

# LAR Library 1.14

Main Page

Related Pages

Modules

Data Structures

Files

## Todo List

---

### **Module larlib\_format\_fn**

---

Review the list of format functions and add new ones.

### **Module larlib\_fs**

---

Directory iteration

# LAR Library 1.14

Main Page	Related Pages	Modules	Data Structures
Files			

## Modules

Here is a list of all modules:

[detail level [1](#) [2](#)]

<a href="#">Basic Definitions</a>	
<a href="#">Bit operations</a>	
<a href="#">Conversion routines</a>	
<a href="#">Date and time</a>	
<a href="#">Double-ended queue</a>	
<a href="#">▼ Buffer formatting</a>	
<a href="#">Field Formatting Functions</a>	Set of pre-defined formatting functions
<a href="#">File System</a>	
<a href="#">INI file parsing</a>	
<a href="#">Logging</a>	
<a href="#">Memory Management</a>	
<a href="#">State-machine</a>	
<a href="#">Table Management</a>	
<a href="#">Multi-threading</a>	

# LAR Library 1.14

Main Page	Related Pages	Modules	Data Structures	
Files	Macros   Typedefs   Enumerations   Functions			
<h2>Basic Definitions</h2>				

## Macros

---

#define **BASE\_MAX**(a, b)

Return the largest of two elements (as defined by the comparison <).

#define **BASE\_MIN**(a, b)

Return the smallest of two elements (as defined by the comparison <).

#define **BASE\_LENGTH**(v)

Return the count of a fixed-size array. [More...](#)

#define **LARLIB\_VERSION** (0x0113)

Version of this LarLib as a 2-byte BCD number. [More...](#)

---

## Typedefs

typedef char **int8\_t**

Signed 8-bit integer. [More...](#)

typedef short **int16\_t**

Signed 16-bit integer. [More...](#)

typedef int **int32\_t**

Signed 32-bit integer. [More...](#)

typedef long long **int64\_t**

Signed 64-bit integer. [More...](#)

typedef unsigned char **uint8\_t**

Unsigned 8-bit integer. [More...](#)

typedef unsigned short **uint16\_t**

Unsigned 16-bit integer. [More...](#)

typedef unsigned int **uint32\_t**

Unsigned 32-bit integer. [More...](#)

typedef unsigned long long **uint64\_t**

Unsigned 64-bit integer. [More...](#)

## Enumerations

```
enum baseErrors_t {
    BASE_ERR_OK = 0, BASE_ERR_INVALID_PARAMETER
= -1, BASE_ERR_DATA_NOT_FOUND = -2,
    BASE_ERR_INVALID_HANDLE = -3,
    BASE_ERR_RESOURCE_PB = -4,
    BASE_ERR_OVERFLOW = -5, BASE_ERR_ACCESS = -6,
    BASE_ERR_CANCEL = -7,
    BASE_ERR_TIMEOUT = -8
}
```

Common error values. [More...](#)

---

## Functions

---

const char \* **larlibGetVersion** (void)

Return a string with the version and revision of LarLib library. More...

---

## Detailed Description

---

## Rationale

Each platform has its own definition of types and error codes. We need a platform-independent basis to use for our own libraries. As an extra we try to reuse the types from C99's `<stdint.h>`.

# **Introduction**

This module includes basic definitions, types and functions that should be used as the environment for other modules.

# Macro Definition Documentation

---

## #define BASE\_LENGTH( v )

---

Return the count of a fixed-size array.

This will **not** work with pointers and other arrays, for example:

```
1 myStruct_t v[10];
2 myStruct_t *pv = v;
3 int nv = BASE_LENGTH(v); // works
4 int npv = BASE_LENGTH(pv); // does NOT
   work
```

## #define LARLIB\_VERSION (0x0113)

---

Version of this LarLib as a 2-byte BCD number.

For example, 0x0123 indicates version 1.23.

This allows both the application code and the (human) reader to know which version of LarLib this header file is associated with.

# Typedef Documentation

---

**typedef short int16\_t**

---

Signed 16-bit integer.

**typedef int int32\_t**

---

Signed 32-bit integer.

**typedef long long int64\_t**

---

Signed 64-bit integer.

**typedef char int8\_t**

---

Signed 8-bit integer.

**typedef unsigned short uint16\_t**

---

Unsigned 16-bit integer.

**typedef unsigned int uint32\_t**

---

Unsigned 32-bit integer.

**typedef unsigned long long uint64\_t**

---

Unsigned 64-bit integer.

**typedef unsigned char uint8\_t**

---

Unsigned 8-bit integer.

# Enumeration Type Documentation

---

## **enum baseErrors\_t**

---

Common error values.

Enumerator	Description
BASE_ERR_OK	No error.
BASE_ERR_INVALID_PARAMETER	Function was given an invalid parameter.
BASE_ERR_DATA_NOT_FOUND	Requested data was not found.
BASE_ERR_INVALID_HANDLE	Given handle is invalid.
BASE_ERR_RESOURCE_PB	Resource allocation problem (usually memory)
BASE_ERR_OVERFLOW	Either parameter is too large, or result buffer too small.
BASE_ERR_ACCESS	Access/permission failure.
BASE_ERR_CANCEL	Canceled by external process (user?)
BASE_ERR_TIMEOUT	Time-limited function has not completed in the requested time.

# Function Documentation

---

## **const char\* larlibGetVersion ( void )**

---

Return a string with the version and revision of LarLib library.

The returned string always follows the format "xx.yy \$Revision: nnn \$", where:

- xx.yy are the major and minor versions (from **LARLIB\_VERSION**);
- nnn is the SVN revision of this build (the length in digits is variable, do not always consider to be 3!).

For example "01.23 \$Revision: 111 \$".

### **Note**

If the version value returned by this function does not match the value of **LARLIB\_VERSION**, there is a mismatch between header and library.

### **Returns**

The library version and SVN revision of LarLib binary library file.

# LAR Library 1.14

[Main Page](#)

[Related Pages](#)

[Modules](#)

[Data Structures](#)

[Files](#)

[Typedefs](#) | [Enumerations](#) | [Functions](#)

## Bit operations

## Typedefs

---

**typedef uint8\_t bitsBuffer\_t**

A bit-buffer is a sequence of 8-bits unsigned integers. More...

---

## Enumerations

---

```
enum bitsBltOp_t {
    BITS_BLT_COPY, BITS_BLT_NOT, BITS_BLT_AND,
    BITS_BLT_OR,
    BITS_BLT_XOR, BITS_BLT_NOT_AND
}
```

Possible bit operations for **bitsBlt()**. More...

---

## Functions

int **bitsSet** (**bitsBuffer\_t** \*b, int index, int value)  
Set or clear a bit in b. [More...](#)

int **bitsSetRange** (**bitsBuffer\_t** \*b, int offset, int count, int value)  
Set a range of bits on a buffer. [More...](#)

int **bitsGet** (const **bitsBuffer\_t** \*b, int index)  
Return the value of a bit in b. [More...](#)

**int64\_t** **bitsExtractInt** (const **bitsBuffer\_t** \*b, int offset, int count)  
Extract a sub-sequence of a bit buffer as an integer. [More...](#)

int **bitsAppend** (const **bitsBuffer\_t** \*src, int srcOffset, int srcCount, **bitsBuffer\_t** \*dst, int dstCount)  
Append one bit buffer to another. [More...](#)

int **bitsAppendInt** (**uint64\_t** bits, int nbits, **bitsBuffer\_t** \*dst, int dstCount)  
Append the bits extracted from an integer to a bit buffer.  
[More...](#)

int **bitsRotateLeft** (**bitsBuffer\_t** \*b, int index, int nbits, int nrotate)  
Rotate b left by nrotate bits. [More...](#)

int **bitsRotateRight** (**bitsBuffer\_t** \*b, int index, int nbits, int nrotate)  
Rotate b right by nrotate bits. [More...](#)

int **bitsShiftLeft** (**bitsBuffer\_t** \*b, int index, int nbits, int nshift)  
Shift b left by nshift bits. [More...](#)

int **bitsShiftRight** (**bitsBuffer\_t** \*b, int index, int nbits, int nshift)  
Shift b right by nshift bits. [More...](#)

int **bitsCopy** (const **bitsBuffer\_t** \*src, int srcOffset, int srcCount,  
**bitsBuffer\_t** \*dst, int dstOffset)  
Copy a sequence of bits from one buffer to another. [More...](#)

int **bitsBlt** (const **bitsBuffer\_t** \*src, int srcOffset, int srcCount,  
enum **bitsBltOp\_t** op, **bitsBuffer\_t** \*dst, int dstOffset)  
Combine two bit buffers using a bit operation. [More...](#)

---

## Detailed Description

---

### Rationale

Many of the protocols, standards and devices we have to deal require manipulation bit by bit of structures. Even though certain algorithms are quite simple (set bit, get bit) others can be complex (rotate, shift).

# Introduction

This module manipulates sequences of bits. All functions operate on "naked" sequences of bytes, this was decided in order to keep this module low level. Users may wrap the functions to provide a more secure API, with bounds checking, for example.

Note that in all cases below, sizes and offsets/indices are given in bits, and all operations consider the left most bit of the left most byte as the bit index zero. As is default in C, all indices are zero-based.

A bit buffer of 16 bits, with the bit index 13 set, equals the binary value "00000000000000100" or the buffer { 0x00, 0x04 }.

Certain operations may be faster if the number of bits to operate is multiple of 8 (whole bytes), but there is no guarantee on this regard.

## Typedef Documentation

---

### **typedef uint8\_t bitsBuffer\_t**

---

A bit-buffer is a sequence of 8-bits unsigned integers.

Note that the actual size *in bits* of the buffer does not need to be a multiple of 8, but memory allocation is done in bytes.

# Enumeration Type Documentation

---

## **enum bitsBltOp\_t**

---

Possible bit operations for **bitsBlt()**.

Enumerator	Description
BITS_BLT_COPY	Copy source bits into destination.
BITS_BLT_NOT	Copy negated source bits into destination.
BITS_BLT_AND	Logical AND of source and destination.
BITS_BLT_OR	Logical OR of source and destination.
BITS_BLT_XOR	Logical XOR of source and destination.
BITS_BLT_NOT_AND	Logical AND of negated source and destination.

# Function Documentation

---

```
int bitsAppend ( const bitsBuffer_t * src,  
                  int               srcOffset,  
                  int               srcCount,  
                  bitsBuffer_t *   dst,  
                  int               dstCount  
                )
```

---

Append one bit buffer to another.

Append the srcCount bits of src starting at offset srcOffset to a bit buffer dst that already have dstCount bits.

No bounds checking is done on dst or src.

## Precondition

dst and src must not overlap.

## Parameters

**src** Source bit buffer.

**srcOffset** Index of first bit to read from src.

**srcCount** Number of bits to append.

**dst** Destination bit buffer.

**dstCount** Current size of dst in bits.

## Returns

The new size of dst on success ( $dstCount + srcCount$ ).

BASE\_ERR\_INVALID\_PARAMETER.

```
int bitsAppendInt ( uint64_t      bits,  
                     int          nbits,
```

```
    bitsBuffer_t * dst,  
    int           dstCount  
)
```

Append the bits extracted from an integer to a bit buffer.

Append, from left to right, the right most `nbits` bits of `bits` into the bit buffer `dst` that already have `dstCount` bits.

For example, given a bit buffer with value "1010", using `bits` = 0xC and `nbits` = 4 would result in the bit buffer "10101100", inserting the sequence "1100". Notice that the bits to insert should be the right most ones on the input integer, but they are inserted from left to right (most significant first).

No bounds checking is done on `dst`.

## Parameters

`bits` Value of the bits to insert.  
`nbits` Number of bits to insert.  
`dst` Destination bit buffer.  
`dstCount` Current size of `dst` in bits.

## Returns

The new size of `dst` on success (`dstCount + nbits`).  
`BASE_ERR_INVALID_PARAMETER`.

```
int bitsBlt ( const bitsBuffer_t * src,  
              int           srcOffset,  
              int           srcCount,  
              enum bitsBltOp_t op,  
              bitsBuffer_t * dst,  
              int           dstOffset  
)
```

Combine two bit buffers using a bit operation.

This is a very general function that allow to combine two bit buffers using one of many bit operations. `dst` is replaced with the result of the operation defined by `op` on both `src` and `dst` (note that certain operations ignore the value of `dst`, for example [BITS\\_BLT\\_COPY](#)).

`src` and `dst` may overlap.

## Parameters

**src** Source bit buffer.  
**srcOffset** First index to read from `src`.  
**srcCount** Number of bits to read.  
**op** Which bit operation to perform.  
**dst** Destination bit buffer.  
**dstOffset** First index to modify of `dst`.

## Returns

`BASE_ERR_OK`.  
`BASE_ERR_INVALID_PARAMETER`.

```
int bitsCopy ( const bitsBuffer_t * src,
                int                  srcOffset,
                int                  srcCount,
                bitsBuffer_t *        dst,
                int                  dstOffset
)
```

Copy a sequence of bits from one buffer to another.

Copy `srcCount` bits starting at bit index `srcOffset` from `src` into `dst` at offset `dstOffset`. This is equivalent to calling [bitsBlt\(\)](#) with the [BITS\\_BLT\\_COPY](#) operation.

No bounds checking is done on `dst` or `src`.

## Parameters

**src** Source bits buffer.  
**srcOffset** Index of first bit to copy, in bits.  
**srcCount** Number of bits to copy.  
**dst** Destination bits buffer.  
**dstOffset** Index of first bit to overwrite in dst.

## Returns

BASE\_ERR\_OK on success.  
BASE\_ERR\_INVALID\_PARAMETER.

```
int64_t bitsExtractInt ( const bitsBuffer_t * b,
                         int                  offset,
                         int                  count
                       )
```

---

Extract a sub-sequence of a bit buffer as an integer.

Interpret **count** bits of **b** starting at **offset** as an integer (most-significant bit first). Note that this function call can only return unsigned integers smaller than  $2^{63}-1$ , since negative numbers are used for error reporting.

