the block,
                                                        // 1.0 results in lossless compression;
#define RED_FIXED_COMPRESSION_RATIO                   2 // e.g. 20% of original si4 size is 0.2 -
                                                        // if lossless satisfies, no
                                                        // compression is done
#define RED_MEAN_RESIDUAL_RATIO                       3 // sum(abs((scaled_data -
                                                        // original_data))) /
                                                        // sum(abs(original_data)), e.g. 5%
                                                        // difference is 0.05
```

RED compress from original_ptr to block_header pointer (compressed data array), according to the specified compression mode. If lossy data is to be returned in the decompressed data buffer, this is generated. The function calls RED_encode_exec() for the actual compression which is described above. It returns the number of bytes (including pad bytes) in the compressed block.

example:

```
RED_PROCESSING_STRUCT  *rps;

rps = RED_allocate_processing_struct(max_samps, RED_MAX_COMPRESSED_BYTES(max_samps), 0,\
       RED_MAX_DIFFERENCE_BYTES(max_samps), 0, 0, password_data);
rps->directives.encryption = NO_ENCRYPTION;
rps->directives.discontinuity == MEF_FALSE;  // not a discontinuity
rps->block_header->number_of_samples = num_samps;
rps->block_header->start_time = start_time;
rps-> original_ptr = data_ptr;
RED_encode_lossy(rps);
```

FUNCTION: RED_filter()

```
// Prototype
void   RED_filter(FILT_PROCESSING_STRUCT *filtps);
```

Applies the filter specified by filtps to it's original data field. The sf8_filt_data are converted to si4s in the filt_data field. The filt_data field of the FILT_PROCESSING_STRUCT can be assigned to the filtered data buffer of a RED_PROCESSING_STRUCT for non-destructive filtering, or to the original data field for destructive filtering, with memory conservation.

FUNCTION: RED_find_extrema()

```
// Prototype
void   RED_find_extrema(si4 *buffer, si8 number_of_samples, TIME_SERIES_INDEX *tsi);
```

Finds the extrema in buffer and enters them into their respective fields in a time series index.

FUNCTION: RED_free_processing_struct()

FUNCTION: RED_free_processing_struct()

```
// Prototype
void   RED_free_processing_struct(RED_PROCESSING_STRUCT *rps);
```

Frees any non-NULL buffer pointers in a RED_PROCESSING_STRUCT, then frees the structure itself.

FUNCTION: RED_generate_lossy_data()

```
// Prototype
void    RED_generate_lossy_data(RED_PROCESSING_STRUCT *rps, si4 *input_buffer, si4
*output_buffer, si1 input_is_detrended);
```

Generates lossy data from input_buffer to output_buffer using the detrained and scale factors from the block_header. If the data is already detrended, it will not be done again. If input_buffer == output_buffer, lossy data will be generated in place.

FUNCTION: RED_initialize_normal_CDF_table()

```
// Prototype
sf8     *RED_initialize_normal_CDF_table(si4 global_flag);
```

Allocates and initializes the RED_normal_CDF_table (normal cumulative distribution function) into heap space. If the global_flag is set, the MEF_globals pointer RED_normal_CDF_table is also set to this value. This function is called by initialize_meflib() and the table is used by RED_test_normality(). The function returns a pointer to the table.

FUNCTION: RED_retrend()

```
// Prototype
si4     *RED_retrend(RED_PROCESSING_STRUCT *rps, si4 *input_buffer, si4 *output_buffer);
```

The function adds the trend specified by the block_header to the data from input_buffer to output_buffer. If the input_buffer == output_buffer retrending is done in place.

FUNCTION: RED_round()

```
// Prototype
si4     RED_round(sf8 val);
```

Rounds sf8 to si4 setting values that exceed RED_POSITIVE_INFINITY to RED_POSITIVE_INFINITY, and values less than RED_NEGATIVE_INFINITY to RED_NEGATIVE_INFINITY.

FUNCTION: RED_scale()

