MED Library C API
version 1.0.1

(medlib_m11)

Library Naming & Tags:

- Format versions will be named as: "MED $F_{maj}.F_{min}$." where "F_{maj}" is the major format version, "F_{min}" is the minor format version.
- Libraries will be named as: "MED $F_{maj}.F_{min}$.L" where "L" is the library version for the major format version. There will be no sub-versioning of the library versions.
- MED major format versions are restricted to one digit (and thus capped at 9 under this schema). MED minor format versions start at zero, and are not restricted to one digit.
- A minor format version is guaranteed to be cross compatible with all versions in it's major category.
- Minor format versions may add fields to the format in protected regions, but no preexisting fields will be removed or moved.
- *Library Tags:* All functions, constants, macros, and data types defined in the library are tagged with the suffix "_mF_{maj}L" (or "MED format F_{maj}, library L" (for format F_{maj}))
- Minor format versions are not included in the tags. This is to keep them concise, and because all minor format versions are cross compatible with all versions in their major category.
- Library version numbering restarts at one with the release of a new major format version.
- There will be no sub-versioning of the library versions.
- All library versions associated with a particular format version are guaranteed to work on MED files of that format version.
- Cross compatibility will be maintained between MED format versions only if practical.
- In this document, the tag "_m1x" refers to MED format version 1, any library version.
- The full library name, & thus the minor format versions it supports, will be given in the header comments.

Examples:

- "MED 1.0.1 C API" is the C API for MED format 1.0, library version 1. This API will work on all MED 1 format files, but not necessarily MED 2-n format files. The tag for this library is "_m11".
- "MED 1.1.2 C API" is the C API for MED format 1.0, library version 2. This API will work on all MED 1.0 and 1.1 format files. The tag for this library is "_m12".
- The tag "_m213" indicates "MED format version 2.0-x, library version 13". This library will work on all MED 2 format files, but not necessarily MED 1 or MED 3-n format files.
- The tags allow different versions of the MED libraries to used within the same code. One such use might be in code for a MED 1 to MED 2 converter.

The MED library is evolved and renamed from the Multiscale Electrophysiology Format (MEF), versions 1-3. The library is open source, and curated by Dark Horse Neuro, Inc. (DHN). Source code is available through the medformat.org website. The layout of the format is described in “MED 1.0 Records Specification”

The MED 1.0 C library is composed of four files:

- medlib_m11.c
- medlib_m11.h
- medrec_m11.c
- medrec_m11.h

The “_m11” suffix denotes “MED 1.0” and is found on all functions and defines in the library. Future library versions will change this suffix accordingly, allowing for easy concurrent use of two library versions, such as in version converters.

The medlib_m11 files contain library functions that are not intended for user manipulation except in collaboration with DHN to maintain library uniformity. In contrast, the medrec_m11 files are designed to allow user addition of custom record types. Generally useful record types that are shared with DHN will be permanently incorporated into this code. The functions required for adding new record types are described in “MED 1.0 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 may require an explicit compiler instruction
                                        // to implement true long floating point math.
                                        // In icc the instruction is:
                                        // “-Qoption,cpp,–extended_float_types”
```

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

```
/**********************************************************************************/
/****************************** MED “Boolean” Schema ******************************/
/**********************************************************************************/

typedef  si1            TERN_m11;       // ternary “Boolean” type
#define  TRUE_m11       1
#define  UNKNOWN_m11    0
#define  FALSE_m11      -1
```

A balanced ternary schema including true, unknown, & false states. This is used throughout the library, and is typically represented by an si1 type, typdef'ed to TERN_m11.

```
/*****************************************************************************/
/********************************** MED Globals ******************************/
/*****************************************************************************/

// Structures
typedef struct {
        // Common MED Structures
        PASSWORD_DATA_m11               password_data;
        // Time Constants
        si8                             recording_time_offset;
        si4                             standard_UTC_offset;
        si1                             standard_timezone_acronym[TIMEZONE_ACRONYM_BYTES_m11];
        si1                             standard_timezone_string[TIMEZONE_STRING_BYTES_m11];
        TERN_m11                        observe_DST;
        TERN_m11                        RTO_known;
        si1                             daylight_timezone_acronym[TIMEZONE_ACRONYM_BYTES_m11];
        si1                             daylight_timezone_string[TIMEZONE_STRING_BYTES_m11];
        DAYLIGHT_TIME_CHANGE_CODE_m11   daylight_time_start_code;  // si1[8] / si8
        DAYLIGHT_TIME_CHANGE_CODE_m11   daylight_time_end_code;  // si1[8] / si8
        TIMEZONE_INFO_m11               *timezone_table;
        ui4                             recording_time_offset_mode;
        // Alignment Fields
        TERN_m11                        universal_header_aligned;
        TERN_m11                        metadata_section_1_aligned;
        TERN_m11                        time_series_metadata_section_2_aligned;
        TERN_m11                        video_metadata_section_2_aligned;
        TERN_m11                        metadata_section_3_aligned;
        TERN_m11                        all_metadata_structures_aligned;
        TERN_m11                        time_series_indices_aligned;
        TERN_m11                        video_indices_aligned;
        TERN_m11                        CMP_block_header_aligned;
        TERN_m11                        CMP_record_header_aligned;
        TERN_m11                        record_header_aligned;
        TERN_m11                        record_indices_aligned;
        TERN_m11                        all_record_structures_aligned;
        TERN_m11                        all_structures_aligned;
        // CMP
        sf8                             *CMP_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                             *SHA_h0_table;
        ui4                             *SHA_k_table;
        // UTF8 tables
        ui4                             *UTF8_offsets_table;
        si1                             *UTF8_trailing_bytes_table;
        // Miscellaneous
        TERN_m11                        verbose;
        ui4                             behavior_on_fail;
} GLOBALS_m11;

// Global Defaults
#define GLOBALS_VERBOSE_DEFAULT_m11                         FALSE_m11
#define GLOBALS_RECORDING_TIME_OFFSET_DEFAULT_m11           0
#define GLOBALS_RECORDING_TIME_OFFSET_NO_ENTRY_m11          0
```

```
#define GLOBALS_RECORDING_TIME_OFFSET_MODE_DEFAULT_m11      (RTO_APPLY_ON_OUTPUT_m11 | RTO_REMOVE_ON_INPUT_m11)
#define GLOBALS_STANDARD_UTC_OFFSET_DEFAULT_m11             0
#define GLOBALS_OBSERVE_DST_DEFAULT                         UNKNOWN_m11
#define GLOBALS_STANDARD_TIMEZONE_ACRONYM_DEFAULT_m11       "UTC"
#define GLOBALS_STANDARD_TIMEZONE_STRING_DEFAULT_m11        "Coordinated Universal Time"
#define GLOBALS_DAYLIGHT_TIMEZONE_ACRONYM_DEFAULT_m11       "UTC"
#define GLOBALS_DAYLIGHT_TIMEZONE_STRING_DEFAULT_m11        "Coordinated Universal Time"
#define GLOBALS_BEHAVIOR_ON_FAIL_DEFAULT_m11                EXIT_ON_FAIL_m11
#define GLOBALS_CRC_MODE_DEFAULT_m11                        CRC_CALCULATE_ON_OUTPUT_m11
#define GLOBALS_ALLOC_TRACKING_DEFAULT_m11                  FALSE_m11
#define GLOBALS_INIT_ALLOC_TRACKING_ARRAY_LEN_m11           100
#define GLOBALS_ALLOC_TRACKING_FUNCTION_STRING_LEN_m11      42
```

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

The recording_time_offset and standard_UTC_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 ********************/
/*****************************************************************************/

// Error Handling Constants
#define USE_GLOBAL_BEHAVIOR_m11         0
#define RESTORE_BEHAVIOR_m11            1
#define EXIT_ON_FAIL_m11                2
#define RETURN_ON_FAIL_m11              4
#define SUPPRESS_ERROR_OUTPUT_m11       8
#define SUPPRESS_WARNING_OUTPUT_m11     16
#define SUPPRESS_ALL_OUTPUT_m11         (SUPPRESS_ERROR_OUTPUT_m11 | SUPPRESS_WARNING_OUTPUT_m11)
#define RETRY_ONCE_m11                  32

// Function Prototypes
void    *e_calloc_m11(ui8 n_members, ui8 el_size, const si1 *function, si4 line, ui4 behavior_on_fail);
void    **e_calloc_2D_m11(ui8 dim1, ui8 dim2, ui8 el_size, const si1 *function, si4 line, ui4 behavior_on_fail);
FILE    *e_fopen_m11(si1 *path, si1 *mode, const si1 *function, si4 line, ui4 behavior_on_fail);
size_t  e_fread_m11(void *ptr, ui8 size, ui8 n_members, FILE *stream, si1 *path, const si1 *function, si4 line,
                ui4 behavior_on_fail);
void    e_free_m11(void *ptr, const si1 *function, si4 line);
void    e_free_2D_m11(void **ptr, si8 dim1, const si1 *function, si4 line);
si4     e_fseek_m11(FILE *stream, ui8 offset, si4 whence, si1 *path, const si1 *function, si4 line, ui4
                behavior_on_fail);
si8     e_ftell_m11(FILE *stream, const si1 *function, si4 line, ui4 behavior_on_fail);
ui8     e_fwrite_m11(void *ptr, ui8 size, ui8 n_members, FILE *stream, si1 *path, const si1 *function, si4 line,
                ui4 behavior_on_fail);
void    *e_malloc_m11(ui8 n_bytes, const si1 *function, si4 line, ui4 behavior_on_fail);
void    *e_realloc_m11(void *ptr, ui8 n_bytes, const si1 *function, si4 line, ui4 behavior_on_fail);

void    **e_realloc_2D_m11(void **curr_ptr, size_t curr_dim1, size_t new_dim1, size_t curr_dim2, size_t
                new_dim2, size_t el_size, const si1 *function, si4 line, ui4 behavior_on_fail);
si4     e_system_m11(si1 *command, TERN_m11 null_std_streams, const si1 *function, si4 line, ui4
                behavior_on_fail);
void    force_behavior_m11(ui4 behavior);
```

These functions are provided for convenience. They call their corresponding standard c functions (e.g. e_calloc_m11() calls calloc()), but have built in error messaging. The behavior_on_fail parameter defines what the function does on failure. They are written to maintain thread-safety.

force_behavior_m11() is described with the general functions, but essentially changes global behavior, and is one of the few library functions that ***is not thread-safe***.

Example:

```
ui4     behavior;
si4     *data;

behavior = (RETURN_ON_FAIL | SUPPRESS_ERROR_OUTPUT);
data = (si4 *) e_calloc_m11((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_m11() will return NULL, as would calloc(), and no error messages will be displayed. If USE_GLOBAL_BEHAVIOR is passed into this parameter, the 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 *********************/
/*****************************************************************************/

// Alignment Function Prototypes
TERN_m11       check_all_alignments_m11(const si1 *function, si4 line);
TERN_m11       check_metadata_alignment_m11(ui1 *bytes);
TERN_m11       check_metadata_section_1_alignment_m11(ui1 *bytes);
TERN_m11       check_metadata_section_3_alignment_m11(ui1 *bytes);
TERN_m11       check_record_header_alignment_m11(ui1 *bytes);
TERN_m11       check_record_indices_alignment_m11(ui1 *bytes);
TERN_m11       check_CMP_block_header_alignment_m11(ui1 *bytes);
TERN_m11       check_CMP_record_header_alignment_m11(ui1 *bytes);
TERN_m11       check_time_series_indices_alignment_m11(ui1 *bytes);
TERN_m11       check_time_series_metadata_section_2_alignment_m11(ui1 *bytes);
TERN_m11       check_universal_header_alignment_m11(ui1 *bytes);
TERN_m11       check_video_indices_alignment_m11(ui1 *bytes);
TERN_m11       check_video_metadata_section_2_alignment_m11(ui1 *bytes);
```

The structures in the MED library are designed such that they can be read in directly from their sources (e.g. disk, network) 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 TRUE, if they do not they return FALSE. Prior to checking, the global alignment flags are each set to UNKNOWN. In addition to a return value, each of these functions also sets its corresponding GLOBAL field to TRUE or FALSE.

The function check_all_alignments_m11() calls all of the other alignment checking functions and returns TRUE if all of those functions return TRUE. This function also takes a function and line argument similar to the error checking functions. This function is called from initialize_medlib_m11(), and so need not be called explicitly if initialize_medlib_m11() 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 (exerpted from check_all_alignments_m11()):

```
…
bytes = (ui1 *) e_malloc_m11(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 = MED_TRUE;
if ((check_universal_header_alignment_m11(bytes)) == MED_FALSE)
        return_value = MED_FALSE;
if ((check_metadata_alignment_m11(bytes)) == MED_FALSE)
        return_value = MED_FALSE;
if ((check_CMP_block_header_alignment_m11(bytes)) == MED_FALSE)
        return_value = MED_FALSE;
if ((check_time_series_indices_alignment_m11(bytes)) == MED_FALSE)
        return_value = MED_FALSE;
if ((check_video_indices_alignment_m11(bytes)) == MED_FALSE)
        return_value = MED_FALSE;
if ((check_record_indices_alignment_m11(bytes)) == MED_FALSE)
        return_value = MED_FALSE;
if ((check_record_header_alignment_m11(bytes)) == MED_FALSE)
        return_value = MED_FALSE;
if ((check_record_structure_alignments_m11(bytes)) == MED_FALSE)
        return_value = MED_FALSE;

free(bytes);

return(return_value);
```

example 2 (the most common use):

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

```
/**********************************************************************************/
/****************************** MED String Functions ******************************/
/**********************************************************************************/

// Standard String Function Prototypes
si4     sprintf_m11(si1 *target, si1 *format, ...);
void    snprintf_m11(si1 *target, si4 target_field_bytes, si1 *format, ...);
si4     strcat_m11(si1 *target_string, si1 *source_string);
si4     strcpy_m11(si1 *target_string, si1 *source_string);
void    strncat_m11(si1 *target_string, si1 *source_string, si4 target_field_bytes);
void    strncpy_m11(si1 *target_string, si1 *source_string, si4 target_field_bytes);

// UTF8 Prototypes: these are described with the other UTF8 functions, but are mentioned here only because they
// are string functions with standard correlates.
si4     UTF8_fprintf_m11(FILE *stream, si1 *fmt, …);
si4     UTF8_printf_m11(si1 *fmt, ...);
si4     UTF8_strlen_m11(si1 *s);
```

As a group, these functions facilitate working with MED strings. When there is a target string, unoccupied bytes are filled with zeros, per the MED specification.

```
/**********************************************************************************/
/************************** General Purpose MED Functions *************************/
/**********************************************************************************/
```