For example, the integer at offset 2, count 3 in "01101000" is "101" whose value is 5.

No bounds checking is done on **dst** or **src**.

## Parameters

**b** Bits buffer to read.  
**offset** Index of first bit to read.  
**count** Number of bits to read.

## Returns

The non-negative value of the bits read.  
BASE\_ERR\_INVALID\_PARAMETER.

```
int bitsGet( const bitsBuffer_t * b,  
             int                  index  
           )
```

---

Return the value of a bit in **b**.

Note that this function does *no* bounds checking, the caller must guarantee that **index** is a valid bit index inside **b**.

### Parameters

**b**      **bitsBuffer\_t** to modify.  
**index** Index of bit to return.

### Returns

The value of the bits at **index** as 0 (clear) or 1 (set)

**BASE\_ERR\_INVALID\_PARAMETER** if **b** == **NULL** or **index** is invalid.

```
int bitsRotateLeft( bitsBuffer_t * b,  
                     int          index,  
                     int          nbits,  
                     int          nrotate  
                   )
```

---

Rotate **b** left by **nrotate** bits.

Note that both the buffer size and the rotation size are *in bits*, it is valid to rotate 3 bits to the left a buffer of 11 bits.

When rotating, the bits that fall off on the left, are re-inserted on the right. For example, rotating "11110000" 1 bit to the left results in "11100001".

### Parameters

**b**      **bitsBuffer\_t** to rotate.

**index** First index on **b** to be affected.  
**nbits** Number of bits of **b** to be affected.  
**nrotate** Number of bits to rotate.

### Returns

BASE\_ERR\_OK on success.  
BASE\_ERR\_INVALID\_PARAMETER if **b** == NULL or **nbits** or **nrotate** are invalid.

```
int bitsRotateRight( bitsBuffer_t * b,  
                      int           index,  
                      int           nbits,  
                      int           nrotate  
)
```

---

Rotate **b** right by **nrotate** bits.

Note that both the buffer size and the rotation size are *in bits*, it is valid to rotate 3 bits to the right a buffer of 11 bits.

When rotating, the bits that fall off on the right, are re-inserted on the left. For example, rotating "00001111" 1 bit to the right results in "10000111".

### Parameters

**b** [bitsBuffer\\_t](#) to rotate.  
**index** First index on **b** to be affected.  
**nbits** Number of bits of **b** to be affected.  
**nrotate** Number of bits to rotate.

### Returns

BASE\_ERR\_OK on success.  
BASE\_ERR\_INVALID\_PARAMETER if **b** == NULL or **nbits** or **nrotate** are invalid.

```
int bitsSet( bitsBuffer_t * b,  
            int          index,  
            int          value  
        )
```

---

Set or clear a bit in **b**.

Note that this function does *no* bounds checking, the caller must guarantee that **index** is a valid bit index inside **b**.

### Parameters

**b** **bitsBuffer\_t** to modify.  
**index** Index of bit to set.  
**value** Zero to clear the bit, non-zero to set it.

### Returns

**BASE\_ERR\_OK** on success.  
**BASE\_ERR\_INVALID\_PARAMETER** if **b** == **NULL** or **index** is invalid.

```
int bitsSetRange( bitsBuffer_t * b,  
                  int          offset,  
                  int          count,  
                  int          value  
                )
```

---

Set a range of bits on a buffer.

Set **count** bits of **b** starting at **index offset** to **value**.

### Parameters

**b** Bit buffer to modify.  
**offset** Index of first bit to modify.  
**count** Number of bits to modify.  
**value** Value to set (zero to clear, non-zero to set).

## Returns

BASE\_ERR\_OK.  
BASE\_ERR\_INVALID\_PARAMETER.

---

```
int bitsShiftLeft ( bitsBuffer_t * b,
                     int             index,
                     int             nbits,
                     int             nshift
)
```

---

Shift b left by nshift bits.

Note that both the buffer size and the shift size are *in bits*, it is valid to shift 3 bits to the left a buffer of 11 bits.

When shifting, zeroes are inserted on the right as the bits move left.

## Parameters

**b** bitsBuffer\_t to shift.  
**index** First index on b to be affected.  
**nbits** Size of b *in bits*.  
**nshift** Number of *bits* to shift b.

## Returns

BASE\_ERR\_OK on success.  
BASE\_ERR\_INVALID\_PARAMETER if b == NULL or nbits or nshift are invalid.

---

```
int bitsShiftRight ( bitsBuffer_t * b,
                      int            index,
                      int            nbits,
                      int            nshift
)
```

---

Shift b right by `nshift` bits.

Note that both the buffer size and the shift size are *in bits*, it is valid to shift 3 bits to the right a buffer of 11 bits.

When shifting, zeroes are inserted on the left as the bits move right.

## Parameters

**b**     `bitsBuffer_t` to shift.

**index** First index on b to be affected.

**nbits** Size of b *in bits*.

**nshift** Number of *bits* to shift b.

## Returns

`BASE_ERR_OK` on success.

`BASE_ERR_INVALID_PARAMETER` if b == NULL or nbits or nshift are invalid.

# LAR Library 1.14

Main Page	Related Pages	Modules	Data Structures	
Files	Enumerations   Functions			
<h2>Conversion routines</h2>				

## Enumerations

---

enum **convErrors\_t** { **CONV\_ERR\_INVALID\_BASE** = -101 }

Error codes specific to this module. [More...](#)

---

## Functions

int **convTxtToInt** (const char \*txt, int cnt, int base, **uint64\_t** \*n)  
Convert Text to Integer. [More...](#)

int **convTxtToInt8** (const char \*txt, int cnt, int base, **uint8\_t** \*n)  
Convert Text to an **uint8\_t**. [More...](#)

int **convTxtToInt16** (const char \*txt, int cnt, int base, **uint16\_t** \*n)  
Convert Text to an **uint16\_t**. [More...](#)

int **convTxtToInt32** (const char \*txt, int cnt, int base, **uint32\_t** \*n)  
Convert Text to an **uint32\_t**. [More...](#)

int **convTxtToInt64** (const char \*txt, int cnt, int base, **uint64\_t** \*n)  
This function is a synonym to **convTxtToInt()**. [More...](#)

int **convIntToTxt** (**uint64\_t** n, char \*txt, int maxcnt, int base)  
Convert Integer to Text. [More...](#)

int **convIntToTxtPad** (**uint64\_t** n, char \*txt, int cnt, int base)  
Convert Integer to Text padding left with zeroes to fill cnt chars. [More...](#)

int **convBufToHex** (const **uint8\_t** \*buf, int nbuf, char \*hex, int maxcnt)  
Convert buffer to hexadecimal string. [More...](#)

int **convHexToBuf** (const char \*hex, int cnt, **uint8\_t** \*buf, int maxbuf)  
Convert hexadecimal string to buffer. [More...](#)

int **convIntToBcd** (**uint64\_t** n, **uint8\_t** \*bcd, int maxcnt, int padRight)  
Convert Integer to BCD. [More...](#)

int **convIntToBcdPad** (**uint64\_t** n, **uint8\_t** \*bcd, int nibbles)  
Convert Integer to BCD with left padding. [More...](#)

int **convBcdToInt** (const **uint8\_t** \*bcd, int cnt, **uint64\_t** \*n)  
Convert BCD to Integer. [More...](#)

int **convTxtToBcd** (const char \*txt, int maxtxt, **uint8\_t** \*bcd, int maxbcd)  
Convert a string of decimal digits (or ISO PAN) to BCD.  
[More...](#)

int **convBcdToTxt** (const **uint8\_t** \*bcd, int maxbcd, char \*txt, int maxtxt)  
Convert a sequence of digits in BCD format to a decimal string. [More...](#)

int **convIntToBuf** (**uint64\_t** n, **uint8\_t** \*buf, int maxbuf)  
Convert an integer to a sequence of bytes in *big-endian* ordering. [More...](#)

int **convIntToBufPad** (**uint64\_t** n, **uint8\_t** \*buf, int maxbuf)  
Convert an integer to a sequence of bytes in *big-endian* ordering and with padding. [More...](#)

**uint64\_t** **convBufToInt** (const **uint8\_t** \*buf, int nbuf)  
Convert a sequence of bytes in *big-endian* ordering to an integer. [More...](#)

int **convBase64ToBuf** (const char \*b64, int cnt, **uint8\_t** \*buf, int maxbuf)  
Decode a base-64 encoded string into a sequence of bytes. [More...](#)

```
int convBufToBase64 (const uint8_t *buf, int nbuf, char *b64,  
int maxb64)  
Encode a sequence of bytes into a Base-64 string. More...
```

---

## Detailed Description

---

### Rationale

There are many existing conversion libraries, but all of them have their own definition of which format is what, how to name them, and API look-and-feel. This module is a unification of all existing libraries with clearer names and error handling.

# Introduction

This package implements routines to convert between a few widely used formats for generic data (specially numerical data):

- *Text (Txt)*: a string representation of a number. It only makes sense with a base. For example, the string "1001" is either 4097 in hexadecimal, 9 in binary or 1001 in decimal. Only bases in the range [2, 36] are supported.
- *Integer (Int)*: internal numerical representation of integers for the current machine. The actual representation as bits in memory is not defined (or, more precisely, is platform dependent). In general only non-negative integers are accepted.
- *Binary-Coded Decimals (Bcd)*: each digit of a decimal number is represented by a 4-bit value, for example the decimal number 1234 is encoded as the number 0x1234. If the number of digits is odd, the number may either start with 0x0 (as in 0x012345) or use 0xF as a terminator (0x12345F).
- *Buffer (Buf)*: a binary representation of numbers in binary format. The decimal number 3405692655 is "CAFEBEEF" in base 16, and the buffer { 0xCA, 0xFE, 0xBE, 0xEF }.
- *Hexadecimal (Hex)*: a sub-class of Text (using base 16). It is differentiated since some functions only provide conversion to or from Hex.
- *Base-64 (Base64)*: the common Base-64 encoding of binary data.

All routines have names with the format `convXXXToYYY()` where `XXX` and `YYY` are either `Bcd`, `Txt`, `Int`, `Buf` or `Hex`, and `XXX` defines the *source* format and `YYY` the *destination* format.

## Note

For enhanced domain, all functions support Integer values represented by an `uint64_t`. This may cause a slight performance penalty in lieu of a much larger Integer range. This also means that most functions are limited to numbers up to  $2^{64}-1$ .

Not all possible conversion routines are implemented, some conversions are used much more often than others, and this module

tries to select the most used ones. Also it should provide a "path" from one format to another, so it may be possible to define a function that convert from any given format to another by calling pre-existing functions (i.e. the conversion graph should be fully connected).

Except otherwise noted, all functions are  $O(n)$  on the number of digits of the input number in whatever base and format it is to be processed.

# Enumeration Type Documentation

---

## **enum convErrors\_t**

---

Error codes specific to this module.

Enumerator	
CONV_ERR_INVALID_BASE	Invalid base for Text parameter.

# Function Documentation

---

```
int convBase64ToBuf ( const char * b64,
                      int          cnt,
                      uint8_t *    buf,
                      int          maxbuf
)
```

---

Decode a base-64 encoded string into a sequence of bytes.

Use the Base-64 index table

"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0::

Example:

```
1| unsigned char buf[24];
2| int n =
  convBase64ToBuf( "Sw5nZW5pY28gZG8gQnJhc2lsIQ==",
  sizeof(buf));
3| // n == 19 && buf == "Ingenico do Brasil!"
```

## Parameters

<b>b64</b>	Input base-64 text, a sequence of chars from the E table above
<b>cnt</b>	Number of chars to decode, or (-1) to use <code>strlen()</code>
<b>[out] buf</b>	Where to store the decoded bytes
<b>maxbuf</b>	Max number of bytes to write to <code>maxbuf</code>

## Returns

The number of bytes that `b64` decode to. If the returned value is larger than `maxbuf` then an overflow was detected, and `buf` was not written to.

`BASE_ERR_INVALID_PARAMETER` if `b64` or `buf` are `NULL`

## See also

[convBufToBase64](#)

<https://en.wikipedia.org/wiki/Base64>

---

```
int convBcdToInt ( const uint8_t * bcd,
                    int             cnt,
                    uint64_t *      n
)
```

Convert BCD to Integer.

### Parameters

**bcd** The BCD number to convert.

**cnt** The max number of bytes to convert. Conversion also stops when a non-decimal nibble is found (usually 0xF).

[out] **n** Where to store the converted number.

### Returns

The number of *decimal digits* of the number stored in **n**.

BASE\_ERR\_INVALID\_PARAMETER if **bcd** or **n** are invalid.

---

```
int convBcdToTxt ( const uint8_t * bcd,
                    int             maxbcd,
                    char *          txt,
                    int             maxtxt
)
```

Convert a sequence of digits in BCD format to a decimal string.

This function was designed to be primarily used with PAN number conversion for ISO messages. It assumes that, if the sequence has an odd number of digits, it will be terminated with 0xF.

If **bcd** includes the code 0xD, it will be changed to the char '=' on **txt**.

Any other digit code in `bcd` terminates execution.

### Note

The output string `txt` is *NOT* zero-terminated!

### Parameters

`bcd` Data in BCD format.

`maxbcd` Max number of bytes to decode from `bcd`.

[out] `txt` Output string, will *not* be zero-terminated!

`maxtxt` Max number of chars to write to `txt`

### Returns

The number of valid digits in `bcd`. If this is larger than `maxtxt` then no conversion was done.

`BASE_ERR_INVALID_PARAMETER` if `bcd` is `NULL` or `txt` is `NULL` and `maxtxt > 0`.

```
int convBufToBase64 ( const uint8_t * buf,  
                      int             nbuf,  
                      char *          b64,  
                      int             maxb64  
                    )
```

---

Encode a sequence of bytes into a Base-64 string.