```
// Prototype
si4	*RED_scale(RED_PROCESSING_STRUCT *rps, si4 *input_buffer, si4 *output_buffer);
```

Scales data from input_buffer to output_buffer by the scale_factor in the block_header. If input_buffer == output_buffer, scaling will be done in place.

FUNCTION: RED_show_block_header()

```
// Prototype
void	RED_show_block_header(RED_BLOCK_HEADER *bh);

// Structure
typedef struct {
	ui4	block_CRC;
	ui1	flags;
	ui1	protected_region[RED_BLOCK_PROTECTED_REGION_BYTES];
	ui1	discretionary_region[RED_BLOCK_DISCRETIONARY_REGION_BYTES];
	sf4	detrend_slope;
	sf4	detrend_intercept;
	sf4	scale_factor;
	ui4	difference_bytes;
	ui4	number_of_samples;
	ui4	block_bytes;
	si8	start_time;
	ui1	statistics[RED_BLOCK_STATISTICS_BYTES];
} RED_BLOCK_HEADER;
```

This function displays the contents of a RED_BLOCK_HEADER structure. Can be useful in debugging code.

FUNCTION: RED_test_normality()

```
// Prototype
sf8	RED_test_normality(si4 *data, ui4 n_samps);
```

Returns the Pearson correlation of the normalized cumulative distribution of the input data to a pure normal cumulative distribution function (a variant of the Kolmogorov-Smirnov test for normality).

FUNCTION: RED_unscale()

```
// Prototype
si4	*RED_unscale(RED_PROCESSING_STRUCT *rps, si4 *input_buffer, si4 *output_buffer);
```

Removes the scale data from input_buffer to output_buffer by the scale_factor in the block_header. If input_buffer == output_buffer, unscaling will be done in place.

FUNCTION: RED_update_RPS_pointers()

```
// Prototype
RED_BLOCK_HEADER       *RED_update_RPS_pointers(RED_PROCESSING_STRUCT *rps, ui1 flags);

// Constants
#define RED_UPDATE_ORIGINAL_PTR       1
#define RED_UPDATE_BLOCK_HEADER_PTR   2  // will also update block_header pointer
#define RED_UPDATE_DECOMPRESSED_PTR   4
```

Convenience function to update the RPS pointers specified by flags. The block_header is updated by the block_header value block_bytes. Other pointers are updated by the block_header value number_of_samples. The function is inline, so there is no extra overhead. The block_header pointer is returned.

example:

```
ui1                    flags;
RED_BLOCK_HEADER       *block_header;

flags = RED_UPDATE_ORIGINAL_PTR | RED_UPDATE_BLOCK_HEADER_PTR | RED_UPDATE_DECOMPRESSED_PTR;
block_header = RED_update_RPS_pointers(rps, flags);
```

```
/*************************************************************************************/
/*********************************** CRC Utilities ********************************/
/*************************************************************************************/
```

FUNCTION: CRC_calculate()

```
// Prototype
ui4     CRC_calculate(ui1 *block_ptr, ui4 block_bytes);

// Constant
#define CRC_START_VALUE              0xFFFFFFFF
```

Returns the CRC of block of size block_bytes, pointed to by block_ptr.

```
crc = CRC_calculate(block_ptr, block_bytes);
```

is equivalent to:

```
crc = CRC_update(block_ptr, block_bytes, CRC_START_VALUE);
```

FUNCTION: CRC_initialize_table()

```
// Prototype
ui4    *CRC_initialize_table(si4 global_flag);
```

Allocates and initializes the CRC table generated from the 32-bit Koopman polynomial into heap space. If global_flag is set, the MEF_globals pointer CRC_table is also set to this value. This function is called by initialize_meflib().

FUNCTION: CRC_update()

```
// Prototype
ui4    CRC_update(ui1 *block_ptr, ui4 block_bytes, ui4 current_crc);
```

Returns the CRC of block of size block_bytes, pointed to by block_ptr, starting CRC value is passed in current_crc.

FUNCTION: CRC_validate()

```
// Prototype
si4    CRC_validate(ui1 *block_ptr, ui4 block_bytes, ui4 crc_to_validate);
```

Returns MEF_TRUE if the calculated CRC of the block pointed to by block_ptr matches the value passed in crc_to_validate. If they do not match, MEF_FLASE is returned.