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

FUNCTION: all_zeros_m11()

```
// Prototype
TERN_m11       all_zeros_m11(ui1 *bytes, si4 field_length);
```

This function returns TRUE if field pointed to by “bytes” contains all zeros, and 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 (extracted from show_universal_header_m11()):

```
if (all_zeros(uh->level_1_password_validation_field, PASSWORD_VALIDATION_FIELD_BYTES) == TRUE_m11)
        printf("Level 1 Password Validation_Field: no entry\n");
```

FUNCTION: allocate_file_processing_struct_m11()

```
// Prototype
FILE_PROCESSING_STRUCT_m11    *allocate_file_processing_struct_m11(FILE_PROCESSING_STRUCT_m11 *fps, si1
*full_file_name, ui4 type_code, si8 raw_data_bytes, FILE_PROCESSING_STRUCT_m11 *proto_fps, si8 bytes_to_copy);
```

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

```
// Structures
typedef struct {
        TERN_m11                                mutex;  // used by MED functions to ensure thread
                                                        // safety
        PASSWORD_DATA_m11                       *password_data;  // this will typically be the same
        si1                                     full_file_name[FULL_FILE_NAME_BYTES_m11];  // full
                                                                // path including extension
        FILE                                    *fp;  // file pointer
        si4                                     fd;  // file descriptor
        si8                                     file_length;
        UNIVERSAL_HEADER_m11                    *universal_header;
        FILE_PROCESSING_DIRECTIVES_m11          directives;
        union {                                 // the MED file types
                METADATA_m11                    *metadata;
                ui1                             *records;
                RECORD_INDEX_m11                *record_indices;
                TIME_SERIES_INDEX_m11           *time_series_indices;
                VIDEO_INDEX_m11                 *video_indices;
        };
        si8                                     raw_data_bytes;
        ui1                                     *raw_data;
        CMP_PROCESSING_STRUCT_m11               *cps;  // associated with time series data FPS, NULL
                                                       // in others
} FILE_PROCESSING_STRUCT_m11;

typedef struct {
        TERN_m11        close_file;
        TERN_m11        flush_after_write;
        TERN_m11        update_universal_header;  // when writing
        TERN_m11        leave_decrypted;  // if encrypted during write, return from write
                                          // function decrypted, also leave times unoffset
        TERN_m11        free_password_data;  // when freeing FPS
        TERN_m11        free_CMP_processing_struct;  // when freeing FPS

        ui4             lock_mode;
```

```
        ui4             open_mode;
} FILE_PROCESSING_DIRECTIVES_m11;


// Constants
#define FPS_FILE_LENGTH_UNKNOWN_m11          -1
#define FPS_FULL_FILE_m11                    -1
#define FPS_NO_LOCK_TYPE_m11                 (~(F_RDLCK | F_WRLCK | F_UNLCK))  // from <fcntl.h>
#define FPS_NO_LOCK_MODE_m11                 0
#define FPS_READ_LOCK_ON_READ_OPEN_m11       1
#define FPS_WRITE_LOCK_ON_READ_OPEN_m11      2
#define FPS_WRITE_LOCK_ON_WRITE_OPEN_m11     4
#define FPS_WRITE_LOCK_ON_READ_WRITE_OPEN_m11        8
#define FPS_READ_LOCK_ON_READ_m11            16
#define FPS_WRITE_LOCK_ON_WRITE_m11          32
#define FPS_NO_OPEN_MODE_m11                 0
#define FPS_R_OPEN_MODE_m11                  1
#define FPS_R_PLUS_OPEN_MODE_m11             2
#define FPS_W_OPEN_MODE_m11                  4
#define FPS_W_PLUS_OPEN_MODE_m11             8
#define FPS_A_OPEN_MODE_m11                  16
#define FPS_A_PLUS_OPEN_MODE_m11             32
#define FPS_GENERIC_READ_OPEN_MODE_m11       (FPS_R_OPEN_MODE_m11 | FPS_R_PLUS_OPEN_MODE_m11 | \
                                             FPS_W_PLUS_OPEN_MODE_m11 | FPS_A_PLUS_OPEN_MODE_m11)
#define FPS_GENERIC_WRITE_OPEN_MODE_m11      (FPS_R_PLUS_OPEN_MODE_m11 | FPS_W_OPEN_MODE_m11 | \
                                             FPS_W_PLUS_OPEN_MODE_m11 | FPS_A_OPEN_MODE_m11 | \
                                             FPS_A_PLUS_OPEN_MODE_m11)
#define FPS_PROTOTYPE_FILE_TYPE_CODE_m11     TIME_SERIES_METADATA_FILE_TYPE_CODE_m11  // any
                                                    // metadata type would do
#define FPS_FD_NO_ENTRY_m11                  -2
#define FPS_FD_EPHEMERAL_m11                 -3


// File Processing Directives Defaults
#define FPS_DIRECTIVE_CLOSE_FILE_DEFAULT_m11                        TRUE_m11
#define FPS_DIRECTIVE_FLUSH_AFTER_WRITE_DEFAULT_m11                 TRUE_m11
#define FPS_DIRECTIVE_FREE_PASSWORD_DATA_DEFAULT_m11                FALSE_m11
#define FPS_DIRECTIVE_FREE_CMP_PROCESSING_STRUCT_DEFAULT_m11        TRUE_m11
#define FPS_DIRECTIVE_UPDATE_UNIVERSAL_HEADER_DEFAULT_m11           FALSE_m11
#define FPS_DIRECTIVE_LEAVE_DECRYPTED_DEFAULT_m11                   FALSE_m11
#define FPS_DIRECTIVE_LOCK_MODE_DEFAULT_m11                         FPS_NO_LOCK_MODE_m11  // Unix
                                                                    // file locking may cause
                                                                    // problems with networked
                                                                    // file systems
// #define FPS_DIRECTIVE_LOCK_MODE_DEFAULT_m11      (FPS_READ_LOCK_ON_READ_OPEN_m11 |
                                                    // FPS_WRITE_LOCK_ON_WRITE_OPEN_m11 |
                                                    // FPS_WRITE_LOCK_ON_READ_WRITE_OPEN_m11)
#define FPS_DIRECTIVE_OPEN_MODE_DEFAULT_m11                         FPS_NO_OPEN_MODE_m11
#define FPS_DIRECTIVE_IO_BYTES_DEFAULT_m11                          FPS_FULL_FILE_m11  // bytes
                                                                    // to read or write
#define FPS_UNIVERSAL_HEADER_ONLY_m11                               0
```

The FILE_PROCESSING_STRUCT (FPS) is the fundamental file handling unit of the MED library. The raw_data field contains the data as it is arranged in the MED 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 is the amount of memory to be allocated to the raw_data field minus the amount needed for a universal header, as all MED files have a universal header.

The FILE_PROCESSING_STRUCT's file_length field reflects the size of the file on disk (in bytes). This is set to zero on allocation, but is updated during read and write operations using MED library functions.

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 plus universal header bytes is greater than raw_data_bytes). The amount of raw_data copied is specified in the bytes_to_copy field. Bytes_to_copy does not include the bytes universal header bytes, as this is assumed. If copying is performed, the universal header's CRC will be not be recalculated, and may be inaccurate. This is updated in write_file_m11() before write out, and so is not usually an issue. It could be explicitly calculated with calculate_CRC_m11().

If the prototype pointer is NULL, the file processing directives and universal header 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 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 FALSE**. lock_mode** specifies *advisory locking* on the file. All the MED 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.

The type_code specifies which of the FILE_PROCESSING_STRUCT pointers will be assigned to the raw_data after the universal header. The type_string field of the universal header is also set by the type_code. If the 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 (just space for a universal header)

```
fps = allocate_file_processing_struct_m11(NULL, NULL, NO_TYPE_CODE_m11, 0, NULL, 0);
NO_TYPE_CODE_m11 == 0, so fps = allocate_file_processing_struct_m11(NULL, NULL, 0, 0, NULL, 0); would be the same
```

example 2: allocate a metadata FILE_PROCESSING_STRUCT and copy its universal header from the prototype FPS, "other_fps"

```
fps = allocate_file_processing_struct_m11(NULL, full_file_name, TIME_SERIES_METADATA_FILE_TYPE_CODE_m11,
METADATA_FILE_BYTES_m11, other_fps, FPS_UNIVERSAL_HEADER_ONLY_m11);
FPS_UNIVERSAL_HEADER_ONLY_m11 == 0, so fps = allocate_file_processing_struct_m11(NULL, full_file_name, TIME_SERIES_METADATA_FILE_TYPE_CODE_m11,
METADATA_FILE_BYTES_m11, other_fps, 0); would be the same
```

example 3: 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_m11(NULL, full_file_name, TIME_SERIES_METADATA_FILE_TYPE_CODE_m11,
METADATA_FILE_BYTES_m11, other_metadata_fps, METADATA_FILE_BYTES_m11);
```

FUNCTION: allocate_metadata_m11()

```
// Prototype
METADATA_m11    *allocate_metadata_m11(METADATA_m11 *metadata, ui1 *data_ptr);

// Structures
typedef struct {
        METADATA_SECTION_1_m11                      section_1;
        union {
                TIME_SERIES_METADATA_SECTION_2_m11  time_series_section_2;
                VIDEO_METADATA_SECTION_2_m11        video_section_2;
        };
        METADATA_SECTION_3_m11                      section_3;
} METADATA_m11;

typedef struct {
        si1     level_1_password_hint[PASSWORD_HINT_BYTES_m11];
```

```
        si1     level_2_password_hint[PASSWORD_HINT_BYTES_m11];
        si1     section_2_encryption_level;
        si1     section_3_encryption_level;
        ui1     protected_region[METADATA_SECTION_1_PROTECTED_REGION_BYTES_m11];
        ui1     discretionary_region[METADATA_SECTION_1_DISCRETIONARY_REGION_BYTES_m11];
} METADATA_SECTION_1_m11;

typedef struct {
        // channel type independent fields
        si1     session_description[METADATA_SESSION_DESCRIPTION_BYTES_m11];  // utf8[511]
        si1     channel_description[METADATA_CHANNEL_DESCRIPTION_BYTES_m11];  // utf8[255]
        si1     segment_description[METADATA_SEGMENT_DESCRIPTION_BYTES_m11];  // utf8[255]
        si1     equipment_description[METADATA_EQUIPMENT_DESCRIPTION_BYTES_m11];  // utf8[510]
        si4     acquisition_channel_number;
        // channel type specific fields
        si1     reference_description
                        [TIME_SERIES_METADATA_REFERENCE_DESCRIPTION_BYTES_m11];  // utf8[255]
        sf8     sampling_frequency;
        sf8     low_frequency_filter_setting;
        sf8     high_frequency_filter_setting;
        sf8     notch_filter_frequency_setting;
        sf8     AC_line_frequency;
        sf8     amplitude_units_conversion_factor;
        si1     amplitude_units_description \
                        [TIME_SERIES_METADATA_AMPLITUDE_UNITS_DESCRIPTION_BYTES_m11];  // utf8[31]
        sf8     time_base_units_conversion_factor;
        si1 time_base_units_description \
                        [TIME_SERIES_METADATA_TIME_BASE_UNITS_DESCRIPTION_BYTES_m11];  // utf8[31]
        si8     absolute_start_sample_number;
        si8     number_of_samples;
        si8     number_of_blocks;
        si8     maximum_block_bytes;
        ui4     maximum_block_samples;
        ui4     maximum_block_difference_bytes;
        sf8     maximum_block_duration;
        si8     number_of_discontinuities;
        si8     maximum_contiguous_blocks;
        si8     maximum_contiguous_block_bytes;
        si8     maximum_contiguous_samples;
        ui1     protected_region[TIME_SERIES_METADATA_SECTION_2_PROTECTED_REGION_BYTES_m11];
        ui1     discretionary_region \
                        [TIME_SERIES_METADATA_SECTION_2_DISCRETIONARY_REGION_BYTES_m11];
} TIME_SERIES_METADATA_SECTION_2_m11;

typedef struct {
        // type-independent fields
        si1     session_description[METADATA_SESSION_DESCRIPTION_BYTES_m11];  // utf8[511]
        si1     channel_description[METADATA_CHANNEL_DESCRIPTION_BYTES_m11];  // utf8[511]
        si1     segment_description[METADATA_SEGMENT_DESCRIPTION_BYTES_m11];  // utf8[511]
        si1     equipment_description[METADATA_EQUIPMENT_DESCRIPTION_BYTES_m11];  // utf8[510]
        si4     acquisition_channel_number;
        // type-specific fields
        si8     horizontal_resolution;
        si8     vertical_resolution;
        sf8     frame_rate;
        si8     number_of_clips;
        si8     maximum_clip_bytes;
        si1     video_format[VIDEO_METADATA_VIDEO_FORMAT_BYTES_m11];  // utf8[31]
        si4     number_of_video_files;
        ui1     protected_region[VIDEO_METADATA_SECTION_2_PROTECTED_REGION_BYTES_m11];
        ui1     discretionary_region[VIDEO_METADATA_SECTION_2_DISCRETIONARY_REGION_BYTES_m11];
} VIDEO_METADATA_SECTION_2_m11;

typedef struct {
```

```
        si8     recording_time_offset;
        DAYLIGHT_TIME_CHANGE_CODE_m11 daylight_time_start_code;  // si1[8] / si8
        DAYLIGHT_TIME_CHANGE_CODE_m11 daylight_time_end_code;  // si1[8] / si8
        si1     standard_timezone_acronym[TIMEZONE_ACRONYM_BYTES_m11];  // ascii[8]
        si1     standard_timezone_string[TIMEZONE_STRING_BYTES_m11];  // ascii[31]
        si1     daylight_timezone_acronym[TIMEZONE_ACRONYM_BYTES_m11];  // ascii[8]
        si1     daylight_timezone_string[TIMEZONE_STRING_BYTES_m11];  // ascii[31]
        si1     subject_name_1[METADATA_SUBJECT_NAME_BYTES_m11];  // utf8[31]
        si1     subject_name_2[METADATA_SUBJECT_NAME_BYTES_m11];  // utf8[31]
        si1     subject_name_3[METADATA_SUBJECT_NAME_BYTES_m11];  // utf8[31]
        si1     subject_ID[METADATA_SUBJECT_ID_BYTES_m11];  // utf8[31]
        si1     recording_country[METADATA_RECORDING_LOCATION_BYTES_m11];  // utf8[63]
        si1     recording_territory[METADATA_RECORDING_LOCATION_BYTES_m11];  // utf8[63]
        si1     recording_city[METADATA_RECORDING_LOCATION_BYTES_m11];  // utf8[63]
        si1     recording_institution[METADATA_RECORDING_LOCATION_BYTES_m11];  // utf8[63]
        si1     geotag_format[METADATA_GEOTAG_FORMAT_BYTES_m11];  // ascii[31]
        si1     geotag_data[METADATA_GEOTAG_DATA_BYTES_m11];  // ascii[1023]
        si4     standard_UTC_offset;
        ui1     protected_region[METADATA_SECTION_3_PROTECTED_REGION_BYTES_m11];
        ui1     discretionary_region[METADATA_SECTION_3_DISCRETIONARY_REGION_BYTES_m11];
} METADATA_SECTION_3_m11;
```

This function allocates a metadata structure (if NULL is passed), and the memory for the structures it contains (again, if NULL is passed). Internal memory is allocated to metadata->metadata, and caller must free that before freeing the metadata structure itself (you can also use free_metadata_m11() which does this for you). The function sets the internal pointers appropriately. Use of this functions is typically unnecessary as allocate_file_processing_struct_m11() for a metadata FPS perform this allocation and assignment.

FUNCTION: calculate_metadata_CRC_m11()