Use the Base-64 index table

"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0+

### Parameters

`buf` Input sequence of bytes

`nbuf` Number of bytes to `buf` to encode

[out] `b64` Where to store the encoded Base-64 chars

`maxb64` Max number of chars to write to `b64`, including term

### Returns

The number of bytes necessary to fully encode buf. If the returned value is greater than maxb64 then an overflow was detected, and b64 was not written. Returns BASE\_ERR\_INVALID\_PARAMETER if buf or b64 are NULL.

## See also

[convBase64ToBuf](#)

<https://en.wikipedia.org/wiki/Base64>

---

```
int convBufToHex ( const uint8_t * buf,
                    int             nbuf,
                    char *          hex,
                    int             maxcnt
)
```

Convert buffer to hexadecimal string.

## Parameters

<b>buf</b>	Buffer to convert.
<b>nbuf</b>	Number of bytes to convert from buffer.
[out] <b>hex</b>	Where to store the converted data.
<b>maxcnt</b>	Max number of digits to write to hex ( <i>including</i> the terminating zero!)

## Note

maxcnt should be at least twice nbuf as each byte converts to two hexadecimal digits.

## Returns

The number of digits written to hex.

BASE\_ERR\_INVALID\_PARAMETER if buf or hex are invalid.

---

```
uint64_t convBufToInt ( const uint8_t * buf,
                        int             nbuf
)
```

Convert a sequence of bytes in *big-endian* ordering to an integer.

The opposite of [convIntToBuf\(\)](#).

### Parameters

**buf** Sequence of bytes to be converted  
**nbuf** How many bytes of buf to convert

### Returns

The converted value of buf[0 .. nbuf - 1]

Zero in any error

```
int convHexToBuf( const char * hex,
                  int          cnt,
                  uint8_t *    buf,
                  int          maxbuf
)
```

Convert hexadecimal string to buffer.

### Parameters

**hex** Hexadecimal string to convert.  
**cnt** Max number of digits to convert from hex. Use  
cnt == -1 to use strlen(hex) instead.  
[out] **buf** Output buffer with converted bytes.  
**maxbuf** Max number of bytes to write to buf.

### Note

maxbuf should be at least half min(strlen(hex), cnt), as each pair of hexadecimal digits convert to one byte.

If the number of hexadecimal digits is odd, the last byte will be padded with 0 to the right. For example "ABCDE" will be encoded as { 0xAB, 0xCD, 0xE0 }.

If a non-hexadecimal digit is found in `hex`, conversion will stop and the number of *complete* bytes written to `buf` is returned. If the number of converted digits is odd, the last (partial) byte is not counted in the return, but will be partially filled as if the number of input digits were odd. (E.g. "ABCDEX" returns 2 and writes {0xAB, 0xCD, 0xE0} to `buf`).

## Returns

The number of bytes written to `buf`.

`BASE_ERR_INVALID_PARAMETER` if `hex` or `buf` are invalid.

```
int convIntToBcd ( uint64_t n,  
                    uint8_t * bcd,  
                    int      maxcnt,  
                    int      padRight  
                )
```

---

Convert Integer to BCD.

## Parameters

<code>n</code>	The integer to convert.
[out] <code>bcd</code>	Where to store the BCD-coded <code>n</code> . Note that <code>bcd</code> will <i>not</i> be zero-terminated!
<code>maxcnt</code>	Max number of bytes to write to <code>bcd</code> .
<code>padRight</code>	If non-zero, the resulting BCD is padded left with a terminating 0xF. If zero odd-sized BCD numbers start with a 0x0 as padding. Note that this flag only applies to BCD numbers where the resulting number of digits is odd (thus requiring an "extra" nibble).

## Returns

The number of bytes necessary to fully convert `n`. If > `maxcnt`, then the conversion was not done, and `bcd` was not modified.

`BASE_ERR_INVALID_PARAMETER` if `bcd` is invalid.

```
int convIntToBcdPad ( uint64_t n,
                      uint8_t * bcd,
                      int      nibbles
)
```

---

Convert Integer to BCD with left padding.

This function is similar to [convIntToBcd\(\)](#), but instead of using the least necessary number of bytes to express the BCD encoded number, it fills all the nibbles requested in the `nibbles` parameter.

If the value of `nibbles` is odd, the last half of the last byte would be unused by the encoding function, and is filled with `0xF` instead.

If the value of `nibbles` is even, all the bytes are wholly used, and there is no need to fill with `0xF` the last byte.

In both cases, the `bcd` output buffer is prefixed with enough `0x00` bytes so that the encoding of `n` uses  $(\text{nibbles} + 1) / 2$  bytes.

## Parameters

<code>n</code>	The integer to convert.
[out] <code>bcd</code>	Where to store the BCD-coded <code>n</code> . Note that <code>bcd</code> will <i>not</i> be zero terminated. It is assumed that <code>bcd</code> has enough space for $(\text{nibble} + 1) / 2$ bytes.
<code>nibbles</code>	The number of <code>nibbles</code> to pad the converted BCD number to. See the notes above for more information.

## Returns

If `nibbles` is too small, return the number of `bytes` that would be required to fully encode `n`

On success return the number of `bytes` actually written to `bcd` (that should always be equal to  $(\text{nibbles} + 1) / 2$ )

BASE\_ERR\_INVALID\_PARAMETER if bcd or nibbles is invalid.

```
int convIntToBuf( uint64_t n,
                  uint8_t* buf,
                  int      maxbuf
)
```

Convert an integer to a sequence of bytes in *big-endian* ordering.

Example:

```
1| uint8_t buf[sizeof(uint64_t)];
2| int i = convIntToBuf(0x12345678, buf,
3| sizeof(buf));
4| // i == 4 && buf == { 0x12, 0x34, 0x56,
5| 0x78 }
```

## Parameters

**n** The integer to convert  
[out] **buf** Where to store the bytes of n  
**maxbuf** Max number of bytes that may be written to buf

## Returns

The number of bytes necessary to fully encode n. If > maxbuf then conversion was not performed and nothing is written to buf  
BASE\_ERR\_INVALID\_PARAMETER if buf is NULL or maxbuf < 0

```
int convIntToBufPad( uint64_t n,
                     uint8_t* buf,
                     int      maxbuf
)
```

Convert an integer to a sequence of bytes in *big-endian* ordering

and with padding.

This call is equivalent to [convIntToBuf\(\)](#) but the output buffer `buf` will be zero-padded to the left so that always `maxbuf` bytes are filled.

Example:

```
1| uint8_t buf[sizeof(uint64_t)];  
2| int i = convIntToBuf(0x12345678, buf,  
3| sizeof(buf));  
 // i == 8 && buf == { 0x00, 0x00, 0x00,  
 //   0x00, 0x12, 0x34, 0x56, 0x78 }
```

## Parameters

**n** The integer to convert  
[out] **buf** Where to store the bytes of `n`  
**maxbuf** Number of *bytes* that will be written to `buf`

## Returns

The number of bytes written to `buf`

In case of overflow, return how many bytes would be necessary to fully encode `n`

`BASE_ERR_INVALID_PARAMETER` if `buf` is `NULL` or `maxbuf < 0`

---

```
int convIntToTxt( uint64_t n,  
                  char * txt,  
                  int    maxcnt,  
                  int    base  
                )
```

Convert Integer to Text.

If `maxcnt` is not enough to store all converted digits plus the terminating zero, conversion will **not** be done, but the number of

digits required for conversion will be returned. Calling this function with `maxcnt == 0` and `txt == NULL` is a valid option to calculate the number of digits necessary to represent a number on a certain base.

## Parameters

- n** Number to be converted.
- [out] **txt** Where the converted digits will be stored.
- maxcnt** Maximum number of digits that may be written to `txt` (*including* the terminating zero!)
- base** Base to use for `txt`.

## Returns

The number of digits that would be necessary to encode `n` totally (including the terminating zero). If the return value is `>= maxcnt` the conversion did not happen and `txt` was never written to.

`CONV_ERR_INVALID_BASE` if `base` is out of range.

`BASE_ERR_INVALID_PARAMETER` if `txt` is invalid.

```
int convIntToTxtPad ( uint64_t n,  
                      char *   txt,  
                      int       cnt,  
                      int       base  
                    )
```

Convert Integer to Text padding left with zeroes to fill `cnt` chars.

For example, converting the number 13 with `cnt = 4` and `base = 10` results in the string `txt = "0013"`. Note that zero is used whichever base is used.

## Note

The output string `txt` is **not** zero-terminated!

## Parameters

**n** Number to be converted.  
[out] **txt** Where the converted digits will be stored.  
**cnt** Number of digits that should be written to txt.  
**base** Base to use for txt.

## Returns

The number of significant digits of **n** in **base** (that is, discounting the zeroes to the left inserted for padding). If the return is larger than **cnt** then **txt** is not large enough and no conversion was done.

**CONV\_ERR\_INVALID\_BASE** if **base** is out of range.

**BASE\_ERR\_INVALID\_PARAMETER** if **txt** is invalid.

```
int convTxtToBcd ( const char * txt,
                    int          maxtxt,
                    uint8_t *    bcd,
                    int          maxbcd
)
```

Convert a string of decimal digits (or ISO PAN) to BCD.

This function was designed to be primarily used with PAN number conversion for ISO messages. If the string is odd-sized, it is padded left with a terminating **0xF**.

If the sequence includes the char '=' , it is changed to **0xD** in the BCD output. Conversion ends at any char that is not a decimal digit or '='.

## Parameters

**txt** The source string. May be zero-terminated or not, see **maxcnt**.  
**maxtxt** Max numbers of chars to convert from **txt**. If (-1) use **strlen(txt)**.  
[out] **bcd** Target buffer for the BCD-coded value.  
**maxbcd** Max number of *bytes* that can be written to **bcd**.

## Returns

The number of *bytes* necessary to fully convert *txt*. If the return value is larger than *maxbcd*, then conversion was not done.

**BASE\_ERR\_INVALID\_PARAMETER** if *txt* is **NULL** or if *bcd* is **NULL** and *maxbcd* > 0

```
int convTxtToInt ( const char * txt,
                  int          cnt,
                  int          base,
                  uint64_t *   n
)
```

---

Convert Text to Integer.

Up to *cnt* digits of *txt* are converted to integer, using *base*. The result is stored in *n*.

## Parameters

**txt** String to convert.

**cnt** Up to *cnt* digits will be read from *txt*. Conversion also stops at any digit out of range. Use *cnt* == -1 to use `strlen(txt)` instead.

**base** Base of the number represented in *txt*.

[out] **n** Where the converted value should be stored.

## Returns

The number of digits converted from *txt*.

**CONV\_ERR\_INVALID\_BASE** if *base* is out of range.

**BASE\_ERR\_INVALID\_PARAMETER** if *txt*, *cnt*, or *n* are invalid.

**BASE\_ERR\_OVERFLOW** if the resulting integer does not fit on a *uint64\_t*

```
int convTxtToInt16 ( const char * txt,
```

```
    int      cnt,  
    int      base,  
    uint16_t * n  
)  
}
```

Convert Text to an `uint16_t`.

Behaves exactly as `convTxtToInt()`, but stores the result on a 16-bit unsigned integer instead.

## Parameters

**txt** String to convert.

**cnt** Up to `cnt` digits will be read from `txt`. Conversion also stops at any digit out of range. Use `cnt == -1` to use `strlen(txt)` instead.

**base** Base of the number represented in `txt`.

[out] **n** Where the converted value should be stored.

## Returns

The number of digits converted from `txt`.

`CONV_ERR_INVALID_BASE` if `base` is out of range.

`BASE_ERR_INVALID_PARAMETER` if `txt`, `cnt`, or `n` are invalid.

`BASE_ERR_OVERFLOW` if the resulting integer does not fit on a `uint16_t`

```
int convTxtToInt32 ( const char * txt,  
                     int      cnt,  
                     int      base,  
                     uint32_t * n  
)  
}
```

Convert Text to an `uint32_t`.

Behaves exactly as `convTxtToInt()`, but stores the result on a 32-bit unsigned integer instead.

## Parameters

- txt** String to convert.
- cnt** Up to **cnt** digits will be read from **txt**. Conversion also stops at any digit out of range. Use **cnt == -1** to use `strlen(txt)` instead.
- base** Base of the number represented in **txt**.
- [out] **n** Where the converted value should be stored.

## Returns

- The number of digits converted from **txt**.
- `CONV_ERR_INVALID_BASE` if **base** is out of range.
- `BASE_ERR_INVALID_PARAMETER` if **txt**, **cnt**, or **n** are invalid.
- `BASE_ERR_OVERFLOW` if the resulting integer does not fit on a `uint32_t`

```
int convTxtToInt64 ( const char * txt,
                      int          cnt,
                      int          base,
                      uint64_t *   n
)
```

---

This function is a synonym to [convTxtToInt\(\)](#).

It is added only for completeness, you may use it to make clear the intent of using a `uint64_t` as the conversion target.

## Parameters

- txt** String to convert.
- cnt** Up to **cnt** digits will be read from **txt**. Conversion also stops at any digit out of range. Use **cnt == -1** to use `strlen(txt)` instead.
- base** Base of the number represented in **txt**.
- [out] **n** Where the converted value should be stored.

## Returns

The number of digits converted from `txt`.  
`CONV_ERR_INVALID_BASE` if `base` is out of range.  
`BASE_ERR_INVALID_PARAMETER` if `txt`, `cnt`, or `n` are invalid.  
`BASE_ERR_OVERFLOW` if the resulting integer does not fit on a `uint64_t`

---

```
int convTxtToInt8( const char * txt,
                    int          cnt,
                    int          base,
                    uint8_t *    n
)
```

---

Convert Text to an `uint8_t`.

Behaves exactly as `convTxtToInt()`, but stores the result on an 8-bit unsigned integer instead.

## Parameters

**txt** String to convert.

**cnt** Up to `cnt` digits will be read from `txt`. Conversion also stops at any digit out of range. Use `cnt == -1` to use `strlen(txt)` instead.

**base** Base of the number represented in `txt`.

[out] **n** Where the converted value should be stored.