```
/*************************************************************************************/
/*********************************** UTF-8 Utilities ********************************/
/*************************************************************************************/
```

```
// Prototypes

si4    UTF8_charnum(si1 *s, si4 offset);  // byte offset to character number

void   UTF8_dec(si1 *s, si4 *i);  // move to previous character

si4    UTF8_escape(si1 *buf, si4 sz, si1 *src, si4 escape_quotes);  // convert UTF-8 "src" to
       // ASCII with escape sequences.

si4    UTF8_escape_wchar(si1 *buf, si4 sz, ui4 ch);  // given a wide character, convert it to an
ASCII escape sequence stored in buf, where buf is "sz" bytes. returns the number of characters
output

si4    UTF8_f
uments may be in UTF-8. You can avoid this function and just use ordinary printf()
```

```
        // if the current locale is UTF-8.

si4     UTF8_hex_digit(si1 c);  // utility predicates used by the above

void    UTF8_inc(si1 *s, si4 *i);  // move to next character

ui4     *UTF8_initialize_offsets_from_UTF8_table(si4 global_flag);

si1     *UTF8_initialize_trailing_bytes_for_UTF8_table(si4 global_flag);

si4     UTF8_is_locale_utf8(si1 *locale);  // boolean function returns if locale is UTF-8, 0
        // otherwise

si1     *UTF8_memchr(si1 *s, ui4 ch, size_t sz, si4 *charn);  // same as the above, but searches
        // a buffer of a given size instead of a NUL-terminated string.

ui4     UTF8_nextchar(si1 *s, si4 *i);  // return next character, updating an index variable

si4     UTF8_octal_digit(si1 c);  // utility predicates used by the above

si4     UTF8_offset(si1 *str, si4 charnum);  // character number to byte offset

si4     UTF8_printf(si1 *fmt, ...);  // printf() where the format string and arguments may be in
        // UTF-8. You can avoid this function and just use ordinary printf() if the current
        // locale is UTF-8.

si4     UTF8_read_escape_sequence(si1 *str, ui4 *dest);  // assuming src points to the character
        // after a backslash, read an escape sequence, storing the result in dest and returning
        // the number of input characters processed

si4     UTF8_seqlen(si1 *s);  // returns length of next UTF-8 sequence

si1     *UTF8_strchr(si1 *s, ui4 ch, si4 *charn);  // return a pointer to the first occurrence of
        // ch in s, or NULL if not found. character index of found character returned in *charn.

si4     UTF8_strlen(si1 *s);  // count the number of characters in a UTF-8 string

si4     UTF8_toucs(ui4 *dest, si4 sz, si1 *src, si4 srcsz);  // convert UTF-8 data to wide
        // character

si4     UTF8_toutf8(si1 *dest, si4 sz, ui4 *src, si4 srcsz);  // convert wide character to UTF-8
data

si4     UTF8_unescape(si1 *buf, si4 sz, si1 *src);  // convert a string "src" containing escape
        // sequences to UTF-8 if escape_quotes is nonzero, quote characters will be preceded by
        // backslashes as well.

si4     UTF8_vfprintf(FILE *stream, si1 *fmt, va_list ap);    // called by UTF8_fprintf()

si4     UTF8_vprintf(si1 *fmt, va_list ap);  // called by UTF8_printf()

si4     UTF8_wc_toutf8(si1 *dest, ui4 ch);  // single character to UTF-8
```

Not all of the UTF-8 functions are used in the library, but they are included in the library for end-user and potential future use. Some of the included functions are used by other UTF-8 functions, and thus require inclusion. Only those functions that are currently used in other (non-UTF-8) meflib functions are described in this section.

FUNCTION: UTF8_initialize_offsets_from_UTF8_table()