```
// Prototype
void  calculate_metadata_CRC_m11(FILE_PROCESSING_STRUCT_m11 *fps);
```

Calculates the CRC for a metadata FPS, and enters that value in the universal header body_CRC.

FUNCTION: calculate_record_data_CRCs_m11()

```
// Prototype
void  calculate_record_data_CRCs_m11(FILE_PROCESSING_STRUCT_m11 *fps, RECORD_HEADER_m11
*record_header, si8 number_of_items);
```

Calculates the CRCs for number_of_items record data entries, and enters their values in the record headers. It also updates the universal header body_CRC for each record. The record_header pointer is the beginning of an array of records.

FUNCTION: calculate_record_indices_CRCs_m11()

```
// Prototype
void  calculate_record_indices_CRCs_m11(FILE_PROCESSING_STRUCT_m11 *fps, RECORD_INDEX_m11
*record_index, si8 number_of_items);
```

Calculates the CRCs for number_of_items record indices, and updates the universal header body_CRC for each index. The record_index pointer is the beginning of an array of record indices.

FUNCTION: calculate_time_series_data_CRCs_m11()

```
// Prototype
void  calculate_time_series_data_CRCs_m11(FILE_PROCESSING_STRUCT_m11 *fps,
CMP_BLOCK_FIXED_HEADER_m11 *block_header, si8 number_of_items);
```

Calculates the CRCs for number_of_items CMP blocks (time series data blocks entries), and enters their values in the CMP block headers. It also updates the universal header body_CRC for each record. The block_header pointer is the beginning of an array of CMP blocks.

FUNCTION: calculate_time_series_indices_CRCs_m11()

```
// Prototype
void  calculate_time_series_indices_CRCs_m11(FILE_PROCESSING_STRUCT_m11 *fps,
TIME_SERIES_INDEX_m11 *time_series_index, si8 number_of_items);
```

Calculates the CRCs for number_of_items time series indices, and updates the universal header body_CRC for each index. The time_series_index pointer is the beginning of an array of time series indices.

FUNCTION: channel_type_from_path_m11()

```
// Prototype
ui4   channel_type_from_path_m11(si1 *path);
```

Returns MED type code based on extension of the passed file name or path. UNKNOWN_CHANNEL_TYPE_m11 is returned for an unrecognized extension.

FUNCTION: check_password_m11()

```
// 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_m11()):

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

FUNCTION: condition_time_slice_m11()

```
// Prototype
void  condition_time_slice_m11(TIME_SLICE_m11 *slice)
```

Does the following to the passed time slice:
1) Offsets unoffset times (using global recording_time_offset)
2) Makes relative times (negative) absolute (using global session_start_time)
3) Sets start_time & end_time to BEGINNING_OF_TIME_m11 & END_OF_TIME_m11 if no index parameters are specified
4) Sets slice->conditioned to TRUE_m11

NOTE: globals session_start_time & recording_time_offset must be set before using this function.

FUNCTION: current_uutc_m11()

```
// Prototype
inline si8  current_uutc_m11(void);
```

Returns the current μUCT based on the system clock.

FUNCTION: days_in_month_m11()

```
// Prototype
inline si4  days_in_month_m11(si4 month, si4 year);
```

Returns the days in a month based on the month and year. Note month is [0 - 11], where January == 0 (as in struct tm.tm_mon in <time.h>). This function expects the full value of the year (note struct tm.tm_year is (year - 1900) in <time.h>).

FUNCTION: decrypt_metadata_m11()

```
// Prototype
TERN_m11    decrypt_metadata_m11(FILE_PROCESSING_STRUCT_m11 *fps);
```

Decrypts metadata in a metadata FPS in place. Returns TRUE_m11 on success, FALSE_m11 on failure.

FUNCTION: decrypt_records_m11()

```
// Prototype
TERN_m11    decrypt_records_m11(RECORD_HEADER_m11 *record_header, si8 number_of_items);
```

Decrypts the number_of_items records pointed to by record_header. Returns TRUE_m11 on success, FALSE_m11 on failure.

FUNCTION: decrypt_time_series_data_m11()

```
// Prototype
TERN_m11    decrypt_time_series_data_m11(CMP_BLOCK_FIXED_HEADER_m11 *block_header, si8
number_of_items);
```

Decrypts the number_of_items CMP blocks pointed to by block_header. Returns TRUE_m11 on success, FALSE_m11 on failure.

FUNCTION: DST_offset_m11()

```
// Prototype
si4   DST_offset_m11(si8 uutc);
```

Returns seconds to add to standard time (as UUTC) to adjust for DST on that date, in the timezone specified in the MED globals.

FUNCTION: error_message_m11()

```
// Prototype
void  error_message_m11(si1 *fmt, ...);
```

Prints an error message to stderr in accordance with the parameters of the behavior_on_fail global variable. Used like fprintf(). Also see message_m11() , warning_message_m11() and force_behavior_m11().

Example:

```
error_message_m11("%s(): Cannot encrypt data (called from line %d)", __FUNCTION__, \
      __LINE__);
```

If you wish to ensure an exit(), regardless go the global behavior, do something like this:

```
if (x == 0) {
      error_message_m11("%s(): Divide by zero", __FUNCTION__);
      exit(1);
}
```

FUNCTION: escape_spaces_m11()

```
// Prototype
void    escape_spaces_m11(si1 *string, si8 buffer_len);
```

Escapes all spaces (inserts a backslash ('\')) before all unescaped spaces in the string.

FUNCTION: extract_path_parts_m11()

```
// Prototype
void  extract_path_parts_m11(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.
example:

```
si1   *passed_session_directory = “/Data/Session_1.medd”;

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 “medd”. 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_m11()

```
// Prototype
si4   extract_terminal_password_bytes_m11(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_m11().

FUNCTION: file_exists_m11()

```
// Prototype
ui4   file_exists_m11(si1 *path);

#define DOES_NOT_EXIST_m11    0
#define FILE_EXISTS_m11       1
#define DIR_EXISTS_m11        2
```

Returns FILE_EXISTS_m11 if path describes a file. Returns DIR_EXISTS_m11 if path describes a directory. Returns DOES_NOT_EXIST_m11 if path does not point to either a file or directory. If path does not begin from root, the current working directory is prepended.

FUNCTION: find_discontinuities_m11()

```
// Prototype
si8   *find_discontinuities_m11(TIME_SERIES_INDEX_m11 *tsi, si8 num_disconts, si8
number_of_indices, TERN_m11 remove_offsets, TERN_m11 return_sample_numbers);
```

Allocates and returns an array of indices (into the tsi array) of blocks with discontinuities. If remove_offsets == TRUE_m11, the tsi start sample numbers will be “unflagged” to their true numbers. If return_sample_numbers == TRUE_m11, start sample numbers will be returned in the array, rather than indices.

FUNCTION: force_behavior_m11()

```
// Constants
#define RESTORE_BEHAVIOR      -1
```

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

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

THIS ROUTINE IS NOT THREAD SAFE: USE WITH CARE IN THREADED CODE.

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

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


//*********************************************************************//
//************  FILE PROCESSING STRUCT STANDARD FUNCTIONS  **************//
//*********************************************************************//
```

FUNCTION: fps_close_m11()

```
// 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_m11()

```
// Constants
#define FPS_NO_LOCK_TYPE                 ~(F_RDLCK | F_WRLCK | F_UNLCK)  // <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_m11(FILE_PROCESSING_STRUCT_m11 *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 from whence the function was called in the case of failure.

NOTE: advisory locks often do not work on networked file systems.

FUNCTION: fps_mutex_off_m11()

```
// Prototype
inline void fps_mutex_off_m11(FILE_PROCESSING_STRUCT_m11 *fps);
```

Removes mutex lock on an FPS. The mutex is used by read_file_m11() & write_file_m11() to make sure fps is not in use while reading or writing. Other functions may also use this mechanism.

FUNCTION: fps_mutex_on_m11()

```
// Prototype
inline void fps_mutex_on_m11(FILE_PROCESSING_STRUCT_m11 *fps);
```

Activates mutex lock on an FPS. The mutex is used by read_file_m11() & write_file_m11() to make sure fps is not in use while reading or writing.Other functions may also use this mechanism.

FUNCTION: fps_open_m11()

```
// 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_m11(FILE_PROCESSING_STRUCT_m11 *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_m11()

```
// Prototype
si4   fps_read_m11(FILE_PROCESSING_STRUCT_m11 *fps, si8 in_bytes, void *ptr, const si1 \
          *function, si4 line, ui4 behavior_on_fail);
```

Locks the file, reads in_bytes bytes, and unlocks the file. Lock type is specified by the FPS directive's lock_mode.

FUNCTION: fps_unlock_m11()

```
// Prototype
si4   fps_unlock_m11(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 from whence the function was called in the case of failure.

NOTE: advisory locks often do not work on networked file systems.

FUNCTION: free_channel_m11()

```
// Prototype
void  free_channel_m11(CHANNEL_m11 *channel, TERN_m11 channel_allocated_en_bloc);
```

Frees all the memory pointed to by a CHANNEL structure including all memory associated with SEGMENT structures within it. If channel_allocated_en_bloc == FALSE_m11, the passed CHANNEL structure will itself be freed also.

FUNCTION: free_file_processing_struct_m11()

```
// Prototype
void  free_file_processing_struct_m11(FILE_PROCESSING_STRUCT_m11 *fps, TERN_m11 \
           allocated_en_bloc);
```

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 == TRUE_m11. If the FPS contains a CMP processing struct and the free_CMP_processing_struct directive == TRUE_m11, this will also be freed. If the free_password_data directive == TRUE_m11, the FILE_PROCESSING_STRUCT's password_data will be freed. If allocated_en_bloc == FALSE_m11, the FPS structure will itself be freed also.

FUNCTION: free_metadata_m11()

```
// Prototype
void  *free_metadata_m11(METADATA_m11 *metadata);
```

Frees passed metadata structure and associated memory.

FUNCTION: free_segment_m11()

```
// Prototype
void  free_segment_m11(SEGMENT_m11 *segment, TERN_m11 segment_allocated_en_bloc);
```

Frees all the memory pointed to by a SEGMENT structure. If segment_allocated_en_bloc == FALSE_m11, the passed SEGMENT structure will itself be freed also.
.

FUNCTION: free_session_m11()

```
// Prototype
void  free_session_m11(SESSION *session);
```

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. The passed SESSION structure will itself be freed also.

FUNCTION: generate_file_list_m11()

```
// Prototype
si1   **generate_file_list_m11(si1 **file_list, si4 n_in_files, si4 *n_out_files, si1 \
            *enclosing_directory, si1 *name, si1 *extension, ui1 path_parts, TERN_m11 \
            free_input_file_list);

// Path Parts
#define PP_PATH_m11             1
#define PP_NAME_m11             2
#define PP_EXTENSION_m11        4
#define PP_FULL_PATH_m11        (PP_PATH_m11 | PP_NAME_m11 | PP_EXTENSION_m11)
```

This function may seem a bit arcane, but it is really quite useful. It returns a list of files given the inputs as follows:

1. If the passed file_list not NULL, a new file list is still returned; the passed list will be freed if free_input_file_list == TRUE_m11. The point of this is to allow passing of lists with regex that will be expanded. If enclosing directory is not NULL, and the passed file list entries do not begin from root, the enclosing directory will be prepended to the file list entries prior to expansion.
2. If the passed file_list is NULL, the enclosing directory, name, or extension are not NULL, they will be incorporated into the a regex expansion. If name is NULL, it is replaced by "*" in the expansion; this is not true for enclosing_directory or extension.

The number of entries in the returned file_list is returned in n_out_files. Generate_file_list_m11() can be used to get a directory list also.

NOTE: If a file_list is passed should not be statically allocated, if free_input_file_list == TRUE_m11.

The path parts specify how the output file_list will be constructed.

Example 1:

```
file_list = generate_file_list_m11(NULL, 0, &n_out_files, "/Data/MED_Files/ \
      Session_1.medd", NULL, "ticd", PP_FULL_PATH_m11, FALSE_m11);
```

Returns list of all time series channel directories (extension == "ticd") in /Data/MED_Files/Session_1, as full paths.

Example 2:

```
file_list = generate_file_list_m11(NULL, 0, &n_out_files, "/Data/MED_Files/ \
      Session_1.medd", NULL, "ticd", PP_NAME_m11, FALSE_m11);
```

Returns list of all time series channel directories names in /Data/MED_Files/Session_1, as names only. Such a list could be used labels in a viewer, or to create a derivative MED session.

Example 3:

```
file_list[0] = “grid[1-3].ticd”;
file_list[1] = “depth[2,4,6].ticd”;
file_list[2] = “micro*”;
file_list[3] = “*.vicd”;

file_list = generate_file_list_m11(file_list, 4, &n_out_files, “/Data/MED_Files/ \
      Session_1.medd”, NULL, NULL, PP_NAME_m11 | PP_EXTENSION_m11, TRUE_m11);
```

Returns the following in file_list (you can imagine the full session directory contents):

```
grid1.ticd
grid2.ticd
grid3.ticd
depth2.ticd
depth4.ticd
depth6.ticd
micro_1.ticd
micro_2.ticd
camera_1.vicd
camera_2.vicd
```

On return, “n_out_files” will contain “10”, and the input file_list will have been freed.