## Returns

The number of digits converted from `txt`.  
`CONV_ERR_INVALID_BASE` if `base` is out of range.  
`BASE_ERR_INVALID_PARAMETER` if `txt`, `cnt`, or `n` are invalid.  
`BASE_ERR_OVERFLOW` if the resulting integer does not fit on a `uint8_t`

---

Generated on Mon Mar 27 2017 15:42:52 for LAR Library by [doxygen](#) 1.8.9.1

# LAR Library 1.14

Main Page	Related Pages	Modules	Data Structures
Files			
<a href="#">Data Structures</a>   <a href="#">Enumerations</a>   <a href="#">Functions</a>			
<h2>Date and time</h2>			

## Data Structures

---

struct **dateDate\_t**

A gregorian date. [More...](#)

struct **dateTime\_t**

A gregorian date with time. [More...](#)

---

## Enumerations

---

```
enum dateMonths_t {  
    DATE_JAN = 1, DATE_FEB, DATE_MAR, DATE_APR,  
    DATE_MAY, DATE_JUN, DATE_JUL, DATE_AUG,  
    DATE_SEP, DATE_OCT, DATE_NOV, DATE_DEC  
}
```

Enumeration of months. [More...](#)

```
enum dateWeekdays_t {  
    DATE_SUN = 0, DATE_MON, DATE_TUE, DATE_WED,  
    DATE_THU, DATE_FRI, DATE_SAT  
}
```

Enumeration of days of a week. [More...](#)

---

## Functions

int **dateGet (dateDate\_t \*d)**

Return the current system date. [More...](#)

int **dateTimeGet (dateTime\_t \*dt)**

Return the current system date and time. [More...](#)

**uint64\_t dateTimeGetMs (void)**

Return an increasing count of milliseconds. [More...](#)

int **dateSet (const dateDate\_t \*d)**

Update the system date. [More...](#)

int **dateTimeSet (const dateTime\_t \*dt)**

Update the system date and time. [More...](#)

int **dateToJulianDay (const dateDate\_t \*d)**

Return the Julian Day Number of d. [More...](#)

int **dateFromJulianDay (int jdn, dateDate\_t \*d)**

Convert the Julian Day Number to the Gregorian calendar.

[More...](#)

int **dateWeekday (const dateDate\_t \*d)**

Return the weekday of a given date. [More...](#)

int **dateIsLeap (uint16\_t year)**

Return non-zero if given year number is a leap year.

[More...](#)

int **dateIsValid (const dateDate\_t \*d)**

Checks if a date is valid. [More...](#)

int **dateTimeIsValid (const dateTime\_t \*dt)**

Checks if both time and date are valid. [More...](#)

int **dateDiff** (const **dateDate\_t** \*a, const **dateDate\_t** \*b)  
Return the difference, in days, between two dates. [More...](#)

int **dateAddDays** (**dateDate\_t** \*d, int ndays)  
Add a number of days to a given date. [More...](#)

int **dateCompare** (const **dateDate\_t** \*a, const **dateDate\_t** \*b)  
Compare two dates. [More...](#)

int **dateTimeAddSeconds** (**dateTime\_t** \*dt, int nsecs)  
Add seconds to a given date and time. [More...](#)

int **dateTimeCompare** (const **dateTime\_t** \*a, const **dateTime\_t** \*b)  
Compare two date times. [More...](#)

int **dateTimeDiff** (const **dateTime\_t** \*a, const **dateTime\_t** \*b)  
Return the difference, in seconds, between two date times.  
[More...](#)

---

## Detailed Description

---

### Rationale

Date handling is widely known to be hard. Our day-to-day uses are basically reading and setting the system date, and doing simple date math to skip over weekends and computing closing and opening times.

# **Introduction**

This module provides basic functionality to work with dates and time. Time-zones or daylight saving time are ignored, all operations are assumed to take place in a common neutral time zone.

# Enumeration Type Documentation

---

## **enum dateMonths\_t**

---

Enumeration of months.

Notice that **DATE\_JAN** is 1, there is no month zero.

Enumerator	
DATE_JAN	January.
DATE_FEB	February.
DATE_MAR	March.
DATE_APR	April.
DATE_MAY	May.
DATE_JUN	June.
DATE_JUL	July.
DATE_AUG	August.
DATE_SEP	September.
DATE_OCT	October.
DATE_NOV	November.
DATE_DEC	December.

## **enum dateWeekdays\_t**

---

Enumeration of days of a week.

Enumerator	
DATE_SUN	Sunday.
DATE_MON	Monday.

DATE_TUE	Tuesday.
DATE_WED	Wednesday.
DATE_THU	Thursday.
DATE_FRI	Friday.
DATE_SAT	Saturday.

# Function Documentation

---

```
int dateAddDays ( dateDate_t * d,  
                  int          ndays  
                )
```

---

Add a number of days to a given date.

## Parameters

[in, out] **d** Date to be modified in place.  
**ndays** Number of days to add, positive or negative.

## Returns

BASE\_ERR\_OK on success.  
BASE\_ERR\_INVALID\_PARAMETER if d == NULL or  
!dateIsValid(d).

```
int dateCompare ( const dateDate_t * a,  
                  const dateDate_t * b  
                )
```

---

Compare two dates.

## Precondition

dateIsValid(a) && dateIsValid(b)

## See also

[dateDiff\(\)](#)

## Parameters

**a** Left-hand of comparison.  
**b** Right-hand of comparison.

## Returns

- > 0 if a > b.
- == 0 if a == b or either a or b are invalid.
- < 0 if a < b.

```
int dateDiff ( const dateDate_t * a,  
               const dateDate_t * b  
             )
```

---

Return the difference, in days, between two dates.

Given the defined possible return values for this function, it is recommended to check if both a and b are valid before calling:

```
1| if (!dateIsValid(a) || !dateIsValid(b))  
   error("invalid dates!");  
2| else return dateDiff(a, b);
```

## Precondition

dateIsValid(a) && dateIsValid(b)

## Parameters

- a** Minuend.
- b** Subtrahend.

## Returns

The difference a - b in days. This maybe a positive or negative number.

Zero if either a or b are invalid. Note that there is no way to differentiate this case from a == b, the caller must guarantee that the preconditions apply.

```
int dateFromJulianDay ( int           jdn,  
                         dateDate_t * d
```

)

Convert the Julian Day Number to the Gregorian calendar.

### See also

[http://en.wikipedia.org/wiki/Julian\\_day](http://en.wikipedia.org/wiki/Julian_day)

### Parameters

**jdn** The Julian Day Number to convert.  
[out] **d** Gregorian date.

### Returns

BASE\_ERR\_OK on success.  
BASE\_ERR\_INVALID\_PARAMETER if d == NULL or jdn < 0.

## int dateGet ( [dateDate\\_t](#) \* **d** )

Return the current system date.

### Parameters

[out] **d** Current system date.

### Returns

BASE\_ERR\_OK on success.  
BASE\_ERR\_INVALID\_PARAMETER if d == NULL.

## int dateIsLeap ( [uint16\\_t](#) **year** )

Return non-zero if given year number is a leap year.

### Parameters

**year** Year to check, with century (for example 1998 or 2011).

### Returns

Zero if year is *not* a leap year.

Non-zero if year is a leap year.

---

### **int dateIsValid ( const dateDate\_t \* d )**

---

Checks if a date is valid.

Takes into account leap years.

#### **Parameters**

**d** Date to verify.

#### **Returns**

Zero if d is invalid, this includes d == NULL.

Non-zero if d is valid.

---

### **int dateSet ( const dateDate\_t \* d )**

---

Update the system date.

#### **Parameters**

**d** Date to set.

#### **Returns**

BASE\_ERR\_OK on success.

BASE\_ERR\_INVALID\_PARAMETER if d == NULL or  
!dateIsValid(d).

BASE\_ERR\_ACCESS if the application does not have  
permission to change the system date.

---

### **int dateTimeAddSeconds ( dateTime\_t \* dt,                           int              nsecs                           )**

---

Add seconds to a given date and time.

## Parameters

[in, out] **dt** Date time to be modified in place.  
**nsecs** Number of seconds to add, positive or negative. The date component is also updated if necessary.

## Returns

BASE\_ERR\_OK on success.  
BASE\_ERR\_INVALID\_PARAMETER if dt == NULL or !timeIsValid(dt).

---

```
int dateTimeCompare ( const dateTime_t * a,  
                      const dateTime_t * b  
)
```

---

Compare two date times.

## Precondition

timeIsValid(a) && timeIsValid(b)

## Parameters

**a** Left-hand of comparison.  
**b** Right-hand of comparison.

## Returns

> 0 if a > b.  
== 0 if a == b or either a or b are invalid.  
< 0 if a < b.

---

```
int dateTimeDiff ( const dateTime_t * a,  
                   const dateTime_t * b  
)
```

---

Return the difference, in seconds, between two date times.

Given the defined possible return values for this function, it is recommended to check if both **a** and **b** are valid before calling:

```
1| if (!timeIsValid(a) || !timeIsValid(b))  
2|     error("invalid!");  
3| else return timeDiff(a, b);
```

### Precondition

`timeIsValid(a) && timeIsValid(b)`

### Parameters

**a** Minuend.

**b** Subtrahend.

### Note

Given the limited return range, this function may not work properly if **a** and **b** are too far way (about 68 years in 32-bit machines).

### Returns

The difference **a** - **b** in seconds. This maybe a positive or negative number.

Zero if either **a** or **b** are invalid. Note that there is no way to differentiate this case from **a == b**, the caller must guarantee that the preconditions apply.

## int dateTimeGet ( **dateTime\_t** \* **dt** )

---

Return the current system date and time.

### Parameters

[out] **dt** Current system date and time.

### Returns

`BASE_ERR_OK` on success.

`BASE_ERR_INVALID_PARAMETER` if **dt** == `NULL`.

---

## `uint64_t dateTimeGetMs ( void )`

Return an increasing count of milliseconds.

Given the architectural differences, this call does not guarantee *from when* the count starts. It may start on the device boot or may be the number of milliseconds since a certain past date.

Also, there is no guarantee that the resolution is effectively "thousands of seconds", it may be "hundreds of seconds", for example.

The only guarantee is that in two sequential calls, the second one will never return a value smaller than the first, and that the returned value increases in a constant frequency.

Given those limitations, this call is intended to be used as a chronometer, not to actually mark current time.

### Returns

Number of milliseconds since a certain date.

---

## `int dateTimeIsValid ( const dateTime_t * dt )`

Checks if both time and date are valid.

### Parameters

**dt** Date/time to verify.

### Returns

Zero if d is invalid, this includes d == NULL.

Non-zero if d is valid.

---

## `int dateTimeSet ( const dateTime_t * dt )`

Update the system date and time.

## Parameters

**dt** Date and time to set.

## Returns

BASE\_ERR\_OK on success.

BASE\_ERR\_INVALID\_PARAMETER if dt == NULL or  
!dateIsValid(dt).

BASE\_ERR\_ACCESS if the application does not have  
permission to change the system date.

## **int dateToJulianDay ( const dateDate\_t \* d )**

---

Return the Julian Day Number of d.

## See also

[http://en.wikipedia.org/wiki/Julian\\_day](http://en.wikipedia.org/wiki/Julian_day)

### Precondition

dateIsValid(d)

## Parameters

**d** Date to convert.

## Returns

The non-negative number of days since January 1, 4713 BC,  
aka Julian Day Number.

BASE\_ERR\_INVALID\_PARAMETER if d == NULL or  
!dateIsValid(d).

## **int dateWeekday ( const dateDate\_t \* d )**

---

Return the weekday of a given date.

### Precondition

dateIsValid(d)

## Parameters

**d** Date to be processed.

## Returns

Weekday of the date in d. See [dateWeekdays\\_t](#).

BASE\_ERR\_INVALID\_PARAMETER if d == NULL or  
!dateIsValid(d).

# LAR Library 1.14

Main Page	Related Pages	Modules	Data Structures
Files			
Data Structures	Data Fields		Data Fields
<h2>dateDate_t Struct Reference</h2> <p>Date and time</p>			

A gregorian date. [More...](#)

```
#include <date.h>
```

## Data Fields

---

**uint8\_t day**

Day of month, range [1,31].

**uint8\_t mon**

Month (see **dateMonths\_t**)

**uint16\_t year**

Non-negative year.

---

## Detailed Description

---

A gregorian date.

---

The documentation for this struct was generated from the following file:

- **date.h**
- 

Generated on Mon Mar 27 2017 15:42:53 for LAR Library by [\*\*doxygen\*\*](#) 1.8.9.1

# LAR Library 1.14

Main Page	Related Pages	Modules	Data Structures
Files			
Data Structures	Data Fields		Data Fields
<h2>dateTime_t Struct Reference</h2> <p>Date and time</p>			

A gregorian date with time. More...

```
#include <date.h>
```

## Data Fields

---

**dateDate\_t date**

Date information.

**uint8\_t hour**

Hour, range [0,23].

**uint8\_t min**

Minutes, range [0,59].

**uint8\_t sec**

Seconds, range [0,59].

---

## Detailed Description

---

A gregorian date with time.

---

The documentation for this struct was generated from the following file:

- **date.h**
- 

Generated on Mon Mar 27 2017 15:42:53 for LAR Library by [doxygen](#) 1.8.9.1

# LAR Library 1.14

Main Page

Related Pages

Modules

Data Structures

Files

[Typedefs](#) | [Functions](#)

## Double-ended queue

## TypeDefs

---

`typedef struct deque_t deque_t`

Opaque type for a double-ended queue.  
[More...](#)

---

## Functions

**deque\_t \* dequeCreate (int sizeHint)**

Create a new deque with space reserved for at least sizeHint elements. [More...](#)

**void dequeDestroy (deque\_t \*d)**

Destroy a deque created with **dequeCreate()**. [More...](#)

**int dequePushBack (deque\_t \*d, void \*p)**

Add an element to the "back" of a deque. [More...](#)

**int dequePopBack (deque\_t \*d)**

Remove the last element of a queue. [More...](#)

**int dequePushFront (deque\_t \*d, void \*p)**

Add an element on the "back" of a deque. [More...](#)

**int dequePopFront (deque\_t \*d)**

Remove the last element of a queue. [More...](#)

**int dequeLength (const deque\_t \*d)**

Return the number of elements inside a deque. [More...](#)

**void \* dequeGet (const deque\_t \*d, int i)**

Return the i'th element on a queue. [More...](#)

**void \* dequeGetFront (const deque\_t \*d)**

Return the first element on a deque. [More...](#)

**void \* dequeGetBack (const deque\_t \*d)**

Return the last element on a deque. [More...](#)

**void \* dequeSet (deque\_t \*d, int i, void \*p)**

Change the value of the i'th element on a deque. [More...](#)

int **dequeInsert** (**deque\_t** \*d, int i, void \*p)  
Insert an element at any position inside the deque.  
[More...](#)

---

int **dequeRemove** (**deque\_t** \*d, int i)  
Remove an element at any position inside the deque.  
[More...](#)

---

## Detailed Description

---

### Introduction

This module implements a double-ended queue (*deque*) of opaque elements that provides *amortized*  $O(1)$  push and pop operations on both ends (but not in the middle) and random item access (*get* and *set* operations).