```
// Prototype
ui4    *UTF8_initialize_offsets_from_UTF8_table(si4 global_flag);
```

Allocates and initializes the offsets_from_UTF8 table into heap space. If global_flag is set, the MEF_globals pointer UTF8_offsets_from_UTF8_table is also set to this value. This function is called by initialize_meflib().

FUNCTION: UTF8_initialize_trailing_bytes_for_UTF8_table()

```
// Prototype
si1    *UTF8_initialize_trailing_bytes_for_UTF8_table(si4 global_flag);
```

Allocates and initializes the trailing_bytes_for_UTF8 table into heap space. If global_flag is set, the MEF_globals pointer UTF8_trailing_bytes_for_UTF8_table is also set to this value. This function is called by initialize_meflib().

FUNCTION: UTF8_fprintf()

```
// Prototype
si4    UTF8_fprintf(FILE *stream, si1 *fmt, …);
```

Used like fprintf(), but accommodates UTF-8 as well as conventional strings.

FUNCTION: UTF8_nextchar()

```
// Prototype
ui4    UTF8_nextchar(si1 *s, si4 *i);
```

Returns the next character in the UTF-8 string s, updating the index variable i. Used by extract_terminal_password_bytes().

FUNCTION: UTF8_printf()

```
// Prototype si4      UTF8_printf(si1 *fmt, ...);
```

Used like printf(), but accommodates UTF-8 as well as conventional strings.

FUNCTION: UTF8_strlen()

```
// Prototype
si4    UTF8_strlen(si1 *s);
```

Returns the number of UTF-8 characters in the UTF-8 string s. Used by check_password().

```
/***********************************************************************************/
/********************************** AES Utilities *******************************/
/***********************************************************************************/
```

```
// Function Prototypes

void   AES_add_round_key(si4 round, ui1 state[][4], ui1 *RoundKey);

void   AES_decrypt(ui1 *in, ui1 *out, si1 *password, ui1 *expanded_key);

void   AES_encrypt(ui1 *in, ui1 *out, si1 *password, ui1 *expanded_key);

void   AES_key_expansion(si4 Nk, si4 Nr, ui1 *RoundKey, si1 *Key);

void   AES_cipher(si4 Nr, ui1 *in, ui1 *out, ui1 state[][4], ui1 *RoundKey);

si4    AES_get_sbox_invert(si4 num);

si4    AES_get_sbox_value(si4 num);

si4    *AES_initialize_rcon_table(si4 global_flag);

si4    *AES_initialize_rsbox_table(si4 global_flag);

si4    *AES_initialize_sbox_table(si4 global_flag);

void   AES_inv_cipher(si4 Nr, ui1 *in, ui1 *out, ui1 state[][4], ui1 *RoundKey);

void   AES_inv_mix_columns(ui1 state[][4]);

void   AES_inv_shift_rows(ui1 state[][4]);

void   AES_inv_sub_bytes(ui1 state[][4]);

void   AES_mix_columns(ui1 state[][4]);

void   AES_shift_rows(ui1 state[][4]);

void   AES_sub_bytes(ui1 state[][4]);
```

Not all of the AES functions are used by the other functions in the library, but are used by other AES functions, and thus require inclusion. Only those functions that are currently used in other (non-AES) meflib functions are described in this section.

FUNCTION: AES_initialize_rcon_table()

```
// Prototype
si4	*AES_initialize_rcon_table(si4 global_flag);
```

Allocates and initializes the AES rcon table into heap space. If global_flag is set, the MEF_globals pointer AES_rcon_table is also set to this value. This function is called by initialize_meflib().

FUNCTION: AES_initialize_rsbox_table()

```
// Prototype
si4	*AES_initialize_rsbox_table(si4 global_flag);
```

Allocates and initializes the AES rsbox table into heap space. If global_flag is set, the MEF_globals pointer AES_rsbox_table is also set to this value. This function is called by initialize_meflib().

FUNCTION: AES_initialize_sbox_table()