FUNCTION: generate_hex_string_m11()

```
// Prototype
si1   *generate_hex_string_m11(ui1 *bytes, si4 num_bytes, si1 *string);

#define HEX_STRING_BYTES_m11(x)         ( ((x) + 1) * 3 )
```

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_m11(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_m11(pwd->level_1_encryption_key, ENCRYPTION_KEY_BYTES, \
      NULL);
printf("Level 1 Encryption Key: %s\n", hex_str);
free(hex_str);
```

FUNCTION: generate_MED_path_components_m11()

```
// Prototype
ui4   generate_MED_path_components_m11(si1 *path, si1 *MED_dir, si1 *MED_name);
```

Given a path to a MED directory or file, returns the full path to the enclosing directory in MED_dir, and the isolated name of the file (no path or extension) in MED_name, and the MED type code of the enclosing directory is the return value. If either MED_dir or MED_name are NULL, they will not be filled in.

The utility of this is to regularize a MED directory or file name into its path (including name and directory extension), and the isolated name. Thus:

generate_MED_path_components_m11(path_to_MED_file, MED_path, MED_name);

AND

generate_MED_path_components_m11(path_to_MED_directory, MED_path, MED_name);

… result in the same contents of MED_path and MED_name and the same returned type for the directory.

These components can be used to easily build other MED file names, given the naming conventions of the MED hierarchy.

FUNCTION: generate_recording_time_offset_m11()

```
// Constants
#define USE_SYSTEM_TIME -1

// Prototype
si8   generate_recording_time_offset_m11(si8 recording_start_time_uutc);
```

The function calculates the recording time offset from the passed recording_start_time_uutc and local time zone information. The recording time offset is stored in the MED globals. If recording_start_time_uutc equals USE_SYSTEM_TIME, the recording time offset will be based on the system time act the time of the function call,.

The function returns the recording time offset (useful only if USE_SYSTEM_TIME was passed).

FUNCTION: generate_segment_name_m11()

```
// Prototype
si1   *generate_segment_name_m11(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_UID_m11()

```
// Prototype
ui8   generate_UID_m11(ui8 *uid);
```

Generates an 8-byte random number that is not one of the MED reserved values. If NULL is passed for uid, this function will return a uid value, but is not thread-safe.

Examples:

```
generate_UUID(&universal_header->file_UID);

or

universal_header->file_UID = generate_UUID(NULL);  // not thread safe
```

FUNCTION: get_segment_range_m11()

```
// Prototype
si4   get_segment_range_m11(si1 **channel_list, si4 n_channels, TIME_SLICE_m11 *slice);

typedef struct {
      TERN_m11    conditioned;
      si8   start_time;
      si8   end_time;
      si8   start_sample_number;  // session-relative (global indexing)
      si8   end_sample_number;  // session-relative (global indexing)
      si8   local_start_sample_number;  // segment-relative (local indexing)
      si8   local_end_sample_number;  // segment-relative (local indexing)
      si8   number_of_samples;
      si4   start_segment_number;
      si4   end_segment_number;
      si8   session_start_time;
      si8   session_end_time;
      si1   sample_number_reference_channel_name[BASE_FILE_NAME_BYTES_m11];  // channel
base name
      si4   sample_number_reference_channel_index;  // index of the sample number
                                                    //reference channel in the session
channel array
} TIME_SLICE_m11;
```

Finds the segment start and en number for the range described in the passed time slice. idx_ref_chan is needed if the channels in the channel list have different sampling frequencies, and the time slice specifies sample numbers. idx_ref_chan is the name of the channel that should be used to decide cutoffs. Other members of the time slice will be filled in. The function returns the number of segments that need to be read.

See heading “Time Slice Usage" below.

FUNCTION: initialize_file_processing_directives_m11()

```
// Structures
typedef struct {
      TERN_m11    close_file;
      TERN_m11    flush_after_write;
      TERN_m11    update_universal_header;  // when writing
```

```
      TERN_m11    leave_decrypted;  // if encrypted during write, return from write
                                    // function decrypted, also leave times unoffset
      TERN_m11    free_password_data;  // when freeing FPS
      TERN_m11    free_CMP_processing_struct;  // when freeing FPS
      ui4         lock_mode;
      ui4         open_mode;
} FILE_PROCESSING_DIRECTIVES_m11;

// File Processing Directives Defaults
#define FPS_DIRECTIVE_CLOSE_FILE_DEFAULT_m11                    TRUE_m11
#define FPS_DIRECTIVE_FLUSH_AFTER_WRITE_DEFAULT_m11             TRUE_m11
#define FPS_DIRECTIVE_FREE_CMP_PROCESSING_STRUCT_DEFAULT_m11    TRUE_m11
#define FPS_DIRECTIVE_UPDATE_UNIVERSAL_HEADER_DEFAULT_m11       FALSE_m11
#define FPS_DIRECTIVE_LEAVE_DECRYPTED_DEFAULT_m11               FALSE_m11
#define FPS_DIRECTIVE_LOCK_MODE_DEFAULT_m11                     FPS_NO_LOCK_MODE_m11
                  // Unix file locking may cause problems with networked file systems
// #define FPS_DIRECTIVE_LOCK_MODE_DEFAULT_m11  (FPS_READ_LOCK_ON_READ_OPEN_m11 | \
      FPS_WRITE_LOCK_ON_WRITE_OPEN_m11 | FPS_WRITE_LOCK_ON_READ_WRITE_OPEN_m11)
#define FPS_DIRECTIVE_OPEN_MODE_DEFAULT_m11                     FPS_NO_OPEN_MODE_m11

// Prototype
FILE_PROCESSING_DIRECTIVES_m11
*initialize_file_processing_directives_m11(FILE_PROCESSING_DIRECTIVES_m11 *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_globals_m11()

```
// Prototype
void  initialize_globals_m11(void);
```

Allocates (if NULL) and initializes globals_m11, the MED globals structure on the global heap. Variables are initialized to their default values, and tables are assigned or constructed. The 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 UTC offset, and a verbose flag which if set will cause many library functions to show the output of their processing.

This function is called by initialize_medlib(), but can be called before initialize_medlib() to change a default value.

Example (used to see verbose output of initialization functions):

```
initialize_globals_m11();
globals_m11->verbose = TRUE_m11;
initialize_medlib();
```

FUNCTION: initialize_medlib_m11()

```
// Prototype
si4   initialize_medlib_m11(void);
```

Initializes MED_globals to default values (if the MED_globals pointer is NULL, which it is at the launch of the library), checks CPU endianness, checks MED 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 MED_TRUE if all structures are aligned, MED_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_medlib() == MED_FALSE) {
      fprintf(stderr, “error initializing medlib => exiting\n”)
      exit(1);
}
MED_globals->verbose = MED_TRUE;    // globals initialized by initialize_medlib(),
                              // default verbose setting is MED_FALSE
```

example 2:

```
initialize_MED_globals();           // globals will not be initialized by
initialize_medlib()
MED_globals->verbose = MED_TRUE;    // default verbose setting is MED_FALSE
initialize_medlib();
```

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

FUNCTION: initialize_metadata_m11()

```
// Prototype
void  initialize_metadata_m11(FILE_PROCESSING_STRUCT_m11 *fps, TERN_m11 \
      initialize_for_update);
```

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 channel type.

If initialize_for_update == TRUE_m11, the following section 2 fields are set to Zero rather than their No Entry values:

```
// time series, section 2
number_of_samples
number_of_blocks
maximum_block_byte
maximum_block_samples
maximum_block_difference_bytes
maximum_block_duration
number_of_discontinuities
maximum_contiguous_blocks
maximum_contiguous_block_bytes
maximum_contiguous_samples

// video, section 2
number_of_clips
maximum_clip_bytes
```

```
number_of_video_files
```

FUNCTION: initialize_time_slice_m11()

```
// Structure
typedef struct {
      TERN_m11    conditioned;
      si8   start_time;
      si8   end_time;
      si8   start_sample_number;  // session-relative (global indexing)
      si8   end_sample_number;  // session-relative (global indexing)
      si8   local_start_sample_number;  // segment-relative (local indexing)
      si8   local_end_sample_number;  // segment-relative (local indexing)
      si8   number_of_samples;
      si4   start_segment_number;
      si4   end_segment_number;
      si8   session_start_time;
      si8   session_end_time;
      si1   sample_number_reference_channel_name[BASE_FILE_NAME_BYTES_m11];  // channel
                                                                              // base name
      si4   sample_number_reference_channel_index;  // index of the index reference
                                          // channel in the session channel array
} TIME_SLICE_m11;

// Prototype
TIME_SLICE_m11  *initialize_time_slice_m11(TIME_SLICE_m11 *slice);
```

The function allocates (if NULL is passed) and initializes the contents of a time slice structure.

See heading “Time Slice Usage" below.

FUNCTION: initialize_universal_header()

```
// Prototype
si4   initialize_universal_header_m11(FILE_PROCESSING_STRUCT_m11 *fps, ui4 type_code, \
      TERN_m11 generate_file_UID, TERN_m11 originating_file);

// Universal Header Structure
typedef struct {
      // start robust mode region
      ui4   header_CRC;     // CRC of the universal header after this field
      ui4   body_CRC;       // CRC of the entire file after the universal header
      si8   file_end_time;
      si8   number_of_entries;
      ui4   maximum_entry_size;
      // end robust mode region
      si4   segment_number;
      union {  // anonymous union
```

```
            struct {
                  si1   type_string[TYPE_BYTES_m11];
                  ui1   MED_version_major;
                  ui1   MED_version_minor;
                  ui1   byte_order_code;
            };
            struct {
                  ui4   type_code;
                  si1   type_string_terminal_zero;  // not used - there for clarity
            };
      };
      si8   session_start_time;
      si8   file_start_time;
      si1   session_name[BASE_FILE_NAME_BYTES_m11];
      si1   channel_name[BASE_FILE_NAME_BYTES_m11];
      si1   anonymized_subject_ID[UNIVERSAL_HEADER_ANONYMIZED_SUBJECT_ID_BYTES_m11];
      ui8   session_UID;
      ui8   channel_UID;
      ui8   segment_UID;
      ui8   file_UID;
      ui8   provenance_UID;
      ui1   level_1_password_validation_field[PASSWORD_VALIDATION_FIELD_BYTES_m11];
      ui1   level_2_password_validation_field[PASSWORD_VALIDATION_FIELD_BYTES_m11];
      ui1   level_3_password_validation_field[PASSWORD_VALIDATION_FIELD_BYTES_m11];
      ui1   protected_region[UNIVERSAL_HEADER_PROTECTED_REGION_BYTES_m11];
      ui1   discretionary_region[UNIVERSAL_HEADER_DISCRETIONARY_REGION_BYTES_m11];
} UNIVERSAL_HEADER_m11;
```

The function sets universal header fields to default values. If generate_file_UID == TRUE_m11, it will be created. If originating_file == TRUE_m11, the provenance_UID will be set to the value of the file_UID. It fills in the current library's MED version and endianness.

example:

```
initialize_universal_header(fps, TIME_SERIES_METADATA_FILE_TYPE_CODE_m11, TRUE_m11,
TRUE_m11);
```

Initializes a time series metadata universal header with a new file UID, and the provenance UID set to the new file UID.

FUNCTION: MED_type_string_from_code_m11()

```
// Prototype
si1   *MED_type_string_from_code_m11(ui4 code);
```

Returns a pointer to a type string for all MED types, or NULL if the code does not exist.

FUNCTION: MED_type_code_from_string_m11()

```
// Prototype
ui4   MED_type_code_from_string_m11(si1 *string);
```

Returns a type code for all MED type strings, or NO_FILE_TYPE_CODE_m11 if no code exists for the passed string.

FUNCTION: merge_metadata_m11()

```
// Prototype
TERN_m11    merge_metadata_m11(FILE_PROCESSING_STRUCT_m11 *md_fps_1, \
            FILE_PROCESSING_STRUCT_m11 *md_fps_2, FILE_PROCESSING_STRUCT_m11 \
            *merged_md_fps);
```

If merged_md_fps == NULL, comparison results will be placed in md_fps_1->metadata.
Returns TRUE_m11 if md_fps_1->metadata == md_fps_2->metadata, FALSE_m11 otherwise.

FUNCTION: merge_universal_headers_m11()

```
// Prototype
TERN_m11    merge_universal_headers_m11(FILE_PROCESSING_STRUCT_m11 *fps_1, \
            FILE_PROCESSING_STRUCT_m11 *fps_2, FILE_PROCESSING_STRUCT_m11 *merged_fps);
```

If merged_fps == NULL, comparison results will be placed in fps_1->universal_header.
Returns TRUE_m11 if fps_1->universal_header == fps_2->universal_header, FALSE_m11 otherwise.

FUNCTION: message_m11()

```
// Prototype
void  message_m11(si1 *fmt, ...);
```

Prints a warning message to stdout unless:
(globals_m11->behavior_on_fail & SUPPRESS_MESSAGE_OUTPUT_m11) != 0
and returns. Used like printf(). Also see error_message_m11(), warning_message_m11(), and force_behavior_m11().

Example:

```
message_m11("%s(): Test Message (called from line %d)", __FUNCTION__, \
__LINE__);
```

FUNCTION: numerical_fixed_width_string_m11()

```
// Prototype
si1   *numerical_fixed_width_string_m11(si1 *string, si4 string_bytes, si4 number);
```

Returns string, prepended with necessary number of zeros, to make a string of string_bytes characters containing the digits of number. If NULL is passed for string, it is allocated, and up to the caller to free. Useful in generating MED segment names, among other things.