Note that the implementation treats elements as opaque pointers, their memory is not automatically released when removed from the queue, the user must do that.

## Typedef Documentation

---

**typedef struct deque\_t deque\_t**

---

Opaque type for a double-ended queue.

All operations on deques receive a parameter of this type.

# Function Documentation

---

## `deque_t* dequeCreate( int sizeHint )`

---

Create a new deque with space reserved for at least `sizeHint` elements.

Creating a deque with `sizeHint` close to the maximum number of elements that will ever be inserted gives the best performance characteristics, since no re-allocation will ever be necessary.

### Parameters

**sizeHint** A hint on the maximum size of elements that will ever be inserted on the deque. The implementation may use a value larger than this on practice.

### Returns

A new `deque_t` instance, or `NULL` on error.

## `void dequeDestroy( deque_t * d )`

---

Destroy a deque created with `dequeCreate()`.

### Parameters

**d** deque to be destroyed. `NULL` is accepted, and ignored.

## `void* dequeGet( const deque_t * d,                   int              i                   )`

---

Return the `i`'th element on a queue.

### Note

An implementation may check boundaries and cause an assertion error on out of bounds access in debug builds. This check may be omitted in non-debug builds. In this case accessing an element out of bounds is undefined behavior. Calling with an invalid **d** parameter may cause an assertion error on debug builds or undefined behavior on non-debug builds.

### Precondition

(**d** != NULL) && (0 <= i < dequeLength(**d**))

### Parameters

- d** A valid deque\_t instance.
- i** Index of the element to be returned.

### Returns

The element at the **i**'th position on the deque **d**.

---

## void\* dequeGetBack ( const deque\_t \* **d** )

Return the last element on a deque.

This call is equivalent to `dequeGet(d, dequeLength(d) - 1)`. The same bound checking notes that apply to [dequeGet\(\)](#) are valid here.

### Note

Calling with an invalid **d** parameter may cause an assertion error on debug builds or undefined behavior on non-debug builds.

### Parameters

- d** A valid deque\_t instance.

### Returns

The element at the last position on the deque **d**.

## `void* dequeGetFront ( const deque_t * d )`

Return the first element on a deque.

This call is equivalent to `dequeGet(d, 0)`. The same bound checking notes that apply to [dequeGet\(\)](#) are valid here.

### Note

Calling with an invalid `d` parameter may cause an assertion error on debug builds or undefined behavior on non-debug builds.

### Parameters

`d` A valid `deque_t` instance.

### Returns

The element at the first position on the deque `d`.

```
int dequeInsert ( deque_t * d,
                  int      i,
                  void *   p
                )
```

Insert an element at any position inside the deque.

Even without any reallocation this is a  $O(n)$  operation, as other elements may need to be moved. Operations on either ends of the deque will behave as if [dequePushFront\(\)](#) and [dequePushBack\(\)](#) were called.

### Note

If there is not enough space on the `deque_t` for a new element it will be resized, which is also an  $O(n)$  operation.

### Parameters

`d` A valid `deque_t` instance.

`i` Index where to insert element. If this operation succeeds,

`dequeGet(d, i) == p.i == dequeLength(d)` is a valid index.

**p** Element to be added.

### Returns

`BASE_ERR_OK` if the element was inserted.

`BASE_ERR_INVALID_HANDLE` if `d` is an invalid `queue_t`.

`BASE_ERR_RESOURCE_PB` if could not expand deque to hold the new element.

`BASE_ERR_DATA_NOT_FOUND` if the index is out of range.

---

## `int dequeLength ( const deque_t * d )`

---

Return the number of elements inside a deque.

Note that this value is not related to the actual memory occupation of a deque. For example, a large `sizeHint` parameter to `dequeCreate()` will reserve much memory to the deque, but `dequeLength()` will return zero.

### Parameters

**d** A valid `deque_t` instance.

### Returns

The number of elements inside the deque  $\geq 0$ .

`BASE_ERR_INVALID_HANDLE` if `d` is an invalid `queue_t`.

---

## `int dequePopBack ( deque_t * d )`

---

Remove the last element of a queue.

### Parameters

**d** A valid `deque_t` instance.

### Returns

`BASE_ERR_OK` if the element was removed.

BASE\_ERR\_INVALID\_HANDLE if **d** is an invalid queue\_t.  
BASE\_ERR\_DATA\_NOT\_FOUND if the deque is empty.

## **int dequePopFront ( deque\_t \* d )**

---

Remove the last element of a queue.

### **Parameters**

**d** A valid deque\_t instance.

### **Returns**

BASE\_ERR\_OK if the element was removed.  
BASE\_ERR\_INVALID\_HANDLE if **d** is an invalid queue\_t.  
BASE\_ERR\_DATA\_NOT\_FOUND if the deque is empty.

## **int dequePushBack ( deque\_t \* d,                       void \*      p                       )**

---

Add an element to the "back" of a deque.

This is equivalent as inserting an element at position  
`dequeLength(d)`.

### **Note**

If there is not enough space on the deque\_t for a new element it will be resized, which is an O(n) operation. Otherwise, this is an O(1) operation.

### **Parameters**

**d** A valid deque\_t instance.

**p** Element to be added.

### **Returns**

BASE\_ERR\_OK if the element was inserted.

BASE\_ERR\_INVALID\_HANDLE if `d` is an invalid queue\_t.  
BASE\_ERR\_RESOURCE\_PB if could not expand deque to hold the new element.

```
int dequePushFront ( deque_t * d,  
                     void *      p  
)
```

---

Add an element on the "back" of a deque.

This is equivalent as inserting an element at position 0.

### Note

If there is not enough space on the deque\_t for a new element it will be resized, which is an O(n) operation. Otherwise, this is an O(1) operation.

### Parameters

- `d` A valid deque\_t instance.
- `p` Element to be added.

### Returns

BASE\_ERR\_OK if the element was inserted.  
BASE\_ERR\_INVALID\_HANDLE if `d` is an invalid queue\_t.  
BASE\_ERR\_RESOURCE\_PB if could not expand deque to hold the new element.

```
int dequeRemove ( deque_t * d,  
                  int        i  
)
```

---

Remove an element at any position inside the deque.

### Note

A check is made if the element is the first or last on the deque,

and `dequePopFront()` or `dequePopBack()` are called on those cases, operations on any other index are O(n).

## Parameters

- d** A valid `deque_t` instance.
- i** Index of element to be removed.

## Returns

- `BASE_ERR_OK` if the element was removed.
- `BASE_ERR_INVALID_HANDLE` if `d` is an invalid `queue_t`.
- `BASE_ERR_DATA_NOT_FOUND` if the index is out of range.

```
void* dequeSet ( deque_t * d,  
                 int      i,  
                 void *   p  
               )
```

---

Change the value of the `i`'th element on a deque.

## Note

An implementation may check boundaries and cause an assertion error on access out of bounds in debug builds. This check may be omitted in non-debug builds. In this case accessing an element out of bounds is undefined behavior.

Calling with an invalid `d` parameter may cause an assertion error on debug builds or undefined behavior on non-debug builds.

## Parameters

- d** A valid `deque_t` instance.
- i** Index of element to set.
- p** Value to be written.

## Returns

The value of the parameter `p`.

---

Generated on Mon Mar 27 2017 15:42:52 for LAR Library by [doxygen](#) 1.8.9.1

# LAR Library 1.14

Main Page	Related Pages	Modules	Data Structures
Files			
<p style="text-align: right;"><a href="#">Modules</a>   <a href="#">Data Structures</a>   <a href="#">Enumerations</a>   <a href="#">Functions</a></p>			
<h2>Buffer formatting</h2>			

# Modules

---

## **Field Formatting Functions**

Set of pre-defined formatting functions.

---

## Data Structures

---

struct **format\_t**

Overall parameters of a format session. [More...](#)

struct **formatField\_t**

Information about each field to be read or written. [More...](#)

---

## Enumerations

---

enum **formatDirection\_t** { **FORMAT\_PACK**, **FORMAT\_UNPACK** }

Used as parameter to the formatting functions to inform if they should handle packing or unpacking. [More...](#)

---

## Functions

---

int **formatPack** (**format\_t** \*fmt, **uint8\_t** \*output, int maxOutput)

Write to output the packed values as described in fmt. [More...](#)

int **formatUnpack** (**format\_t** \*fmt, **uint8\_t** \*input, int inputSize)

Read from input and unpack the fields to fmt. [More...](#)

---

## Detailed Description

---

### Rationale

One of the basic operations necessary to perform during communication and storage is generating a formatted buffer coalesced from many data items, perhaps with necessary conversions, for example ISO8583 message packing. This module is an extension of existing ISO8583 message-packing libraries, with additions so it can be used for other packing formats, and can be highly customized by the user.

# Introduction

*Formatting* (or *packing*) a buffer is the generation of a buffer from a collection of (many) data elements possibly spread in many data structures, with added conversion and formatting. The inverse process is called *unpacking*.

The design of this module is such that the user needs to create and maintain a single table with the list of fields, and how to format each, and this same structure can be used both for packing and unpacking. Some pre-defined functions for this are included with the library (see [Field Formatting Functions](#)), but the user is free to create new ones.

See the documentation of [formatField\\_t](#) and [format\\_t](#) for information on how to fill those structures. See below for more information about how each field is used during packing and unpacking.

# Packing

On a call to **formatPack()** the following algorithm is followed for each field:

1. If filter != NULL and filter(fmt, field) == 0, ignore this field
  2. Set bodySize = 0;
  3. If body != NULL, call it to write the field data;
    - Set bodySize to the return value of body;
  4. If prefix != NULL, call it;
    - Note that prefix can access bodySize to know the number of bytes actually written by body, and size for the user-provided size information;
  5. If suffix != NULL, call it;
    - Note that suffix can access bodySize to know the number of bytes actually written by body, and size for the user-provided size information;
  6. Swap body and prefix so prefix is before body on the output buffer.
- The formatting of the size and padding is independent of the formatting of the actual field data;
  - bodySize rarely needs to be filled by the user, it is filled on runtime with the value returned by body;
  - Fields that require padding to specific sizes can break this into a body that write the actual data and a prefix and/or suffix that deals with the padding;
  - On the call to prefix or suffix the size field indicates the number given by the caller when filling the **formatField\_t** structure, while bodySize is the actual number of bytes written by body;
  - Padding is usually done by a suffix that writes size - bodySize bytes;
  - prefix can be used to write a size field with the actual number of bytes written, as stored in bodySize;
  - A separator can be generated using the suffix function.

# Unpacking

On a call to **formatUnpack()** the following algorithm is followed for each field:

1. If filter != NULL and filter(fmt, field) == 0, ignore this field
  2. Set bodySize = size;
  3. If prefix != NULL, call it;
    - If prefix deals with the actual field length, it *must* fill bodySize with the number of bytes that need to be read!
  4. If body != NULL, call it to read the field data;
    - body should *always* read bodySize bytes, *not* size bytes;
  5. If suffix != NULL, call it;
    - Note that suffix can access bodySize to know the actual size of body in bytes;
- Remember that prefix should set the bodysize field, and that body must use bodySize instead of size to know how many bytes to read (if prefix is NULL then bodySize = size);
  - If suffix is used for padding or separator, then it may only skip the necessary number of bytes;

# **Examples**

Here we list a sample of common field types and a simple ISO8583-like field list with a few bits.

## Fixed-size Fields

Fixed-size fields are fields where the number of bytes written by `body` is always equal to `size` as given by the user on the field table. This usually means that those fields include no prefix or suffix, but they may require a `sufix` function to write a fixed separator.

Example: a fixed-size field with 16 bytes.

```
field.index = fieldIndex;
field.prefix = NULL;
field.body = formatBodyMemcpy;    // copy exactly
                                 field.size bytes
field.suffix = NULL;
field.size = 16;                  // size must be
                                 provided
field.data = dataPtr;
field.bodySize = 0;
```

## Variable-size Fields

Variable-size fields require a size prefix to indicate the actual size of the body. Usually they are string data (as card-holder name) or numerical data without padding.

Example: a variable-size field of a string.

```
field.index = fieldIndex;
field.prefix = formatSizeLL;      // size as 2-
    digit decimal number
field.body = formatBodyStrcpy;    // copy until
    '\0'
field.suffix = NULL;
field.size = 0;                  // size is not
    relevant
field.data = stringPtr;
field.bodySize = 0;
```

## Padded Fields

Padded fields are a combination of fixed and variable-sized fields. They have a size prefix but are padded to fit exactly `size` bytes.

Example: a string field padded with spaces to 20 bytes.

```
formatField_t field;
field.index = fieldIndex;
field.prefix = formatSizeLL;      // size as 2-
    digit decimal number
field.body = formatBodyStrcpy;    // copy until
    '\0'
field.suffix = formatPadSpaces;  // pad with
    spaces
field.size = 20;                  // desired total
    size of field
field.data = stringPtr;
field.bodySize = 0;
```

# Filtering

By filling the `filter` field of the `format_t` structure, the caller may allow conditional processing of fields. The most common use is during ISO8583 processing, but the system is flexible enough for other uses. During execution the `filter` function is called for each field, passing a pointer to both the `format_t` and the specific `formatField_t`.