```
// Prototype
si4	*AES_initialize_sbox_table(si4 global_flag);
```

Allocates and initializes the AES sbox table into heap space. If global_flag is set, the MEF_globals pointer AES_sbox_table is also set to this value. This function is called by initialize_meflib().

FUNCTION: AES_decrypt()

```
// Prototype
void	AES_decrypt(ui1 *in, ui1 *out, si1 *password, ui1 *expanded_key);
```

Decrypts a 16 byte (128 bit) block of AES-128 encrypted data in the “in” buffer to the “out” buffer. The decryption can be done in place (“in” equals “out”), and is most often done this way within the library functions. Either expanded_key or password must be non-NULL. If both are non-NULL, the expanded key will be used, as it is more efficient. An expanded key can be obtained from the function AES_key_expansion(). If a password is to be used, an expanded key is generated from it, used, and discarded. A password is a 16 byte sequence. If, as is usually the case, this is a string, unused bytes should be zeroed, as these bytes, while meaningless to the string, cannot vary for reproducible decryption. If a UTF-8 string is used for a password, the meflib routines extract the terminal (most unique) bytes from each character to be used as the password bytes. This can be done with the function extract_terminal_password_bytes(); it is not done in this function.

FUNCTION: AES_encrypt()

```
// Prototype
void    AES_encrypt(ui1 *in, ui1 *out, si1 *password, ui1 *expanded_key);
```

Encrypts a 16 byte (128 bit) block of data in the "in" buffer to the "out" buffer using the AES-128 algorithm. The encryption can be done in place ("in" equals "out"), and is most often done this way within the library functions. Either expanded_key or password must be non-NULL. If both are non-NULL, the expanded key will be used, as it is more efficient. An expanded key can be obtained from the function AES_key_expansion(). If a password is to be used, an expanded key is generated from it, used, and discarded. A password is a 16 byte sequence. If, as is usually the case, this is a string, unused bytes should be zeroed, as these bytes, while meaningless to the string, cannot vary for reproducible encryption. If a UTF-8 string is used for a password, the meflib routines extract the terminal (most unique) bytes from each character to be used as the password bytes. This can be done with the function extract_terminal_password_bytes(); it is not done in this function.

FUNCTION: AES_key_expansion()

```
// Prototype
void    AES_key_expansion(ui1 *expanded_key, si1 *key);
```

Generates an expanded key from a key. A key is a 16 byte sequence. If, as is usually the case, the key is a password, unused bytes should be zeroed, as these bytes, while meaningless to the string, cannot vary for reproducible encryption / decryption. If a UTF-8 string is used for a password, the meflib routines extract the terminal (most unique) bytes from each character to be used as the password bytes. This can be done with the function extract_terminal_password_bytes(); it is not done in this function.

```
/**************************************************************************************/
/*********************************** SHA Utilities ***********************************/
/**************************************************************************************/
```

```
// Function Prototypes

ui4	*SHA256_initialize_h0_table(si4 global_flag);

ui4	*SHA256_initialize_k_table(si4 global_flag);

void	sha256(const ui1 *message, ui4 len, ui1 *digest);

void	SHA256_final(SHA256_ctx *ctx, ui1 *digest);

void	SHA256_init(SHA256_ctx *ctx);

void	SHA256_transf(SHA256_ctx *ctx, const ui1 *message, ui4 block_nb);

void	SHA256_update(SHA256_ctx *ctx, const ui1 *message, ui4 len);
```

SHA-256 is the 256-bit version of the SHA-2 cryptographic hash function. Only the 256-bit version is included in the library. Not all of the SHA functions are used by other functions in the library, but are used by other SHA functions, and thus require inclusion. Only those functions that are currently used in other (non-SHA) meflib functions are described in this section.

FUNCTION: SHA256_initialize_h0_table()

```
// Prototype
ui4	*SHA256_initialize_h0_table(si4 global_flag);
```

Allocates and initializes SHA AES h0 table into heap space. If global_flag is set, the MEF_globals pointer SHA_h0_table is also set to this value. This function is called by initialize_meflib().

FUNCTION: SHA256_initialize_k_table()