FUNCTION: pad_m11()

```
// Prototype
si8   pad_m11(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: process_password_data_m11()

```
// Prototype
TERN_m11    process_password_data_m11(si1 *unspecified_password, si1 \
      *L1_password, si1 *L2_password, si1 *L3_password, si1 *L1_hint, si1 *L2_hint, \
      FILE_PROCESSING_STRUCT_m11 *fps);

// Structures
typedef struct {
      ui1   level_1_encryption_key[ENCRYPTION_KEY_BYTES_m11];
      ui1   level_2_encryption_key[ENCRYPTION_KEY_BYTES_m11];
      si1   level_1_password_hint[PASSWORD_HINT_BYTES_m11];
      si1   level_2_password_hint[PASSWORD_HINT_BYTES_m11];
      ui1   access_level;
      ui1   processed;  // 0 or 1 (not ternary)
} PASSWORD_DATA_m11;
```

Fills global PASSWORD_DATA structure and set it to processed.

If no passwords are passed (all are NULL), it simply sets the PASSWORD_DATA structure to an access_level of zero (access to unencrypted data only).

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 MED files.

If a level_1_password, level_2_password, or level_3_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 MED 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 MED files.*

NOTE: If the FILE_PROCESSING_STRUCT corresponds to a metadata file, the password hints will be copied to the PASSWORD_DATA structure, otherwise they will be left empty. See set_time_and_password_data_m11() also.

The function also sets processed field to TRUE.

Example 1:

```
process_password_data(password, NULL, NULL, NULL, NULL, NULL, fps);
```

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:

```
process_password_data(NULL, level_1_password, level_2_password, \
```

```
NULL, NULL, NULL, fps);
```

In writing a new MED 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: read_channel_m11()

```
// Prototype
CHANNEL_m11 *read_channel_m11(CHANNEL_m11 *chan, si1 *chan_dir, TIME_SLICE_m11 *slice, \
      si1 *password, TERN_m11 read_time_series_data, TERN_m11 \
      read_record_data);

// Channel Types
UNKNOWN_CHANNEL_TYPE_m11      -1    // (NO_FILE_TYPE_CODE_m11)
TIME_SERIES_CHANNEL_TYPE_m11  1     // (TIME_SERIES_CHANNEL_DIRECTORY_TYPE_CODE_m11)
VIDEO_CHANNEL_TYPE_m11        2     // (VIDEO_CHANNEL_DIRECTORY_TYPE_CODE_m11)

// Structure
typedef struct {
      FILE_PROCESSING_STRUCT_m11    *metadata_fps;  // ephemeral
      FILE_PROCESSING_STRUCT_m11    *record_data_fps;
      FILE_PROCESSING_STRUCT_m11    *record_indices_fps;
      si4                           number_of_segments;
      SEGMENT_m11                   **segments;
      si1                           path[FULL_FILE_NAME_BYTES_m11];
      si1                           name[BASE_FILE_NAME_BYTES_m11];
      TIME_SLICE_m11                time_slice;
} CHANNEL_m11;
```

This function will read the channel pointed to by chan_dir (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. A password should be passed to read encrypted fields if the global password structure has not already been set. In the case that no data is encrypted or only unencrypted data is needed, NULL can be passed for both fields.

If read_time_series_data == TRUE_m11, the time series segment data specified by slice will be read into its SEGMENT structures; otherwise only the segment data's universal header will be read into this field and the file pointer will be left at the beginning of the CMP block data for random or sequential reading. The same is true for the read_record_data parameter.

The CHANNEL metadata structure is ephemeral, i.e is the same as those contained in a segment FPS, but is not associated with a real file, rather it is constructed via a merge of all the segment metadata structures. 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. Likewise the universal header of this FPS represents a merge of the metadata universal headers from the segments.

The function returns a pointer to a CHANNEL structure, which is allocated if NULL was passed for chan.

example:

```
(void) read_MED_channel(&session->channels[i], full_file_name, NULL, NULL, password,
MED_FALSE, MED_FALSE);
```

This call reads the channel specified by full_file_name into a preallocated CHANNEL structure. A password is passed. The read_time_series_data and read_record_data flags are set to MED_FALSE, so only the universal headers will be read in from the segment files. All segments will be read because slice was NULL.

See heading “Time Slice Usage" below.

FUNCTION: read_file_m11()

```
// Prototype
FILE_PROCESSING_STRUCT_m11    *read_file_m11(FILE_PROCESSING_STRUCT_m11 *fps, si1 \
      *full_file_name, si8 number_of_items, ui1 **data_ptr_ptr, si8 *items_read, si1 \
      *password, ui4 behavior_on_fail);

// Structures
typedef struct {
      TERN_m11                              mutex;
      si1                                   full_file_name[FULL_FILE_NAME_BYTES_m11];
      FILE                                  *fp;  // file pointer
      si4                                   fd;   // file descriptor
      si8                                   file_length;
      UNIVERSAL_HEADER_m11                  *universal_header;
      FILE_PROCESSING_DIRECTIVES_m11        directives;
      METADATA_m11                          *metadata;
      TIME_SERIES_INDEX_m11                 *time_series_indices;
      VIDEO_INDEX_m11                       *video_indices;
      ui1                                   *records;
      RECORD_INDEX_m11                      *record_indices;
      si8                                   raw_data_bytes;
      ui1                                   *raw_data;
      CMP_PROCESSING_STRUCT_m11             *cps;
} FILE_PROCESSING_STRUCT_m11;

typedef struct {
      TERN_m11    close_file;
      TERN_m11    flush_after_write;
      TERN_m11    update_universal_header;
      TERN_m11    leave_decrypted;
      TERN_m11    free_password_data;  // when freeing FPS
      TERN_m11    free_CMP_processing_struct;  // when freeing FPS
      ui4         lock_mode;
      ui4         open_mode;
} FILE_PROCESSING_DIRECTIVES_m11;

// Constants
#define FPS_FULL_FILE_m11                 -1
#define FPS_UNIVERSAL_HEADER_ONLY_m11     0
```

The function reads any MED file type, identified by its full path in full_file_name, (or the same field in a passed FPS), into a FILE_PROCESSING_STRUCT (FPS). If NULL is passed for the FPS one will be allocated. 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 FALSE_m11, the file will be left open, otherwise it will be closed after reading. The number_of_items parameter specifies how much of the file to read, how this translates into bytes will depend upon the file type.

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

If data_ptr_ptr is NULL data is read into the FPS's raw_data array. If *data_ptr_ptr is within the raw_data array boundaries, the data will be read into *data_ptr_ptr, and the array will be enlarged as necessary (modifying *data_ptr_ptr). If *data_ptr_ptr is custom, data will be read to there, but read_file_m11() will have no way of knowing if the target array is large enough, so this is the caller's responsibility.

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

Example:

```
segment->time_series_data_fps = read_file_m11(NULL, full_file_name, \
      UNIVERSAL_HEADER_ONLY_m11, NULL, NULL, NULL, password_data, USE_GLOBAL_BEHAVIOR);
```

Reads the time series data file pointed to by full_file_name. read_MED_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. Only the universal header read in as the number_of_items is 0 (== UNIVERSAL_HEADER_ONLY_m11). The file is closed after reading as the FPS directive's default for close_file is TRUE_m11.

FUNCTION: read_segment_m11()

```
// Prototype
SEGMENT_m11 *read_segment_m11(SEGMENT_m11 *seg, si1 *seg_dir, TIME_SLICE_m11 *slice, \
      si1 *password, TERN_m11 read_time_series_data, TERN_m11 \
      read_record_data);

// Structure
typedef struct {
      FILE_PROCESSING_STRUCT_m11    *metadata_fps;  // also used as prototype
      FILE_PROCESSING_STRUCT_m11    *time_series_data_fps;
      FILE_PROCESSING_STRUCT_m11    *time_series_indices_fps;
      FILE_PROCESSING_STRUCT_m11    *video_indices_fps;
      FILE_PROCESSING_STRUCT_m11    *record_data_fps;
      FILE_PROCESSING_STRUCT_m11    *record_indices_fps;
      FILE_PROCESSING_STRUCT_m11    *segmented_session_record_data_fps;
      FILE_PROCESSING_STRUCT_m11    *segmented_session_record_indices_fps;
      si1                           path[FULL_FILE_NAME_BYTES_m11];
      si1                           name[SEGMENT_BASE_FILE_NAME_BYTES_m11];
      TIME_SLICE_m11                time_slice;
} SEGMENT_m11;
```

This function will read the segment pointed to by seg_dir (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.

If read_time_series_data == TRUE_m11 (and it is a time series segment), the time series data will be read into the SEGMENT structure's data_fps raw_data according to the limits in slice, or the full file if slice is NULL; otherwise only the segment data's universal header will be read and the file will be left open with the file pointer pointing to the CMP blocks. If read_record_data == TRUE_m11, the same process is followed, but for segment records.

The function returns a pointer to the SEGMENT structure.

example:

```
SEGMENT                         *segment;

segment = read_MED_segment(NULL, full_file_name, NULL, NULL, MED_TRUE, MED_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 files, if present. This is an uncommon use for large data files as reading all of the data into memory is frequently impractical.

See heading "Time Slice Usage" below.

FUNCTION: read_session_m11()

```
// Prototype
SESSION_m11 *read_session_m11(si1 *sess_dir, si1 **chan_list, si4 n_chans, \
      TIME_SLICE_m11 *slice,  si1 *idx_ref_chan, si1 *password, \
      TERN_m11 read_time_series_data, TERN_m11 read_record_data);

typedef struct {
      FILE_PROCESSING_STRUCT_m11    *time_series_metadata_fps;  // ephemeral
      FILE_PROCESSING_STRUCT_m11    *video_metadata_fps;  // ephemeral
      si4                           number_of_segments;
      si4                           number_of_time_series_channels;
      CHANNEL_m11                   **time_series_channels;
      si4                           number_of_video_channels;
      CHANNEL_m11                   **video_channels;
      FILE_PROCESSING_STRUCT_m11    *record_data_fps;
      FILE_PROCESSING_STRUCT_m11    *record_indices_fps;
      FILE_PROCESSING_STRUCT_m11    **segmented_record_data_fps;
      FILE_PROCESSING_STRUCT_m11    **segmented_record_indices_fps;
      si1                           path[FULL_FILE_NAME_BYTES_m11];
      si1                           name[BASE_FILE_NAME_BYTES_m11];
      TIME_SLICE_m11                time_slice;
} SESSION_m11;
```

This function will read all the files associated with the session pointed to by sess_dir (full path to the session directory) , or the channels listed in chan_list, 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.

If read_time_series_data == TRUE_m11 (for time series segments), the time series data will be read into the SEGMENT structure's data_fps raw_data according to the limits in slice, or the full file if slice is NULL; otherwise only the segment data's universal header will be read and the file will be left open with the file pointer pointing to the CMP blocks. If read_record_data == TRUE_m11, the same process is followed, but for segment records.

The function returns a pointer to the SEGMENT structure.

Example:

```
SESSION          *session;

session = read_session_m11(NULL, session_directory, NULL, 0, NULL, NULL, password, NULL,
MED_FALSE, MED_FALSE);
SESSION_m11 *read_session_m11(si1 *sess_dir, si1 **chan_list, si4 n_chans, \
      TIME_SLICE_m11 *slice,  si1 *idx_ref_chan, si1 *password, \
      TERN_m11 read_time_series_data, TERN_m11 read_record_data);
```

This call will allocate a SESSION structure and read all files associated with the MED session specific by sess_dir 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, but 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.

See heading “Time Slice Usage" below.

FUNCTION: read_time_series_data_m11()

```
// Prototype
si8   read_time_series_data_m11(SEGMENT_m11 *seg, si8 local_start_idx, \
      si8 local_end_idx, TERN_m11 alloc_cps);
```

This function reads a sample range from a segment. The parameters local_start_idx and local_end_idx are segment relative. The data are decompressed to seg->time_series_data_fps->cps->decompressed_ptr, and this pointer is updated.

It returns the number of samples read, which may be less than (local_end_idx - local_start_idx + 1) if either exceeds the segment boundaries.

FUNCTION: reallocate_file_processing_struct_m11()

```
// Prototype
si4   reallocate_file_processing_struct_m11(FILE_PROCESSING_STRUCT_m11 *fps, si8 \
      new_raw_data_bytes);
```

This function reallocates the raw_data array to new_raw_data_bytes bytes. 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: recover_passwords_m11()

```
// Prototype
void    recover_passwords_m11(si1 *L3_password, UNIVERSAL_HEADER_m11 *universal_header);
```

This function will recover level 1 & level 2 passwords if they were created with a level 3 password, and it is passed.

FUNCTION: remove_recording_time_offset_m11()

```
// Prototype
inline void remove_recording_time_offset_m11(si8 *time);
```

The global recording time offset is removed from the passed µUTC time.

FUNCTION: sample_number_for_uutc_m11()

```
// Target Value Constants
#define DEFAULT_MODE_m11        0
#define FIND_START_m11          1
#define FIND_END_m11            2
#define FIND_CENTER_m11         3
#define FIND_CURRENT_m11        4
#define FIND_NEXT_m11           5
#define FIND_CLOSEST_m11        6

// Prototype
si8   sample_number_for_uutc_m11(si8 ref_sample_number, si8 ref_uutc, si8 target_uutc,
sf8 sampling_frequency, FILE_PROCESSING_STRUCT_m11 *time_series_indices_fps, ui1 mode);
```

Return a sample number for a given uutc. Mode should be a member of { FIND_CURRENT_m11, FIND_CLOSEST_m11, FIND_NEXT_m11 }.

FIND_CURRENT_m11 (default): sample period within which the target_uutc falls
FIND_CLOSEST_m11: sample number closest to the target_uutc
FIND_NEXT_m11: sample number following the sample period within which the target_uutc falls

(FIND_NEXT_m11 == FIND_CURRENT_m11 + 1)

Examples:

1) Return sample number extrapolated from ref_sample_number
   ```
   sample_number_for_uutc_m11(ref_sample_number, ref_uutc, target_uutc,
   sampling_frequency, NULL, 0, mode);
   ```

2) Return sample number extrapolated from closest time series index in local (segment-relative) sample numbering
   ```
   sample_number_for_uutc_m11(SAMPLE_NUMBER_NO_ENTRY_m11, UUTC_NO_ENTRY_m11, \
   target_uutc, sampling_frequency, tsi, number_of_indices, mode);
   ```

3) Return sample number extrapolated from closest time series index in absolute (channel-relative) sample numbering. In this case ref_sample_number is the segment absolute start sample number.
   ```
   sample_number_for_uutc_m11(ref_sample_number, UUTC_NO_ENTRY_m11, \
   target_uutc, sampling_frequency, tsi, number_of_indices, mode);
   ```

FUNCTION: search_segment_metadata_m11()

```
// Prototype
TERN_m11    search_segment_metadata_m11(si1 *MED_dir, TIME_SLICE_m11 *slice);
```

Used to find segment range encompassed by slice when Sgmt records do not exist *(plug: they are optional, but tiny, and make this process much more efficient)*. Returns TRUE_m11 on success, FALSE_m11 on failure.

See heading “Time Slice Usage" below.

FUNCTION: search_Sgmt_records_m11()

```
// Prototype
TERN_m11    search_Sgmt_records_m11(si1 *MED_dir, TIME_SLICE_m11 *slice);
```

Used to find segment range encompassed by slice *(plug: Sgmt records are optional, but tiny, and make this process much more efficient)*. Returns TRUE_m11 on success, FALSE_m11 on failure.

See heading “Time Slice Usage" below.

FUNCTION: segment_sample_number_to_time_m11()

```
// Prototype
si8     segment_sample_number_to_time_m11(si1 *seg_dir, si8 local_sample_number, si8
absolute_start_sample_number, sf8 sampling_frequency, ui1 mode)
```

For a given segment, this function returns a µUTC time for a given sample index. If absolute_start_sample_number == SAMPLE_NUMBER_NO_ENTRY_m11 or sampling_frequency == FREQUENCY_NO_ENTRY_m11, the function will get these values from the segment metadata file. If they are known, passing them makes the function more efficient.