A filter implementation may use the `fieldMap` pointer to store the set of allowed fields. For example, for ISO8583, the `fieldMap` could be a pointer to the actual ISO8583 bitmap as in the packed buffer.

A common framework is to use a body writer function that both pack/unpack the bitmap and set the `fieldMap` field, for example:

```
int filterISO8583(const format_t *fmt, const
    formatField_t *field)
{
    return (fmt->fieldMap == NULL)
        || (field->index < 0)
        || bitsGet(fmt->fieldMap, field->index) == 1;
}
```

Is a function that checks the bit field->index of fmt->fieldMap to choose if a field should or not be part of the packing/unpacking process.

In order for this to work, the user must also define, inside the field list, where the bitmap is stored, and properly associate the value of the `fieldMap` field. The following function does both:

```
if (direction == FORMAT_PACK) {
    memcpy(buffer, field->data, field->size);
    ret = field->size;
}
else {
    memcpy(field->data, buffer, field->bodySize);
    ret = field->bodySize;
}
fmt->fieldMap = field->data;
return ret;
}
```

And can be used as:

```
bitsBuffer_t iso_bitmap[NBITMAP];
formatField_t field = { -1, NULL,
    formatBodyISO8583, NULL, sizeof(iso_bitmap),
    iso_bitmap, 0 };
```

## A simple body function

This is a simple function that could be used as the body of a field. It copies variable-sized zero-terminated strings:

```
int formatBodyStrcpy(uint8_t *buffer,
    int limit,
    formatDirection_t direction,
    format_t *fmt,
    formatField_t *field)
{
    if (direction == FORMAT_PACK) {
        int n = strlen(field->data);
        memcpy(buffer, field->data, n);
        return n;
    }
    else {
        memcpy(field->data, buffer, field->bodySize);
        ((char *)field->data)[field->bodySize] = '\0';
        return field->bodySize;
    }
}
```

Note how it return `strlen()` on packing, and both uses and return `bodySize` on unpack.

## A simple prefix function

Following the example above, this is a prefix function that write the value of bodySize as a two-digit decimal number:

```
int formatSizeAsc2(uint8_t *buffer,
    int limit,
    formatDirection_t direction,
    format_t *fmt,
    formatField_t *field)
{
    if (direction == FORMAT_PACK) {
        if (limit < 2)
            return BASE_ERR_OVERFLOW;
        if (field->bodySize < 0 || field->bodySize > 99)
            return BASE_ERR_INVALID_PARAMETER;
        convIntToTxtPad(field->bodySize, (char *) buffer,
                        2, 10);
        return 2;
    }
    else {
        uint64_t value;
        convTxtToInt((const char *) buffer, 2, 10,
                     &value);
        field->bodySize = (int) value;
        return 2;
    }
}
```

Note how it return the number of bytes read or written on both pack and unpack, and how it uses bodysize on packing and stores to it on unpacking. (This version uses the functions from [Conversion routines](#)).

## A simple suffix function

Now a function that does padding with spaces: it writes enough spaces to compensate by the difference between `size` and `bodySize`, if any.

```
int formatPadSpaces(uint8_t *buffer,
                    int limit,
                    formatDirection_t direction,
                    format_t *fmt,
                    formatField_t *field)
{
    if (direction == FORMAT_PACK) {
        memset(buffer, ' ', field->size - field-
               >bodySize);
    }
    return field->size - field->bodySize;
}
```

Note how, on unpacking, the function ignores the actual buffer contents and just return how many bytes need to be skipped over.

## A (small) complete example

Putting together the pieces above we can create a simple formatter table for a buffer with only variable-sized string fields.

First, define the list of fields and their format:

```
char string1[NSTRING1];
char string2[NSTRING2];
char iso_bitmap[NBITMAP];
formatField_t fields[] = {
    // position of the bitmap on the formatted buffer
    { -1, NULL, formatBodyISO8583, NULL,
      sizeof(iso_bitmap), iso_bitmap, 0 },
    // no padding, with size prefix
    { 0, formatSizeLL, formatBodyStrcpy, NULL, 0,
      string1, 0 },
    // padding with spaces
    { 1, formatSizeLL, formatBodyStrcpy,
      formatPadSpaces, NSTRING2, string2, 0 },
};

```

Create the `format_t` instance with the fields defined above:

```
format_t fmt = {
    // list of fields
    fields,
    // number of fields
    sizeof(fields) / sizeof(fields[0]),
    // filter function
    &formatFilterISO8583,
    // the other fields are initialized to zero
};
```

Set the bits on `iso_bitmap` to identify which fields are part of the buffer. (When unpacking, the buffer is read from the input buffer).

```
// destination buffer
uint8_t output[NOUTPUT];
// clear list of bits
bitsSetRange(iso_bitmap, 0, 8 * NBITMAP, 0);
// enable field indexes 0 and 1
bitsSet(iso_bitmap, 0, 1);
bitsSet(iso_bitmap, 1, 1);
// call formatting
formatPack(&fmt, output, sizeof(output));
```

# Enumeration Type Documentation

---

## **enum formatDirection\_t**

---

Used as parameter to the formatting functions to inform if they should handle packing or unpacking.

Enumerator	
FORMAT_PACK	Write to buffer.
FORMAT_UNPACK	Read from buffer.

# Function Documentation

---

```
int formatPack (format_t * fmt,  
                uint8_t * output,  
                int       maxOutput  
)
```

---

Write to `output` the packed values as described in `fmt`.

## Parameters

`fmt` `format_t` with packing parameters.  
`output` Where to store the packed message.  
`maxOutput` Size of `output`, in bytes.

## Returns

Number of bytes written on success.

Negative on error.

```
int formatUnpack (format_t * fmt,  
                  uint8_t * input,  
                  int       inputSize  
)
```

---

Read from `input` and unpack the fields to `fmt`.

## Parameters

`fmt` `format_t` with packing parameters.  
`input` Where to read the packed message.  
`inputSize` Size of `input`, in bytes.

## Returns

Number of bytes actually read from `input`, on success.

Negative on error.

# LAR Library 1.14

Main Page	Related Pages	Modules	Data Structures
Files	Functions		

## Field Formatting Functions

[Buffer formatting](#)

Set of pre-defined formatting functions. [More...](#)

## Functions

int **formatFilterISO8583** (const **format\_t** \*fmt, const **formatField\_t** \*field)

Use this function as the filter function for a ISO8583 buffer.

[More...](#)

int **formatBodyISO8583** (**uint8\_t** \*buffer, int limit, **formatDirection\_t** direction, **format\_t** \*fmt, **formatField\_t** \*field)

Read/write the bitmap of a ISO8583 message. [More...](#)

int **formatBodyPacked** (**uint8\_t** \*buffer, int limit, **formatDirection\_t** direction, **format\_t** \*fmt, **formatField\_t** \*field)

Format a field as a recursive call to **formatPack()** and **formatUnpack()**. [More...](#)

int **formatBodySkip** (**uint8\_t** \*buffer, int limit, **formatDirection\_t** direction, **format\_t** \*fmt, **formatField\_t** \*field)

Skip field->size bytes from message. [More...](#)

int **formatBodyMemcpy** (**uint8\_t** \*buffer, int limit, **formatDirection\_t** direction, **format\_t** \*fmt, **formatField\_t** \*field)

Copy field->size bytes of field->data to/from buffer. [More...](#)

int **formatBodyStrcpy** (**uint8\_t** \*buffer, int limit, **formatDirection\_t** direction, **format\_t** \*fmt, **formatField\_t** \*field)

Formatting for variable-sized zero-terminated strings. [More...](#)

int **formatBodyUint64Dec** (**uint8\_t** \*buffer, int limit, **formatDirection\_t** direction, **format\_t** \*fmt, **formatField\_t** \*field)

Format an **uint64\_t** field as a decimal string. [More...](#)

int **formatBodyUint32Dec** (**uint8\_t** \*buffer, int limit, **formatDirection\_t** direction, **format\_t** \*fmt, **formatField\_t** \*field)

Format an **uint32\_t** field as a decimal string. [More...](#)

int **formatBodyUint16Dec** (**uint8\_t** \*buffer, int limit,  
**formatDirection\_t** direction, **format\_t** \*fmt, **formatField\_t** \*field)  
Format an **uint16\_t** field as a decimal string. [More...](#)

int **formatBodyUint8Dec** (**uint8\_t** \*buffer, int limit,  
**formatDirection\_t** direction, **format\_t** \*fmt, **formatField\_t** \*field)  
Format an **uint8\_t** field as a decimal string. [More...](#)

int **formatBodyUint64Buf** (**uint8\_t** \*buffer, int limit,  
**formatDirection\_t** direction, **format\_t** \*fmt, **formatField\_t** \*field)  
Format an **uint64\_t** field as a sequence of bytes. [More...](#)

int **formatBodyBufHex** (**uint8\_t** \*buffer, int limit, **formatDirection\_t**  
direction, **format\_t** \*fmt, **formatField\_t** \*field)  
Format an **buffer** field as a hexadecimal string. [More...](#)

int **formatSizeAsc2** (**uint8\_t** \*buffer, int limit, **formatDirection\_t**  
direction, **format\_t** \*fmt, **formatField\_t** \*field)  
Read or write the value of bodySize as a 2-digit base-10 string.  
[More...](#)

int **formatSizeAsc3** (**uint8\_t** \*buffer, int limit, **formatDirection\_t**  
direction, **format\_t** \*fmt, **formatField\_t** \*field)  
Read or write the value of bodySize as a 3-digit base-10 string.  
[More...](#)

int **formatSizeBcd2** (**uint8\_t** \*buffer, int limit, **formatDirection\_t**  
direction, **format\_t** \*fmt, **formatField\_t** \*field)  
Read or write the value of bodySize as a 2-digit BCD number (1  
byte). [More...](#)

int **formatSizeBcd3** (**uint8\_t** \*buffer, int limit, **formatDirection\_t**  
direction, **format\_t** \*fmt, **formatField\_t** \*field)  
Read or write the value of bodySize as a 3-digit BCD number (2  
bytes). [More...](#)

int **formatPadSpaces** (**uint8\_t** \*buffer, int limit, **formatDirection\_t** direction, **format\_t** \*fmt, **formatField\_t** \*field)  
Write or skip field->size - field->bodySize spaces. [More...](#)

int **formatPadZeroes** (**uint8\_t** \*buffer, int limit, **formatDirection\_t** direction, **format\_t** \*fmt, **formatField\_t** \*field)  
Write or skip field->size - field->bodySize ASCII '0' chars.  
[More...](#)

int **formatPadZeroBytes** (**uint8\_t** \*buffer, int limit, **formatDirection\_t** direction, **format\_t** \*fmt, **formatField\_t** \*field)  
Write or skip field->size - field->bodySize 0x00 bytes. [More...](#)

int **formatPadSkip** (**uint8\_t** \*buffer, int limit, **formatDirection\_t** direction, **format\_t** \*fmt, **formatField\_t** \*field)  
Skip exactly field->size - field->bodySize bytes. [More...](#)

---

## Detailed Description

---

Set of pre-defined formatting functions.

### | **Todo:**

Review the list of format functions and add new ones.

# Function Documentation

---

```
int formatBodyBufHex ( uint8_t *          buffer,
                      int               limit,
                      formatDirection_t direction,
                      format_t *        fmt,
                      formatField_t *   field
)
```

---

Format an `buffer` field as a hexadecimal string.

`field->data` must point to an `char` array, its values will be read and written as a string in base-16.

## Note

No padding is done, the return is the number of hexadecimal digits actually used. Use `prefix` or `suffix` for padding.

## Parameters

**buffer** Output buffer.  
**limit** Max number of bytes to write to `buffer`.  
**direction** `FORMAT_PACK` or `FORMAT_UNPACK`  
**fmt** Formatting context  
**field** Information about the field.

## Returns

Number of bytes actually written to `buffer`, or number of bytes read.

Negative number on error.

```
int formatBodyISO8583 ( uint8_t *          buffer,
                        int               limit,
```

```
    formatDirection_t direction,  
    format_t *          fmt,  
    formatField_t *     field  
)
```

Read/write the bitmap of a ISO8583 message.

field->size *must* include the size of the bitmap, in bytes (8 for 64-bits, 16 for 128-bits).

### Parameters

**buffer** Output buffer.  
**limit** Max number of bytes to write to buffer.  
**direction** FORMAT\_PACK or FORMAT\_UNPACK  
**fmt** Formatting context  
**field** Information about the ISO8583 bitmap.

### Returns

Positive number of bytes actually read or written.

Negative number on error.

```
int formatBodyMemcpy ( uint8_t *          buffer,  
                      int                limit,  
                      formatDirection_t direction,  
                      format_t *          fmt,  
                      formatField_t *     field  
)
```

Copy field->size bytes of field->data to/from buffer.

### Parameters

**buffer** Output buffer.  
**limit** Max number of bytes to write to buffer.  
**direction** FORMAT\_PACK or FORMAT\_UNPACK  
**fmt** Formatting context

**field** Information about the field.

### Returns

Positive number of bytes actually read or written (always `field->size`).  
Negative number on error.

```
int formatBodyPacked ( uint8_t * buffer,
                      int limit,
                      formatDirection_t direction,
                      format_t * fmt,
                      formatField_t * field
)
```

---

Format a field as a recursive call to `formatPack()` and `formatUnpack()`.

Packs the field by calling `formatPack(field->data, buffer, limit)` and unpacks the field by calling `formatUnpack(field->data, buffer, field->bodySize)`.

### Parameters

**buffer** Output buffer.

**limit** Max number of bytes to write to buffer.

**direction** `FORMAT_PACK` or `FORMAT_UNPACK`

**fmt** Formatting context

**field** Information about this field, `field->data` should point to a `format_t` instance.

### Returns

Positive number of bytes actually read or written.  
Negative number on error.

```
int formatBodySkip ( uint8_t * buffer,
```

```
    int          limit,  
    formatDirection_t direction,  
    format_t *      fmt,  
    formatField_t * field  
)
```

Skip `field->size` bytes from message.