```
// Prototype
ui4	*SHA256_initialize_k_table(si4 global_flag);
```

Allocates and initializes SHA AES k table into heap space. If global_flag is set, the MEF_globals pointer SHA_k_table is also set to this value. This function is called by initialize_meflib().

FUNCTION: sha256()

```
// Prototype
```

```
void    sha256(const ui1 *message, ui4 len, ui1 *digest);

// Constant
#define SHA256_OUTPUT_SIZE    256
```

Returns a 256 byte SHA-2 hash of the message (of length len) in digest. This function is used by process_password_data().

Mefrec API

User defined records are defined and coded in “mefrec.c” and “mefrec.h”. The functions required for adding a new record type are described here. Record types themselves are described in the file “MEF 3 Records Specification”.

All records have an identically structured record header, followed by a customizable body. The body length must be padded out to a multiple of 16 bytes in length to facilitate individual record encryption with AES-128.

Structures within records should have all members aligned to their type and the total size evenly divisible by 8 (for 64-bit CPUs).

Records are named with 4 ascii characters and have a major and minor version associated with them so that they can evolve, as needed, with time. These 4 characters also define a type code as the bytes of a 4 byte unsigned integer. **Note that translation of ascii to hexadecimal on little endian machines requires reversing the byte ordering the hexadecimal representation.**

Each new record type should have two associated functions: a “show” function, and an “alignment” function. “Show” functions display the contents of the records and have the following form:

Name: show_mefrec_*xxxx*_type()
where “xxxx” is the record type name.

Prototype: `void      show_mefrec_xxxx_type(RECORD_HEADER *record_header);`
where a RECORD_HEADER is a structure defined in “meflib.h”

The “show” function should handle all versions of the record type. An example “show function is shown below for the “Note” record type.

```
void    show_mefrec_Note_type(RECORD_HEADER *record_header)
{
        si1    *Note;


        // Version 1.0
        if (record_header->version_major == 1 && record_header->version_minor == 0) {
                Note = (si1 *) record_header + MEFREC_Note_1_0_TEXT_OFFSET;
                UTF8_printf("Note text: %s\n", Note);
        }
        // Unrecognized record version
        else {
                printf("Unrecognized Note version\n");
        }
```

```
        return;
}
```

All show function constants are defined in “mefrec.h”. The function show_record() defined in mefrec.c must be modified in the switch statement, copied below, to add new record types.

```
switch (type_code) {
        case MEFREC_Note_TYPE_CODE:
                show_mefrec_Note_type(record_header);
                break;
        case MEFREC_Seiz_TYPE_CODE:
                show_mefrec_Seiz_type(record_header);
                break;
        case MEFREC_SyLg_TYPE_CODE:
                show_mefrec_SyLg_type(record_header);
                break;
        case MEFREC_UnRc_TYPE_CODE:
        default:
                printf("\"%s\" (0x%x) is an unrecognized record type\n", \
                record_header>type_string, type_code);
                break;
        }
```

“Alignment” functions have the following form:

Name: check_mefrec_xxxx_type_alignment()
where “xxxx” is the record type name.

Prototype: `si4        check_mefrec_Note_type_alignment(ui1 *bytes);`
where “bytes” is an optional buffer against which to check alignment

New record “alignment” functions check the alignment of any structures represented in the record body. Those structures are defined in “mefrec.h”. The function check_record_structure_alignments() defined in mefrec.c must be modified in the serial if statements, copied below, to add a new record type.

```
if ((check_mefrec_Note_type_alignment(bytes)) == MEF_FALSE)
        return_value = MEF_FALSE;
if ((check_mefrec_Seiz_type_alignment(bytes)) == MEF_FALSE)
        return_value = MEF_FALSE;
if ((check_mefrec_SyLg_type_alignment(bytes)) == MEF_FALSE)
        return_value = MEF_FALSE;
```