The returned value is an absolute index, numbered relative to the session, not the segment.

FUNCTION: set_global_time_constants_m11()

```
// Structure
typedef struct {
      si1   country[METADATA_RECORDING_LOCATION_BYTES_m11];
      si1   country_acronym_2_letter[3]; // two-letter acronym; (ISO 3166 ALPHA-2)
      si1   country_acronym_3_letter[4]; // three-letter acronym (ISO-3166 ALPHA-3)
      si1   territory[METADATA_RECORDING_LOCATION_BYTES_m11];
      si1   territory_acronym[TIMEZONE_STRING_BYTES_m11];
      si1   standard_timezone[TIMEZONE_STRING_BYTES_m11];
      si1   standard_timezone_acronym[TIMEZONE_ACRONYM_BYTES_m11];
      si4   standard_UTC_offset; // seconds
      si4   observe_DST;
      si1   daylight_timezone[TIMEZONE_STRING_BYTES_m11];
      si1   daylight_timezone_acronym[TIMEZONE_ACRONYM_BYTES_m11];
      si1   daylight_time_start_description[METADATA_RECORDING_LOCATION_BYTES_m11];
      si8   daylight_time_start_code;  // DAYLIGHT_TIME_CHANGE_CODE_m11
      si1   daylight_time_end_description[METADATA_RECORDING_LOCATION_BYTES_m11];
```

```
      si8   daylight_time_end_code;  // DAYLIGHT_TIME_CHANGE_CODE_m11
} TIMEZONE_INFO_m11;

// Prototype
void  set_global_time_constants_m11(TIMEZONE_INFO_m11 *timezone_info, si8
session_start_time);
```

Uses information in timezone_info to find the matching entry in the global timezone table, and then set time globals accordingly. Only enough information to make a solitary match is necessary. If there are multiple possible matches, a prompt dialog is initiated. If a value is passed for session_start_time (I.e. not zero), the global recording_time_offset is set also.

session_start_time parameter:
- pass zero to just match the timezone info (e.g. to get the standard_UTC_offset based on locale)
    - global recording_time_offset is not set
    - set global recording_time_offset by calling generate_recording_time_offset_m11() separately after generating the session_start_time parameter
- pass CURRENT_TIME_m11 to use the current time to generate the global recording_time_offset
- pass an unoffset µUTC to use that to generate the global recording_time_offset

FUNCTION: set_time_and_password_data_m11()

```
// Prototype
TERN_m11 set_time_and_password_data_m11(si1 *unspecified_password, si1 \
      *MED_directory, si1 *section_2_encryption_level, si1 *section_3_encryption_level);
```

Finds the closest metadata file given a MED directory, and reads it using the unspecified password. The metadata is used to fill in the global time data, and the unspecified password is used, in conjunction with the metadata file, to fill the global a password data structure. The metadata section 2 & 3 encryption levels are also returned. The metadata file is closed and freed. This function is useful when beginning to read a series of files in a MED session, as these functions need only be performed once. TRUE is returned on success, FALSE on failure.

NOTE: reading any metadata file with read_file_m11(),and then freeing it, will accomplish the same thing as this function.

FUNCTION: show_file_processing_struct_m11()

```
// Prototype
void  show_file_processing_struct_m11(FILE_PROCESSING_STRUCT_m11 *fps);

// Structures
typedef struct {
      TERN_m11                          mutex;
      si1                               full_file_name[FULL_FILE_NAME_BYTES_m11];
      FILE                              *fp;    // file pointer
      si4                               fd;     // file descriptor
      si8                               file_length;
      UNIVERSAL_HEADER_m11              *universal_header;
      FILE_PROCESSING_DIRECTIVES_m11    directives;
      PASSWORD_DATA_m11                 *password_data;
      METADATA_m11                      *metadata;
      TIME_SERIES_INDEX_m11             *time_series_indices;
      VIDEO_INDEX_m11                   *video_indices;
```

```
      ui1                                 *records;
      RECORD_INDEX_m11                    *record_indices;
      si8                                 raw_data_bytes;
      ui1                                 *raw_data;
      CMP_PROCESSING_STRUCT_m11           *cps;
} FILE_PROCESSING_STRUCT_m11;
```

Displays all the elements of a FILE_PROCESSING_STRUCT_m11 structure.

FUNCTION: show_globals_m11()

```
// Prototype
void  show_globals_m11();
```

Displays MED globals.

FUNCTION: show_metadata_m11()

```
// Structure
typedef struct {
      METADATA_SECTION_1_m11                    section_1;
      union {
            TIME_SERIES_METADATA_SECTION_2_m11  time_series_section_2;
            VIDEO_METADATA_SECTION_2_m11        video_section_2;
      };
      METADATA_SECTION_3_m11                    section_3;
} METADATA_m11;

// Prototype
void  show_metadata_m11(FILE_PROCESSING_STRUCT_m11 *fps, METADATA_m11 *md);
```

Displays all the elements of a METADATA structure of the type specified by the passed FPS or METADATA structure pointer. Only one of the two parameters needs to be passed. The other can be NULL.

FUNCTION: show_password_data_m11()

```
// Structures
typedef struct {
      ui1   level_1_encryption_key[ENCRYPTION_KEY_BYTES_m11];
      ui1   level_2_encryption_key[ENCRYPTION_KEY_BYTES_m11];
      si1   level_1_password_hint[PASSWORD_HINT_BYTES_m11];
      si1   level_2_password_hint[PASSWORD_HINT_BYTES_m11];
      ui1   access_level;
      ui1   processed;  // 0 or 1 (not ternary)
} PASSWORD_DATA_m11;

// Prototype
void  show_password_data_m11(FILE_PROCESSING_STRUCT_m11 *fps);
```

Displays all the elements of a PASSWORD_DATA structure. Only one of the two parameters needs to be passed; the other can be NULL.

FUNCTION: show_records_m11()

```
// Constants
#define UNKNOWN_NUMBER_OF_ENTRIES_m11     -1
#define ALL_TYPES_CODE_m11          (ui4) 0xFFFFFFFF

// Prototype
void  show_records_m11(FILE_PROCESSING_STRUCT_m11 *fps, ui4 type_code);
```

This function displays the contents of the records data file of type type_code, or all records if type_code == ALL_TYPES_CODE_m11. If the record needs to be decrypted and the access level is sufficient, the record will be decrypted. show_records_m11() calls show_record_m11() for each record to be displayed. show_record_m11() resides in the medrec_m11.c file. The number of records is read from the universal header's number_of_entries field. If that field contains UNKNOWN_NUMBER_OF_ENTRIES_m11, the function will still work (but could fail in the case of an incomplete terminal record).

FUNCTION: show_time_slice_m11()

```
// Prototype
void  show_time_slice_m11(TIME_SLICE_m11 *slice);
```

This function displays the contents of a time slice.

FUNCTION: show_timezone_info_m11()

```
// Structure
typedef struct {
      si1   country[METADATA_RECORDING_LOCATION_BYTES_m11];
      si1   country_acronym_2_letter[3]; // (ISO 3166 ALPHA-2)
      si1   country_acronym_3_letter[4]; // (ISO-3166 ALPHA-3)
      si1   territory[METADATA_RECORDING_LOCATION_BYTES_m11];
      si1   territory_acronym[TIMEZONE_STRING_BYTES_m11];
      si1   standard_timezone[TIMEZONE_STRING_BYTES_m11];
      si1   standard_timezone_acronym[TIMEZONE_ACRONYM_BYTES_m11];
      si4   standard_UTC_offset; // seconds
      si4   observe_DST;
      si1   daylight_timezone[TIMEZONE_STRING_BYTES_m11];
      si1   daylight_timezone_acronym[TIMEZONE_ACRONYM_BYTES_m11];
      si1   daylight_time_start_description[METADATA_RECORDING_LOCATION_BYTES_m11];
      si8   daylight_time_start_code;  // DAYLIGHT_TIME_CHANGE_CODE_m11 as si8
      si1   daylight_time_end_description[METADATA_RECORDING_LOCATION_BYTES_m11];
      si8   daylight_time_end_code;  // DAYLIGHT_TIME_CHANGE_CODE_m11 as si8
} TIMEZONE_INFO_m11;
```

```
// Prototype
void  show_timezone_info_m11(TIMEZONE_INFO_m11 *timezone_entry);
```

This function displays the contents of a TIMEZONE_INFO_m11 structure.

FUNCTION: show_universal_header_m11()

```
// Structure
typedef struct {
      // start robust mode region
      ui4   header_CRC;     // CRC of the universal header after this field
      ui4   body_CRC;       // CRC of the entire file after the universal header
      si8   file_end_time;
      si8   number_of_entries;
      ui4   maximum_entry_size;
      // end robust mode region
      si4   segment_number;
      union {  // anonymous union
            struct {
                  si1   type_string[TYPE_BYTES_m11];
                  ui1   MED_version_major;
                  ui1   MED_version_minor;
                  ui1   byte_order_code;
            };
            struct {
                  ui4   type_code;
                  si1   type_string_terminal_zero;  // not used - there for clarity
            };
      };
      si8   session_start_time;
      si8   file_start_time;
      si1   session_name[BASE_FILE_NAME_BYTES_m11];
      si1   channel_name[BASE_FILE_NAME_BYTES_m11];
      si1   anonymized_subject_ID[UNIVERSAL_HEADER_ANONYMIZED_SUBJECT_ID_BYTES_m11];
      ui8   session_UID;
      ui8   channel_UID;
      ui8   segment_UID;
      ui8   file_UID;
      ui8   provenance_UID;
      ui1   level_1_password_validation_field[PASSWORD_VALIDATION_FIELD_BYTES_m11];
      ui1   level_2_password_validation_field[PASSWORD_VALIDATION_FIELD_BYTES_m11];
      ui1   level_3_password_validation_field[PASSWORD_VALIDATION_FIELD_BYTES_m11];
      ui1   protected_region[UNIVERSAL_HEADER_PROTECTED_REGION_BYTES_m11];
      ui1   discretionary_region[UNIVERSAL_HEADER_DISCRETIONARY_REGION_BYTES_m11];
} UNIVERSAL_HEADER_m11;
```

```
// Prototype
void  show_universal_header_m11(FILE_PROCESSING_STRUCT_m11 *fps, UNIVERSAL_HEADER_m11
*uh);
```

This function displays the contents of a UNIVERSAL_HEADER_m11 structure. Only one of the two parameters needs to be passed; the other can be NULL.

SECTION: Time Slice Usage

```
// Structure
typedef struct {
      TERN_m11    conditioned;
      si8   start_time;
      si8   end_time;
      si8   start_sample_number;  // session-relative (global indexing)
      si8   end_sample_number;  // session-relative (global indexing)
      si8   local_start_sample_number;  // segment-relative (local indexing)
      si8   local_end_sample_number;  // segment-relative (local indexing)
      si8   number_of_samples;
      si4   start_segment_number;
      si4   end_segment_number;
      si8   session_start_time;
      si8   session_end_time;
      si1   sample_number_reference_channel_name[BASE_FILE_NAME_BYTES_m11];  // channel
                                                                              // base name
      si4   sample_number_reference_channel_index;  // index of the index reference
                                          // channel in the session channel array
} TIME_SLICE_m11;

// Constants
#define BEGINNING_OF_SAMPLE_NUMBERS_m11   ((si8) 0x0000000000000000)
#define END_OF_SAMPLE_NUMBERS_m11         ((si8) 0x7FFFFFFFFFFFFFFF)
#define SAMPLE_NUMBER_NO_ENTRY_m11        ((si8) 0x8000000000000000)
#define UUTC_NO_ENTRY_m11                 ((si8) 0x8000000000000000)
#define UUTC_EARLIEST_TIME_m11            ((si8) 0x0000000000000000)
                                          // 00:00:00.000000 Thursday, 1 Jan 1970, UTC
#define UUTC_LATEST_TIME_m11              ((si8) 0x7FFFFFFFFFFFFFFF)
                                          // 04:00:54.775808 Sunday, 10 Jan 29424, UTC
#define BEGINNING_OF_TIME_m11             UUTC_EARLIEST_TIME_m11
#define END_OF_TIME_m11                   UUTC_LATEST_TIME_m11
```

The time slice is used throughout the library to pass a set of parameters that describe a portion of time that occurred within a MED session. The slice will accept either times or indices as boundaries. The search functions fill in the unfilled values, i.e. if times are passed, samples will be returned. The boundary values themselves might also be modified, if for instance the session end time occurs before the requested time.

To begin with the first sample in a session:
`slice->start_time = BEGINNING_OF_TIME_m11;` **…AND…** `slice->start_sample_number = SAMPLE_NUMBER_NO_ENTRY_m11;`
…OR…
`slice->start_sample_number = BEGINNING_OF_SAMPLE_NUMBERS_m11;` **…AND…** `slice->start_time = UUTC_NO_ENTRY_m11;`

To finish with the last sample in a session:
`slice->end_time = END_OF_TIME_m11;` **…AND…** `slice->end_sample_number = SAMPLE_NUMBER_NO_ENTRY_m11;`

…OR…
`slice->end_sample_number = END_OF_SAMPLE_NUMBERS_m11;` **…AND…** `slice->end_time = UUTC_NO_ENTRY_m11;`

NOTE: If indices are passed as the search medium, and the sampling frequency varies between channels in the selected channel set, an **index_reference_channel_name** must be supplied. This is the channel to which the passed indices refer. The other channels will be returned based on the points in time that the passed indices refer to in the index reference channel. If no index reference channel is passed when one is required, a warning is displayed and the first channel in the channel set will be used as the index reference channel.

The functions handle offset and un-offset times without modification. If the times are < 0, they are considered to be microseconds **relative to the session start time**. E.g. to get the first second of data: `slice->start_time = 0;` `slice->end_time = 999999`. These times are considered relative. On return, the times will be replaced by absolute times (offset applied).

FUNCTION: time_string_m11()

```
// Text Color String Constants
#define TC_BLACK_m11            "\033[30m"
#define TC_RED_m11              "\033[31m"
#define TC_GREEN_m11            "\033[32m"
#define TC_YELLOW_m11           "\033[33m"
#define TC_BLUE_m11             "\033[34m"
#define TC_MAGENTA_m11          "\033[35m"
#define TC_CYAN_m11             "\033[36m"
#define TC_WHITE_m11            "\033[37m"
#define TC_BRIGHT_BLACK_m11     "\033[30;1m"
#define TC_BRIGHT_RED_m11       "\033[31;1m"
#define TC_BRIGHT_GREEN_m11     "\033[32;1m"
#define TC_BRIGHT_YELLOW_m11    "\033[33;1m"
#define TC_BRIGHT_BLUE_m11      "\033[34;1m"
#define TC_BRIGHT_MAGENTA_m11   "\033[35;1m"
#define TC_BRIGHT_CYAN_m11      "\033[36;1m"
#define TC_BRIGHT_WHITE_m11     "\033[37;1m"
#define TC_RESET_m11            "\033[0m"

// Prototype
si1   *time_string_m11(si8 uutc, si1 *time_str, TERN_m11 fixed_width, TERN_m11 \
relative_days, si4 colored_text, ...);

// time_str buffer should be of length TIME_STRING_BYTES_m11;
```

Returns a string with local date and time from a µUTC time. If time_str is not NULL it will be used for the return string; if not a pointer to a static string is returned, which is not thread safe.