### Parameters

**buffer** Output buffer.  
**limit** Max number of bytes to write to buffer.  
**direction** FORMAT\_PACK or FORMAT\_UNPACK  
**fmt** Formatting context  
**field** Information about the field.

### Returns

Positive number of bytes actually skipped (always `field->size`).

Negative number on error.

```
int formatBodyStrcpy ( uint8_t *          buffer,  
                      int             limit,  
                      formatDirection_t direction,  
                      format_t *       fmt,  
                      formatField_t * field  
)
```

Formatting for variable-sized zero-terminated strings.

### Parameters

**buffer** Output buffer.  
**limit** Max number of bytes to write to buffer.  
**direction** FORMAT\_PACK or FORMAT\_UNPACK  
**fmt** Formatting context  
**field** Information about the field.

## Returns

Number of bytes actually written to buffer, or number of bytes read (always bodySize).

Negative number on error.

```
int formatBodyUint16Dec ( uint8_t *          buffer,
                           int               limit,
                           formatDirection_t direction,
                           format_t *         fmt,
                           formatField_t *    field
                         )
```

---

Format an `uint16_t` field as a decimal string.

`field->data` must point to an `unsigned short`, its value will be read and written as a string in base-10.

## Note

No padding with zeroes is done, the return is the number of decimal digits actually used. Use `prefix` or `suffix` for padding. Also note that negative numbers are not supported!

## Parameters

<b>buffer</b>	Output buffer.
<b>limit</b>	Max number of bytes to write to buffer.
<b>direction</b>	FORMAT_PACK or FORMAT_UNPACK
<b>fmt</b>	Formatting context
<b>field</b>	Information about the field.

## Returns

Number of bytes actually written to buffer, or number of bytes read.

Negative number on error.

```
int formatBodyUint32Dec ( uint8_t *          buffer,
```

```
    int          limit,  
    formatDirection_t direction,  
    format_t *      fmt,  
    formatField_t * field  
)
```

Format an `uint32_t` field as a decimal string.

`field->data` must point to an `unsigned int`, its value will be read and written as a string in base-10.

### Note

No padding with zeroes is done, the return is the number of decimal digits actually used. Use `prefix` or `suffix` for padding. Also note that negative numbers are not supported!

### Parameters

**buffer** Output buffer.  
**limit** Max number of bytes to write to `buffer`.  
**direction** `FORMAT_PACK` or `FORMAT_UNPACK`  
**fmt** Formatting context  
**field** Information about the field.

### Returns

Number of bytes actually written to `buffer`, or number of bytes read.

Negative number on error.

```
int formatBodyUint64Buf ( uint8_t *          buffer,  
                          int            limit,  
                          formatDirection_t direction,  
                          format_t *      fmt,  
                          formatField_t * field  
)
```

Format an `uint64_t` field as a sequence of bytes.

`field->data` must point to an `uint64_t`, its value will be read and written as a sequence of bytes in *big-endian* order.

## Parameters

<b>buffer</b>	Output buffer.
<b>limit</b>	Max number of bytes to write to buffer.
<b>direction</b>	<code>FORMAT_PACK</code> or <code>FORMAT_UNPACK</code>
<b>fmt</b>	Formatting context
<b>field</b>	Information about the field.

## Returns

Number of bytes actually written to `buffer`, or number of bytes read.

Negative number on error

```
int formatBodyUint64Dec ( uint8_t * buffer,
                           int limit,
                           formatDirection_t direction,
                           format_t * fmt,
                           formatField_t * field
                         )
```

Format an `uint64_t` field as a decimal string.

`field->data` must point to an `uint64_t`, its value will be read and written as a string in base-10.

## Note

No padding with zeroes is done, the return is the number of decimal digits actually used. Use `prefix` or `suffix` for padding. Also note that negative numbers are not supported!

## Parameters

<b>buffer</b>	Output buffer.
<b>limit</b>	Max number of bytes to write to buffer.
<b>direction</b>	FORMAT_PACK or FORMAT_UNPACK
<b>fmt</b>	Formatting context
<b>field</b>	Information about the field.

## Returns

Number of bytes actually written to buffer, or number of bytes read.  
Negative number on error.

---

```
int formatBodyUint8Dec ( uint8_t *          buffer,
                        int               limit,
                        formatDirection_t direction,
                        format_t *        fmt,
                        formatField_t *   field
)
```

Format an `uint8_t` field as a decimal string.

`field->data` must point to an `unsigned char`, its value will be read and written as a string in base-10.

## Note

No padding with zeroes is done, the return is the number of decimal digits actually used. Use `prefix` or `suffix` for padding.  
Also note that negative numbers are not supported!

## Parameters

<b>buffer</b>	Output buffer.
<b>limit</b>	Max number of bytes to write to buffer.
<b>direction</b>	FORMAT_PACK or FORMAT_UNPACK
<b>fmt</b>	Formatting context
<b>field</b>	Information about the field.

## Returns

Number of bytes actually written to buffer, or number of bytes read.

Negative number on error.

---

```
int formatFilterISO8583 ( const format_t *      fmt,
                           const formatField_t * field
                         )
```

---

Use this function as the `filter` function for a ISO8583 buffer.

This function uses `field->index` as the index of a bit in `fmt->fieldMap` to check if this field should be read or written.

Always return non-zero (*allowed*) if `fieldMap` is NULL or `field->index` is negative.

## Parameters

**fmt** Formatting instance.

**field** Pointer to field data.

## Returns

Non-zero if `field` should be read or written.

Zero if not.

---

```
int formatPadSkip ( uint8_t *          buffer,
                     int              limit,
                     formatDirection_t direction,
                     format_t *        fmt,
                     formatField_t *   field
                   )
```

---

Skip exactly `field->size - field->bodySize` bytes.

## Attention

Nothing is written to the output buffer during FORMAT\_PACK!

## Parameters

- buffer** Output buffer.
- limit** Max number of bytes to write to buffer.
- direction** FORMAT\_PACK or FORMAT\_UNPACK
- fmt** Formatting context
- field** Information about the field.

## Returns

field->size - field->bodySize

---

```
int formatPadSpaces ( uint8_t *          buffer,
                      int               limit,
                      formatDirection_t direction,
                      format_t *        fmt,
                      formatField_t *   field
)
```

Write or skip field->size - field->bodySize spaces.

## Parameters

- buffer** Output buffer.
- limit** Max number of bytes to write to buffer.
- direction** FORMAT\_PACK or FORMAT\_UNPACK
- fmt** Formatting context
- field** Information about the field.

## Returns

field->size - field->bodySize

Negative number on error.

---

```
int formatPadZeroBytes ( uint8_t *          buffer,
```

```
    int          limit,
    formatDirection_t direction,
    format_t *      fmt,
    formatField_t * field
)
```

Write or skip `field->size - field->bodySize` 0x00 bytes.

### Parameters

**buffer** Output buffer.  
**limit** Max number of bytes to write to buffer.  
**direction** FORMAT\_PACK or FORMAT\_UNPACK  
**fmt** Formatting context  
**field** Information about the field.

### Returns

`field->size - field->bodySize`

Negative number on error.

```
int formatPadZeroes ( uint8_t *          buffer,
                      int             limit,
                      formatDirection_t direction,
                      format_t *       fmt,
                      formatField_t *  field
)
```

Write or skip `field->size - field->bodySize` ASCII '0' chars.

### Parameters

**buffer** Output buffer.  
**limit** Max number of bytes to write to buffer.  
**direction** FORMAT\_PACK or FORMAT\_UNPACK  
**fmt** Formatting context  
**field** Information about the field.

## Returns

`field->size - field->bodySize`

Negative number on error.

---

```
int formatSizeAsc2 ( uint8_t *          buffer,
                     int               limit,
                     formatDirection_t direction,
                     format_t *        fmt,
                     formatField_t *   field
)
```

---

Read or write the value of `bodySize` as a 2-digit base-10 string.

## Parameters

**buffer** Output buffer.

**limit** Max number of bytes to write to `buffer`.

**direction** `FORMAT_PACK` or `FORMAT_UNPACK`

**fmt** Formatting context

**field** Information about the field. Only `bodySize` is used.

## Returns

Number of bytes actually written to `buffer`, or number of bytes read.

Negative number on error.

---

```
int formatSizeAsc3 ( uint8_t *          buffer,
                     int               limit,
                     formatDirection_t direction,
                     format_t *        fmt,
                     formatField_t *   field
)
```

---

Read or write the value of `bodySize` as a 3-digit base-10 string.

## Parameters

**buffer** Output buffer.  
**limit** Max number of bytes to write to buffer.  
**direction** FORMAT\_PACK or FORMAT\_UNPACK  
**fmt** Formatting context  
**field** Information about the field. Only bodySize is used.

## Returns

Number of bytes actually written to buffer, or number of bytes read.  
Negative number on error.

```
int formatSizeBcd2( uint8_t * buffer,  
                     int limit,  
                     formatDirection_t direction,  
                     format_t * fmt,  
                     formatField_t * field  
)
```

---

Read or write the value of bodySize as a 2-digit BCD number (1 byte).

## Parameters

**buffer** Output buffer.  
**limit** Max number of bytes to write to buffer.  
**direction** FORMAT\_PACK or FORMAT\_UNPACK  
**fmt** Formatting context  
**field** Information about the field. Only bodySize is used.

## Returns

Number of bytes actually written to buffer, or number of bytes read.  
Negative number on error.

```
int formatSizeBcd3 ( uint8_t *          buffer,
                     int               limit,
                     formatDirection_t direction,
                     format_t *        fmt,
                     formatField_t *   field
)
```

---

Read or write the value of `bodysize` as a 3-digit BCD number (2 bytes).

### Parameters

<b>buffer</b>	Output buffer.
<b>limit</b>	Max number of bytes to write to <code>buffer</code> .
<b>direction</b>	<code>FORMAT_PACK</code> or <code>FORMAT_UNPACK</code>
<b>fmt</b>	Formatting context
<b>field</b>	Information about the field. Only <code>bodysize</code> is used.

### Returns

Number of bytes actually written to `buffer`, or number of bytes read.

Negative number on error.

# LAR Library 1.14

Main Page	Related Pages	Modules	Data Structures
Files			
Data Structures	Data Fields	Data Fields	
Buffer formatting		Data Fields	
<h2>format_t Struct Reference</h2> <p>Overall parameters of a format session. More...</p>			

```
#include <format.h>
```

## Data Fields

**formatField\_t \*** **fields**

Pointer to array of fields to be processed.

**int** **nfields**

Number of entries in **fields**.

**int(\*) filter** )(const **format\_t** \*fmt, const **formatField\_t** \*field)

This function is called before each field is processed, receiving a pointer to the field being processed and the overall **format\_t**. More...

**void \*** **fieldMap**

Should point to an array with the information that is read by **filter** to decide if a field is or not at the buffer.

**int** **errorCode**

Last error code during pack / unpack.

**formatField\_t \*** **errorField**

Pointer to **formatField\_t** where error was detected.  
More...

## Detailed Description

---

Overall parameters of a format session.

# Field Documentation

---

## **formatField\_t\*** **format\_t::errorField**

---

Pointer to **formatField\_t** where error was detected.

NULL if no error was found during processing, or error was outside a field.

## **int(\* format\_t::filter) (const format\_t \*fmt, const formatField\_t \*field)**

---

This function is called before each field is processed, receiving a pointer to the field being processed and the overall **format\_t**.

It should return *non-zero* if the field should be written or read and zero if not.

### Parameters

**fmt** Descriptor of format instance.

**field** Descriptor of field to be formatted.

### Returns

Non-zero if **field** should be read/written.

Zero if **field** should be ignored.

---

The documentation for this struct was generated from the following file:

- **format.h**
-

# LAR Library 1.14

Main Page	Related Pages	Modules	Data Structures
Files			
Data Structures	Data Fields		
			Data Fields
<h2>formatField_t Struct Reference</h2> <p>Buffer formatting</p>			

Information about each field to be read or written. [More...](#)

```
#include <format.h>
```

## Data Fields

int **index**

Index of this field (usually on `fieldMap` in the parent `format_t`).  
[More...](#)

int(\*) **prefix** )(`uint8_t` \*buffer, int limit, `formatDirection_t` direction,  
`format_t` \*fmt, `formatField_t` \*field)  
Pack/unpack prefix of field.

int(\*) **body** )(`uint8_t` \*buffer, int limit, `formatDirection_t` direction,  
`format_t` \*fmt, `formatField_t` \*field)  
Pack/unpack actual field contents.

int(\*) **suffix** )(`uint8_t` \*buffer, int limit, `formatDirection_t` direction,  
`format_t` \*fmt, `formatField_t` \*field)  
Pack/unpack suffix of field.

int **size**

Default size of the field. [More...](#)

void \* **data**

Pointer to actual field data. [More...](#)

int **bodySize**

Actual size of the field, as returned by body. [More...](#)

## Detailed Description

---

Information about each field to be read or written.

# Field Documentation

---

## **int formatField\_t::bodySize**

---

Actual size of the field, as returned by body.

Default to the same value of `size`.

## **void\* formatField\_t::data**

---

Pointer to actual field data.

Should be filled by the `body` function.

## **int formatField\_t::index**

---

Index of this field (usually on `fieldMap` in the parent [format\\_t](#)).

If not used should be set to -1.

## **int formatField\_t::size**

---

Default size of the field.

Should be filled by user, zero if not relevant.