If uutc == 0, the current time will be returned.

If fixed_width == TRUE_m11, the string will be formatted so that all times have the same width.

If relative_days == TRUE_m11, the string date will be displayed as days from session start; first day is “Day 0001”. This is often a more intuitive format, particularly for offset times, which otherwise would display day 1 as “January 1, 1970”.

If colored_text == TRUE_m11, two more arguments are expected: date_color & time_color. The arguments should be one of the defined text color strings (above).

If a recording time offset is applied the standard timezone acronym is set to “oUTC”, and the standard timezone string is set to "offset Coordinated Universal Time”.

If Daylight Saving Time (DST) was in effect at the passed time, in the recording locale, it will be applied and reflected in the timezone acronym and string.

FUNCTION: unescape_spaces_m11()

```
// Prototype
void  unescape_spaces_m11(si1 *string);
```

This function removes backslashes occurring before spaces in the passed string.

FUNCTION: uutc_for_sample_number_m11()

```
// Prototype
si8   uutc_for_sample_number_m11(si8 ref_sample_number, si8 ref_uutc, si8 \
      target_sample_number, sf8 sampling_frequency, FILE_PROCESSING_STRUCT_m11 \
      *time_series_indices_fps, ui1 mode);

// Target Value Constants
#define DEFAULT_MODE_m11        0
#define FIND_START_m11          1
#define FIND_END_m11            2
#define FIND_CENTER_m11         3
#define FIND_CURRENT_m11        4
#define FIND_NEXT_m11           5
#define FIND_CLOSEST_m11        6
```

Return a µUTC for a given sample number. Mode should be a member of { FIND_START_m11, FIND_END_m11, FIND_CENTER_m11 }.

Sample time is defined as the period from sample onset until the next sample.
FIND_START_m11 (default): first uutc >= start of target_sample_number period
FIND_END_m11: last uutc < start of next sample period
FIND_CENTER_m11: uutc closest to the center of the sample period

Examples:
1) Return uutc extrapolated from ref_uutc with sample numbering in context of ref_sample_number & target_sample_number
   ```
   uutc = uutc_for_sample_number_m11(ref_sample_number, ref_uutc, target_sample_number, \
          sampling_frequency, NULL, mode);
   ```

2) Return uutc extrapolated from ref_uutc, assumed to occur at sample number 0, with local (segment-relative) sample numbering
   ```
   uutc = uutc_for_sample_number_m11(SAMPLE_NUMBER_NO_ENTRY_m11, ref_uutc, \
          target_sample_number, sampling_frequency, NULL, mode);
   ```

3) Return uutc extrapolated from ref_uutc with local (segment-relative) sample numbering
   ```
   uutc = uutc_for_sample_number_m11(SAMPLE_NUMBER_NO_ENTRY_m11, UUTC_NO_ENTRY_m11, \
   ```

```
        target_sample_number, sampling_frequency, time_series_indices_fps, mode);
```

4) Return uutc extrapolated from closest time series index using absolute sample numbering

```
   uutc = uutc_for_sample_number_m11(ref_sample_number, UUTC_NO_ENTRY_m11, \
        target_sample_number, sampling_frequency, time_series_indices_fps, mode);
```

FUNCTION: validate_record_data_CRCs_m11()

```
// Prototype
TERN_m11    validate_record_data_CRCs_m11(RECORD_HEADER_m11 *record_header, si8 \
            number_of_items);
```

The function validates the record header CRCs for number_of_items records pointed to by record_header. Returns TRUE_m11 if all CRCs are valid, FALSE_m11 otherwise.

FUNCTION: validate_time_series_data_CRCs_m11()

```
// Prototype
TERN_m11    validate_time_series_data_CRCs_m11(CMP_BLOCK_FIXED_HEADER_m11 \
            *block_header, si8 number_of_items);
```

The function validates the CMP block header CRCs for number_of_items CMP blocks pointed to by block_header. Returns TRUE_m11 if all CRCs are valid, FALSE_m11 otherwise.

FUNCTION: warning_message_m11()

```
// Prototype
void  warning_message_m11(si1 *fmt, ...);
```

Prints a warning message to stderr unless:
(globals_m11->behavior_on_fail & SUPPRESS_WARNING_OUTPUT_m11) != 0
and returns. Used like fprintf(). Also see error_message_m11(), message_m11(), and force_behavior_m11().

Example:

```
warning_message_m11("%s(): No samples in block (called from line %d)\n", __FUNCTION__, \
__LINE__);
```

```
/************************************************************************************/
/********************************** CMP Functions ***********************************/
/************************************************************************************/

// Structures
typedef struct {
      // CMP block header fixed region start
      ui8     block_start_UID;
      ui4     block_CRC;
```

```
      ui4     block_flags;
      si8     start_time;
      si4     acquisition_channel_number;
      ui4     total_block_bytes;
      // CMP block encryption start
      ui4     number_of_samples;
      ui2     number_of_records;
      ui2     record_region_bytes;
      ui4     parameter_flags;
      ui2     parameter_region_bytes;
      ui2     protected_region_bytes;
      ui2     discretionary_region_bytes;
      ui2     model_region_bytes;
      ui4     total_header_bytes;
      // CMP block header variable region start
} CMP_BLOCK_FIXED_HEADER_m11;

typedef struct {
      ui4         mode;  // CMP_COMPRESSION_MODE_m11, CMP_DECOMPRESSION_MODE_m11
      ui4         algorithm;  // RED, PRED, or MBE
      si1         encryption_level;  // encryption level for data blocks:
                                     // passed in compression
      TERN_m11    fall_through_to_MBE;  // if MBE would be smaller than RED/PRED,
                                        // use MBE for that block
      TERN_m11    reset_discontinuity;  // if discontinuity directive == TRUE_m11,
                                        // reset to FALSE_m11 after compressing the
                                        // block
      TERN_m11    include_noise_scores;
      TERN_m11    no_zero_counts;  // in RED & PRED codecs (when blocks must be
                                   // encoded with non-block statistics. This is a
                                   // special case.)
      TERN_m11    free_password_data;  // when freeing CPS
      TERN_m11    set_derivative_level;  // value passed in "derivative_level"
                                         // parameter
      TERN_m11    find_derivative_level;  // mutually exclusive with
                                          // “set_derivative_level”
      // lossy compression directives
      TERN_m11    detrend_data;  // Lossless operation, but most useful for lossy
                                 // compression.
      TERN_m11    require_normality;  // For lossy compression - use lossless if data
                                      // amplitudes are too oddly distributed.  Pairs
                                      // with "minimum_normality" parameter.
      TERN_m11    use_compression_ratio;  // Used in "find" directives. Mutually
                                          // exclusive with
                                          // “use_mean_residual_ratio”.
      TERN_m11    use_mean_residual_ratio;  // Used in "find" directives. Mutually
                                            // exclusive with
                                            // “use_compression_ratio”.
      TERN_m11    use_relative_ratio;  // divide goal ratio by the block coefficient
                                       // of variation in lossy compression routines
```

```
                                        // (more precision in blocks with higher
                                        // variance)
      TERN_m11    set_amplitude_scale;  // value passed in "amplitude_scale"
                                        // parameter
      TERN_m11    find_amplitude_scale;  // mutually exclusive with
                                         // “set_amplitude_scale”
      TERN_m11    set_frequency_scale;  // value passed in "frequency_scale"
                                        // parameter
      TERN_m11    find_frequency_scale;  // mutually exclusive with
                                         // “set_frequency_scale”
} CMP_DIRECTIVES_m11;

typedef struct {
      si4         number_of_block_parameters;
      ui4         *block_parameters;  // pointer beginning of parameter region of block
                                      // header
      ui4         block_parameter_map[CMP_PF_PARAMETER_FLAG_BITS_m11];
      si4         minimum_sample_value;  // found on compression, stored for use in
                                         // METADATA (and MBE, if used)
      si4         maximum_sample_value;  // stored for use in METADATA (and MBE, if used)
      TERN_m11    discontinuity;  // set if block is first after a discontinuity, passed
                                  // in compression, returned in decompression
      ui1         no_zero_counts_flag;
      ui1         derivative_level;  // used by with set_derivative_level directive, also
                                     // returned in decode
      // lossy compression parameters
      sf8         goal_ratio;  // either compression ratio or mean residual ratio
      sf8         actual_ratio;  // either compression ratio or mean residual ratio
      sf8         goal_tolerance;  // tolerance for lossy compression mode goal, value of
                                   // <= 0.0 uses default values, which are returned
      si4         maximum_goal_attempts;  // maximum loops to attain goal compression
      ui1         minimum_normality;
      sf4         amplitude_scale;  // used with set_amplitude_scale directive
      sf4         frequency_scale;  // used with set_frequency_scale directive
      ui2         user_number_of_records;
      ui2         user_record_region_bytes;  // set by user to reserve bytes for records
                                             // in header
      ui4         user_parameter_flags;  // user bits to be set in parameter flags of
                                         // block header (library flags will be set
                                         // automatically)
      ui2         protected_region_bytes;  // not currently used
      ui2         user_discretionary_region_bytes;  // set by user to reserve bytes for
                                                    // discretionary region in header
} CMP_COMPRESSION_PARAMETERS_m11;


typedef struct {
      TERN_m11    mutex;
      ui4         **count;  // used by RED/PRED encode & decode
      CMP_STATISTICS_BIN_m11  **sorted_count;  // used by RED/PRED encode & decode
```

```
      ui8         **cumulative_count;  // used by RED/PRED encode & decode
      ui8         **minimum_range;  // used by RED/PRED encode & decode
      ui1         **symbol_map;  // used by RED/PRED encode & decode
      si4         *input_buffer;
      ui1         *compressed_data;  // passed in decompression, returned in compression,
                                     // should not be updated
      CMP_BLOCK_FIXED_HEADER_m11    *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
      si1         *difference_buffer;  // passed in both compression & decompression
      si1         *derivative_buffer;  // used if needed in compression & decompression,
                                       // size of maximum block differences
      si4         *detrended_buffer;  // used if needed in compression, size of
                                      // decompressed block
      si4         *scaled_amplitude_buffer;  // used if needed in compression, size of
                                             // decompressed block
      si4         *scaled_frequency_buffer;  // used if needed in compression, size of
                                             // decompressed block
      CMP_DIRECTIVES_m11      directives;
      ui1         *records;
      CMP_PARAMETERS_m11      parameters;
      ui1         *protected_region;
      ui1         *discretionary_region;
      ui1         *model_region;
      PASSWORD_DATA_m11 *password_data;
} CMP_PROCESSING_STRUCT_m11;

// Macros
#define CMP_MAX_DIFFERENCE_BYTES_m11(block_samps)     // full si4 plus 1 keysample flag
                                                      // byte per sample
#define CMP_MAX_COMPRESSED_BYTES_m11(block_samps, n_blocks) // (no compression + header +
                                                            // maximum pad bytes) for
                                                            // n_blocks blocks
// NOTE: does not take variable region bytes into account and assumes fall through to MBE
#define CMP_IS_DETRENDED_m11(block_header_ptr)
#define CMP_VARIABLE_REGION_BYTES_v1_m11(block_header_ptr) // does not require
                                                           // total_header_bytes
#define CMP_VARIABLE_REGION_BYTES_v2_m11(block_header_ptr) // requires total_header_bytes

// Compression Modes
#define CMP_COMPRESSION_MODE_NO_ENTRY_m11       ((ui1) 0)
#define CMP_DECOMPRESSION_MODE_m11              ((ui1) 1)
```

```
#define CMP_COMPRESSION_MODE_m11                ((ui1) 2)

// Lossy Compression Modes
#define CMP_AMPLITUDE_SCALE_MODE_m11            ((ui1) 1)
#define CMP_FREQUENCY_SCALE_MODE_m11            ((ui1) 2)
```

FUNCTION: CMP_allocate_processing_struct_m11()

```
// Prototype
CMP_PROCESSING_STRUCT_m11
*CMP_allocate_processing_struct_m11(CMP_PROCESSING_STRUCT_m11 *cps, ui4 mode, si8
data_samples, si8 compressed_data_bytes, si8 difference_bytes, ui4 block_samples,
CMP_PROCESSING_DIRECTIVES_m11 *directives, CMP_COMPRESSION_PARAMETERS_m11 *parameters);
```

Allocates a CMP_PROCESSING_STRUCT (CPS). Within the CPS, 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. If the cps parameter is not NULL, it will not be allocated. This is most common when an array of CPS's has been pre-allocated, and this function is being used to perform initialization, and allocation it's contents.

If directives or parameters are NULL, they are set to their defaults, if not, their contents are copied into the CPS.

```
data_samples: the size of the input or output data arrays, may be large enough to hold
many blocks
block_samples: number of samples in a single block (typically maximum number)
global_flag: assign the allocated CPS to the global CPS pointer.
```

example:

```
cps = CMP_allocate_processing_struct_m11(NULL, CMP_DECOMPRESSION_MODE_m11, max_samps,
CMP_MAX_COMPRESSED_BYTES_m11(max_samps, 1), 0,
CMP_MAX_DIFFERENCE_BYTES_m11(max_samps), max_samps, NULL, NULL, FALSE_m11);
```

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

FUNCTION: CMP_check_CPS_allocation_m11()

```
// Prototype
TERN_m11    CMP_check_CPS_allocation_m11(CMP_PROCESSING_STRUCT_m11 *cps);
```

Checks that the appropriate buffers are allocated in a CPS for the type of operation being performed. The operation is determined by the values of the members of the CPS's compression and directives structures. It returns TRUE_m11 if the appropriate buffers are allocated and FALSE_m11 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:

```
cps = CMP_allocate_processing_struct_m11(NULL, CMP_DECOMPRESSION_MODE_m11, max_samps,
CMP_MAX_COMPRESSED_BYTES_m11(max_samps, 1), 0,
CMP_MAX_DIFFERENCE_BYTES_m11(max_samps), max_samps, NULL, NULL, FALSE_m11);

cps->compression.mode = RED_FIXED_COMPRESSION_RATIO;

force_behavior(RETURN_ON_FAIL);
CMP_check_CPS_allocation_m11(cps);
force_behavior(RESTORE_BEHAVIOR);
```

FUNCTION: CMP_decode_m11()

```
// Prototype
void  CMP_decode_m11(CMP_PROCESSING_STRUCT_m11 *cps);
```

Decompress data passed in the CPS from block_header pointer to the CPS 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 retrending are performed as necessary. The block discontinuity status is returned in the discontinuity directive.

FUNCTION: CMP_decrypt_m11()

```
// Prototype
si4   CMP_decrypt_m11(CMP_BLOCK_FIXED_HEADER_m11 *block_header);
```

Decrypts data in the block pointed to by the block_header pointer in place.

FUNCTION: CMP_detrend_m11()

```
// Prototype
void  CMP_detrend_m11(si4 *input_buffer, si4 *output_buffer, si8 len,
CMP_PROCESSING_STRUCT_m11 *cps);
```

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

FUNCTION: CMP_free_processing_struct_m11()

```
// Prototype
void  CMP_free_processing_struct_m11(CMP_PROCESSING_STRUCT_m11 *cps));
```

Frees any arrays allocated within the CPS, and the CPS itself.

FUNCTION: CMP_get_variable_region_m11()

```
// Prototype
void  CMP_get_variable_region_m11(CMP_PROCESSING_STRUCT_m11 *cps);
```

Used by CMP_decode() to calculate the size of the variable region of a CMP block header, based on the header flags.

FUNCTION: CMP_MBE_decode_m11()

```
// Prototype
void  CMP_MBE_decode_m11(CMP_PROCESSING_STRUCT_m11 *cps);
```

Decompress data from block_header pointer to decompressed_ptr in CPS if compressed with Minimal Bit Encoding (MBE). This is called by CMP_decode_m11() for MBE encoded blocks.

FUNCTION: CMP_PRED_decode_m11()

```
// Prototype
void  CMP_PRED_decode_m11(CMP_PROCESSING_STRUCT_m11 *cps);
```

Decompress data from block_header pointer to decompressed_ptr in CPS if compressed with the Predictive RED (PRED) algorithm. This is called by CMP_decode_m11() for PRED encoded blocks.

FUNCTION: CMP_quantile_value_m11()

```
// Prototype
sf8   CMP_quantile_value_m11(sf8 *x, si8 len, sf8 quantile, TERN_m11 preserve_input, sf8
*buff);
```

Returns the requested quantile value i.e 0.0 == minimum, 1.0 == maximum, 0.5 == median, etc. If the input array can be destroyed, set preserve_input to TRUE_m11, as this is most efficient, if it cannot, set preserve_input to FALSE_m11. Under these circumstances a buffer is required, if one is passed it will be used, if not it will be allocated and freed with each call.

FUNCTION: CMP_RED_decode_m11()

```
// Prototype
void  CMP_RED_decode_m11(CMP_PROCESSING_STRUCT_m11 *cps);
```

Decompress data from block_header pointer to decompressed_ptr in CPS if compressed with the Range Encoded Derivatives (RED) algorithm. This is called by CMP_decode_m11() for RED encoded blocks.

FUNCTION: CMP_retrend_m11()

```
// Prototype
void  CMP_retrend_m11(si4 *input_buffer, si4 *output_buffer, si8 len, sf8 m, sf8 b);
```

Retrend data from input_buffer to output_buffer. If input_buffer == output_buffer retrending data will be done in place. “m” & “b” are slope & intercept of trendline.

FUNCTION: CMP_round_m11()

```
// Prototype
inline si4      CMP_round_m11(sf8 val);
```

Return rounded si4 from sf8, taking into account the MED reserved values: NAN_m11, POSITIVE_INFINITY_m11, & NEGATIVE_INFINITY_m11.

FUNCTION: CMP_show_block_header_m11()

```
// Prototype
void  CMP_show_block_header_m11(CMP_BLOCK_FIXED_HEADER_m11 *bh);
```

Displays contents of a CMP block header. Useful in debugging.

FUNCTION: CMP_show_block_model_m11()

```
// Prototype
void  CMP_show_block_model_m11(CMP_BLOCK_FIXED_HEADER_m11 *block_header);
```

Displays the compression model vales and parameters in CMP block header. Useful in debugging.

FUNCTION: CMP_unscale_amplitude_m11()

```
// Prototype
void  CMP_unscale_amplitude_m11(si4 *input_buffer, si4 *output_buffer, si8 len, sf8
scale_factor);
```

Unscale amplitude from input_buffer to output_buffer. If input_buffer == output_buffer scaling will be done in place.

FUNCTION: CMP_unscale_frequency_m11()

```
// Prototype
void  CMP_unscale_frequency_m11(si4 *input_buffer, si4 *output_buffer, si8 len, sf8
scale_factor);
```

Unscale frequency from input_buffer to output_buffer. If input_buffer == output_buffer scaling will be done in place.

FUNCTION: CMP_update_CPS_pointers_m11()

```
// Prototype
inline CMP_BLOCK_FIXED_HEADER_m11
*CMP_update_CPS_pointers_m11(CMP_PROCESSING_STRUCT_m11 *cps, ui1 flags);

// Update CPS Pointer Flags
#define CMP_UPDATE_ORIGINAL_PTR_m11       ((ui1) 1)
#define CMP_UPDATE_BLOCK_HEADER_PTR_m11   ((ui1) 2)
#define CMP_UPDATE_DECOMPRESSED_PTR_m11   ((ui1) 4)
```

A function to update pointers in the CPS during rounds of compression or decompression. The examples below will make its utility more clear.

Example 1: Increment original_ptr & block_header in CPS during sequential compression

```
for (i = start_block; i < end_block; ++i) {
      CMP_encode_m11(cps);
      cps->block_header = CMP_update_CPS_pointers_m11(cps, \
            CMP_UPDATE_ORIGINAL_PTR_m11 | CMP_UPDATE_BLOCK_HEADER_PTR_m11);
}
```

Example 2: Increment block_header & decompressed_ptr pointers in CPS during sequential decompression

```
for (i = start_block; i < end_block; ++i) {
      CMP_decode_m11(cps);
      cps->block_header = CMP_update_CPS_pointers_m11(cps, \
            CMP_UPDATE_BLOCK_HEADER_PTR_m11 | CMP_UPDATE_DECOMPRESSED_PTR_m11);
}
```

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

FUNCTION: CRC_calculate_m11()

```
// Prototype
inline ui4  CRC_calculate_m11(const ui1 *block_ptr, si8 block_bytes);

// Constant
#define CRC_POLYNOMIAL_m11      ((ui4) 0xEDB88320)
#define CRC_START_VALUE_m11     ((ui4) 0x00000000)
```

Returns the 32-bit CRC for a block of length block_bytes, pointed to by block_ptr.

Note library CRC routines are customized to the polynomial defined above; it cannot be changed arbitrarily.

```
crc = CRC_calculate_m11(block_ptr, block_bytes);
```

is equivalent to:

```
crc = CRC_update_m11(CRC_calculate_m11, block_bytes, CRC_START_VALUE);
```

FUNCTION: CRC_combine_m11()

```
// Prototype
ui4   CRC_combine_m11(ui4 block_1_crc, ui4 block_2_crc, si8 block_2_bytes);
```

Returns the 32-bit CRC for two blocks with known CRCs as if they were joined and calculated as one. This allows efficient incremental addition of blocks with known CRCs as occurs in CMP blocks and MED records.

Example: update universal header body CRC from with record CRC (from calculate_record_data_CRCs_m11()):

```
// calculate record CRC
record_header->record_CRC = CRC_calculate_m11((ui1 *) record_header + \
      RECORD_HEADER_CRC_START_OFFSET_m11, record_header->total_bytes - \
      RECORD_HEADER_CRC_START_OFFSET_m11);

// calculate record header CRC
header_CRC = CRC_calculate_m11((ui1 *) record_header, \
      RECORD_HEADER_CRC_START_OFFSET_m11);

// combine record header & record body CRCs
full_record_CRC = CRC_combine_m11(header_CRC, record_header->record_CRC, \
      record_header->total_bytes - RECORD_HEADER_CRC_START_OFFSET_m11);

// combine universal_header->body_CRC & full_record_CRC
fps->universal_header->body_CRC = CRC_combine_m11(fps->universal_header->body_CRC, \
      full_record_CRC, record_header->total_bytes);
```

FUNCTION: CRC_initialize_table_m11()

```
// Prototype
ui4   **CRC_initialize_table_m11(TERN_m11 global_flag);
```

Allocates and initializes the CRC table generated from the CRC polynomial into heap space. If global_flag is set, the MED_globals pointer CRC_table is also set to this value. This function is called by initialize_medlib_m11().

FUNCTION: CRC_update_m11()

```
// Prototype
inline ui4  CRC_update_m11(const ui1 *block_ptr, si8 block_bytes, ui4 current_crc);
```

Returns the CRC of a block based on the current CRC of the bytes preceding that block.

FUNCTION: CRC_validate_m11()

```
// Prototype
inline TERN_m11   CRC_validate_m11(const ui1 *block_ptr, si8 block_bytes, ui4
crc_to_validate);
```

Returns TRUE_m11 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, FALSE_m11 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) medlib functions are described in this section.

FUNCTION: UTF8_initialize_tables_m11()

```
// Prototype
TERN_m11    UTF8_initialize_tables_m11(void);
```

Allocates and initializes the UTF8 tables into heap space. This function is called by initialize_medlib().

FUNCTION: UTF8_fprintf()

```
// Prototype
si4   UTF8_fprintf_m11(FILE *stream, si1 *fmt, …);
```

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

FUNCTION: UTF8_nextchar_m11()

```
// Prototype
ui4   UTF8_nextchar_m11(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_m11()

```
// Prototype si4  UTF8_printf_m11(si1 *fmt, ...);
```

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

FUNCTION: UTF8_strlen_m11()

```
// Prototype
si4   UTF8_strlen_m11(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_key_expansion(si4 Nk, si4 Nr, ui1 *RoundKey, si1 *Key);
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]);
```

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) medlib functions are described in this section.

FUNCTION: AES_initialize_tables_m11()

```
// Prototype
TERN_m11    AES_initialize_tables_m11(void);
```

Allocates and initializes the AES tables into heap space. This function is called by initialize_medlib().

FUNCTION: AES_decrypt_m11()

```
// Prototype
void  AES_decrypt_m11(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 medlib 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_m11()

```
// Prototype
void  AES_key_expansion_m11(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 medlib 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
ui1         *SHA_hash_m11(const ui1 *data, si8 len, ui1 *hash);
void        SHA_finalize_m11(SHA_CTX_m11 *ctx, ui1 *hash);
void        SHA_initialize_m11(SHA_CTX_m11 *ctx);
TERN_m11    SHA_initialize_tables_m11(void);
void        SHA_transform_m11(SHA_CTX_m11 *ctx, const ui1 *data);
void        SHA_update_m11(SHA_CTX_m11 *ctx, const ui1 *data, si8 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) medlib functions are described in this section.

FUNCTION: SHA_initialize_tables()

```
// Prototype
ui4   *SHA_initialize_h0_tables(void);
```

Allocates and initializes SHA tables into heap space. This function is called by initialize_medlib().

FUNCTION: SHA_hash_m11()

```
// Prototype
ui1   *SHA_hash_m11(const ui1 *data, si8 len, ui1 *hash);

// Constant
#define SHA_OUTPUT_SIZE 256
```

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

MED Records API

User defined records are defined and coded in “medrec_m11.c” and “medrec_m11.h”. The functions required for adding a new record type are described here. Record types themselves are described in the file “MED 1.0 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 in the hexadecimal representation.**

Each new record type should have two or three associated functions:
1) a “show” function)
2) an “alignment” function

“Show” functions display the contents of the records and have the following form:

Name: show_rec_*xxxx*_type_m11()
(where “xxxx” is the record type name)

```
// Prototype
void  show_medrec_xxxx_type_m11(RECORD_HEADER_m11 *record_header);
```
(where a RECORD_HEADER_m11 is a structure defined in “medlib_m11.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_rec_Note_type_m11(RECORD_HEADER_m11 *record_header)
{
      si1   *Note;


      // Version 1.0
      if (record_header->version_major == 1 && record_header->version_minor == 0) {
            Note = (si1 *) record_header + REC_Note_v10_TEXT_OFFSET_m11;
            UTF8_printf("Note text: %s\n", Note);
      }
      // Unrecognized record version
      else {
            warning_message_m11(“Unrecognized Note version (%hhd.%hhd)\n”, \
            record_header->version_major, record_header->version_minor);
      }

      return;
}
```

All show function constants are defined in “medrec_m11.h”. The function show_record_m11() defined in medrec_m11.c must be modified in the switch statement, copied below, to add new record types.

```
switch (type_code) {
```

```
	case REC_Sgmt_TYPE_CODE_m11:
		show_rec_Sgmt_type_m11(record_header);
		break;
	case REC_Stat_TYPE_CODE_m11:
		show_rec_Stat_type_m11(record_header);
		break;
	case REC_Note_TYPE_CODE_m11:
		show_rec_Note_type_m11(record_header);
		break;
	case REC_Seiz_TYPE_CODE_m11:
		show_rec_Seiz_type_m11(record_header);
		break;
	case REC_SyLg_TYPE_CODE_m11:
		show_rec_SyLg_type_m11(record_header);
		break;
	default:
		warning_message_m11("%s(): 0x%x is an unrecognized record type code", \
			__FUNCTION__, type_code);
		break;
}
```

"Alignment" functions have the following form:

Name: check_rec_xxxx_type_alignment_m11()
(where "xxxx" is the record type name)

```
// Prototype
TERN_m11	check_rec_xxxx_type_alignment_m11(ui1 *bytes);
```
(where "bytes" is an optional buffer to check alignment with)

New record "alignment" functions check the alignment of any structures represented in the record body. Those structures are defined in "medrec_m11.h". The function check_record_structure_alignments_m11() defined in medrec_m11.c must be modified in the serial if statements, copied below, to add a new record type.

```
if ((check_rec_Sgmt_type_alignment_m11(bytes)) == FALSE_m11)
	return_value = FALSE_m11;
if ((check_rec_Stat_type_alignment_m11(bytes)) == FALSE_m11)
	return_value = FALSE_m11;
if ((check_rec_Note_type_alignment_m11(bytes)) == FALSE_m11)
	return_value = FALSE_m11;
if ((check_rec_Seiz_type_alignment_m11(bytes)) == FALSE_m11)
	return_value = FALSE_m11;
if ((check_rec_SyLg_type_alignment_m11(bytes)) == FALSE_m11)
	return_value = FALSE_m11;
```