---

The documentation for this struct was generated from the following file:

- [format.h](#)
-

Generated on Mon Mar 27 2017 15:42:53 for LAR Library by [doxygen](#) 1.8.9.1

# LAR Library 1.14

Main Page	Related Pages	Modules	Data Structures	
Files	File System			
	<a href="#">Typedefs</a>   <a href="#">Enumerations</a>   <a href="#">Functions</a>			

## Typedefs

---

```
typedef struct fsFile_t fsFile_t  
    Opaque type of a file.
```

---

## Enumerations

---

```
enum fsOpenFlags_t {  
    FS_OPEN_READ = 1, FS_OPEN_WRITE = 2,  
    FS_OPEN_CREATE = 4, FS_OPEN_RESET = 8,  
    FS_OPEN_APPEND = 16  
}  
Flags for fsOpen(). More...
```

```
enum fsSeekWhence_t { FS_WHENCE_SET, FS_WHENCE_END,  
    FS_WHENCE_CUR }  
Possible seek positions, see fsSeek(). More...
```

---

## Functions

int **fsSetFileNamePrefix** (const char \*prefix)

Set the common file name prefix. [More...](#)

int **fsPushFileNamePrefix** (const char \*prefix)

Push a new item on the file name prefix stack. [More...](#)

int **fsPopFileNamePrefix** (void)

Remove the top item on the file name prefix stack.

[More...](#)

const char \* **fsGetFileNamePrefix** (void)

Return the current prefix being used. [More...](#)

int **fsOpen** (const char \*name, int flags, **fsFile\_t** \*\*file)

Open the selected file to input and/or output. [More...](#)

int **fsClose** (**fsFile\_t** \*f)

Close a file handle. [More...](#)

int **fsRead** (**fsFile\_t** \*f, int n, void \*p)

Try to read a number of bytes from a file. [More...](#)

int **fs.ReadByte** (**fsFile\_t** \*f)

Try to read the next byte from the file. [More...](#)

int **fsReadUntil** (**fsFile\_t** \*f, const **uint8\_t** \*delims, int ndelim, int n, void \*p)

Read bytes from the input file until either one of delims bytes is found, or n bytes have been read. [More...](#)

int **fsWrite** (**fsFile\_t** \*f, int n, const void \*p)

Try to write a number of bytes to a file. [More...](#)

int **fsWriteByte** (**fsFile\_t** \*f, int b)  
Try to write a byte to a file. [More...](#)

int **fsTell** (**fsFile\_t** \*f)  
Return the current read/write position of a file. [More...](#)

int **fsSeek** (**fsFile\_t** \*f, int whence, int offset)  
Set the read/write position for a file. [More...](#)

int **fsLength** (**fsFile\_t** \*f)  
Return the length of an open file in bytes. [More...](#)

int **fsRemove** (const char \*name)  
Delete file name from the file system. [More...](#)

int **fsRename** (const char \*oldname, const char \*newname)  
Rename the file oname to newname. [More...](#)

int **fsExist** (const char \*name)  
Check if file name exists. [More...](#)

---

## Detailed Description

---

## Rationale

File system API differs heavily between platforms. But more than 90% of all uses require the same simple basic functions. By defining them in a portable manner we can handle most porting issues without needing complicated implementations on simpler platforms.

# Introduction

This module defines a basic abstraction over the common operations of all file systems. Functionality is added only if easily implementable on *all* supported file systems, so it may not be enough/adequate for certain operations, specially in case of access to some platform-dependent meta-data.

The sort of operations covered here are:

- Opening files;
- Basic I/O (block/string oriented I/O);

Note that some directory-related operations are not included (change current directory, create/remove directory). In place of this, it is used the concept of *file name prefix* (see [fsSetNamePrefix\(\)](#)). Each operation on files names will prepend this prefix to the file name before operating. This allows for quick adaptation between architectures that have different concepts for directory/volume/working-unit.

The file name prefixes are also stored as a stack, allowing entries to be "pushed" and "poped" (see [fsPushFileNamePrefix\(\)](#) and [fsPopFileNamePrefix\(\)](#)), this way a code that temporarily needs to set the file prefix do not need to store the current one on a temporary variable to restore later.

## Telium

On *Telium* architecture, use the file name prefix to define which volume to use, but note that:

- The prefix string should include the full volume name, terminating with '/': "/VOLUME/";
- The volume is *not* automatically created, this must be done outside this library

# **Unicapt**

Global files are currently not supported on Unicapt32.

## **| Todo:**

Directory iteration

# Enumeration Type Documentation

---

## **enum fsOpenFlags\_t**

---

Flags for **fsOpen()**.

The `FS_OPEN_READ` and `FS_OPEN_WRITE` options are not strict, i.e., a file open for reading may have write allowed and a file open for writing may have read allowed. The strictest permission possible for a given architecture will be used.

Enumerator	
<code>FS_OPEN_READ</code>	Open file for reading.
<code>FS_OPEN_WRITE</code>	Open file for writing.
<code>FS_OPEN_CREATE</code>	Create file if not found.
<code>FS_OPEN_RESET</code>	Reset file if present.
<code>FS_OPEN_APPEND</code>	Move position to <code>FS_WHENCE_END</code> after open, forces <code>FS_OPEN_WRITE</code> .

## **enum fsSeekWhence\_t**

---

Possible seek positions, see **fsSeek()**.

Enumerator	
<code>FS_WHENCE_SET</code>	Seek from the start of file.
<code>FS_WHENCE_END</code>	Seek from the end of file.
<code>FS_WHENCE_CUR</code>	Seek from the current position.

# Function Documentation

---

## **int fsClose ( fsFile\_t \* f )**

---

Close a file handle.

### **Parameters**

**f** Handle to open file.

### **Returns**

BASE\_ERR\_OK on success.

BASE\_ERR\_INVALID\_HANDLE if **f** is invalid or NULL.

### **See also**

[fsOpen](#)

## **int fsExist ( const char \* name )**

---

Check if file name exists.

### **Parameters**

**name** Name of file to check.

### **Returns**

Non-zero if file exists

Zero otherwise

### **Attention**

The fact that the file exists does *not* imply that it can be opened by the caller application!

## **const char\* fsGetFileNamePrefix ( void )**

---

Return the current prefix being used.

### Returns

Pointer to the prefix in use. The returned string *must* not be changed, use [fsSetFileNamePrefix\(\)](#) to do this instead.

### See also

[fsSetFileNamePrefix](#)

## int fsLength ( [fsFile\\_t](#) \* f )

---

Return the length of an open file in bytes.

The current file pointer is not changed.

This is equivalent to:

```
1 int fileSize(fsFile_t *f) {
2     // FIXME: no error handling!
3     int orig = fsTell(f);
4     int siz = fsSeek(f, FS_WHENCE_END, 0);
5     fsSeek(f, FS_WHENCE_SET, orig);
6     return siz;
7 }
```

### Parameters

**f** Handle to open file.

### Returns

Non-negative size of file *f*.

BASE\_ERR\_INVALID\_HANDLE if *f* is invalid.

BASE\_ERR\_ACCESS if this operation is not allowed on this type of file.

## int fsOpen ( const char \* name,

```
    int          flags,  
    fsFile_t ** file  
)
```

Open the selected file to input and/or output.

Note the double indirection on the `file` parameter, to use this function:

```
1 fsFile_t *file;  
2 int error = fsOpen(name, FS_OPEN_READ |  
3   FS_OPEN_WRITE | FS_OPEN_CREATE, &file);  
4 // use file...  
4 fsClose(file);
```

## Parameters

**name** File name. The actual name sent to the OS has the value defined by [fsSetFileNamePrefix\(\)](#) prepended.

**flags** File opening flags OR'ed together, see [fsOpenFlags\\_t](#). At least one of **FS\_OPEN\_READ** or **FS\_OPEN\_WRITE** must be set.

[out] **file** On successful return, a handle to the open file.

## Returns

`BASE_ERR_OK` on success.

`BASE_ERR_DATA_NOT_FOUND` if file was not found.

`BASE_ERR_RESOURCE_PB` if could not allocate a new handle.

`BASE_ERR_ACCESS` do not have permission to access file.

`BASE_ERR_OVERFLOW` if `name` (or the concatenation of `name` and `prefix`) is too large.

`BASE_ERR_INVALID_PARAMETER` if `name`, `flags` or `file` are invalid.

## See also

[fsClose](#)

---

### **int fsPopFileNamePrefix ( void )**

---

Remove the top item on the file name prefix stack.

If the file name prefix stack has at least two elements, this call will remove the "top" element of the stack. If the list has a single element, this call has no effect (and the current value is kept).

#### Returns

BASE\_ERR\_OK

---

### **int fsPushFileNamePrefix ( const char \* prefix )**

---

Push a new item on the file name prefix stack.

This call will change the value that [fsGetFileNamePrefix\(\)](#) returns but keep the previous values untouched. A call to [fsPopFileNamePrefix\(\)](#) may be used to restore previous values.

#### Parameters

**prefix** File name prefix to use.

#### Returns

BASE\_ERR\_OK on success

BASE\_ERR\_INVALID\_PARAMETER if prefix is not valid on current architecture.

BASE\_ERR\_OVERFLOW if prefix is too large, or if file name prefix stack is too long

---

### **int fsRead ( fsFile\_t \* f,               int          n,               void \*      p**

)

Try to read a number of bytes from a file.

Reads up to *n* bytes from the file, depending on availability. Never reads more than *n* bytes.

## Parameters

**f** Handle to open file.

**n** Number of bytes to read.

[out] **p** Where to store the bytes read. Must have place for upto *n* bytes.

## Returns

Number of bytes actually read (zero if no more bytes to read – end of file).

BASE\_ERR\_INVALID\_HANDLE if *f* is invalid.

BASE\_ERR\_ACCESS if do not have permission to read from *f*.

BASE\_ERR\_INVALID\_PARAMETER if *p* is NULL.

## See also

[fs.ReadByte](#) [fs.ReadUntil](#)

## int fs.ReadByte ( [fsFile\\_t](#) \* **f** )

Try to read the next byte from the file.

## Parameters

**f** Handle to open file.

## Returns

Non-negative value of byte read, a return less than zero signals error.

BASE\_ERR\_INVALID\_HANDLE if *f* is invalid.

BASE\_ERR\_ACCESS if do not have permission to read from *f*.

BASE\_ERR\_DATA\_NOT\_FOUND if reached end of file.

## See also

[fsRead](#) [fsReadUntil](#)

```
int fsReadUntil( fsFile_t * f,  
                  const uint8_t * delims,  
                  int ndelim,  
                  int n,  
                  void * p  
)
```

Read bytes from the input file until either one of `delims` bytes is found, or `n` bytes have been read.

The delimiter character is included on the output buffer.

Note that `p` is not zero-terminated, the caller must use the return from this function to determine the number of bytes written to `p` and place the terminating zero if necessary.

Example:

```
1| char buf[N];  
2| int nread;  
3| nread = fsReadUntil(f, "\n", -1, N - 1,  
buf);  
4| if (nread >= 0) buf[nread] = '\0'; //  
force zero-termination
```

## Parameters

**f** Handle to open file.

**delims** Bytes to use as delimiters.

**ndelim** Number of bytes in `delims`. May be negative to use `strlen(delims)`.

**n** Read upto `n` bytes from input.

**[out] p** Where to store the data read, must have space for upto n bytes.

### Returns

Non-negative number of bytes read (zero if end of file).  
BASE\_ERR\_INVALID\_HANDLE if f is invalid.  
BASE\_ERR\_ACCESS if do not have permission to read from f.  
BASE\_ERR\_INVALID\_PARAMETER if delims or p are NULL.

### See also

[fsRead](#) [fsReadByte](#)

---

## int fsRemove ( const char \* name )

Delete file name from the file system.

The behavior is undefined if the file is currently open.

### Parameters

**name** Name of file to delete.

### Returns

BASE\_ERR\_OK if file was deleted.  
BASE\_ERR\_DATA\_NOT\_FOUND if name does not exist.  
BASE\_ERR\_ACCESS if could not access file (in use?)  
BASE\_ERR\_RESOURCE\_PB on file system access error

---

## int fsRename ( const char \* oldname,                  const char \* newname                  )

Rename the file oname to newname.

Both oldname and newname should not be open. The behavior is

undefined if `oldname` and `newname` refer to different directories or volumes (most implementations will fail).

## Parameters

**oldname** Original (current) name of file.  
**newname** New name of file.

## Returns

`BASE_ERR_OK` if file was deleted.  
`BASE_ERR_DATA_NOT_FOUND` if `oldname` does not exist.  
`BASE_ERR_ACCESS` if could not access file (in use?)  
`BASE_ERR_RESOURCE_PB` on file system access error

```
int fsSeek ( fsFile_t * f,  
            int      whence,  
            int      offset  
        )
```

---

Set the read/write position for a file.

Trying to move before the start of the file or after one past the end of the file has undefined behavior, it may fail or move the position to one of the edges of the file, depending on the platform. Do *not* depend on either behavior.

## Parameters

**f** Handle to open file.  
**whence** Where to seek from, see [fsSeekWhence\\_t](#).  
**offset** Distance to seek, positive to seek forward, negative to seek backward.

## Returns

Non-negative new position inside the file.  
`BASE_ERR_INVALID_HANDLE` if `f` is invalid.  
`BASE_ERR_ACCESS` if this operation is not allowed on this type of file.

BASE\_ERR\_INVALID\_PARAMETER if whence is not  
**FS\_WHENCE\_SET**, **FS\_WHENCE\_END** or  
**FS\_WHENCE\_CUR**.

## See also

[fsTell](#)

---

## int **fsSetFileNamePrefix** ( const char \* **prefix** )

Set the common file name prefix.

Each operation on files by name prepend this prefix (without any processing!) before calling the OS.

### Parameters

**prefix** File name prefix to use.

### Returns

BASE\_ERR\_OK on success.

BASE\_ERR\_INVALID\_PARAMETER if prefix is not valid on current architecture.

BASE\_ERR\_OVERFLOW if prefix is too large.

## See also

[fsGetFileNamePrefix](#) [fsOpen](#)

---

## int **fsTell** ( **fsFile\_t** \* **f** )

Return the current read/write position of a file.

### Parameters

**f** Handle to open file.

### Returns

Non-negative file position on success.

BASE\_ERR\_INVALID\_HANDLE if f is invalid.

`BASE_ERR_ACCESS` if this operation is not allowed on this type of file.

**See also**

[fsSeek](#)

---

```
int fsWrite ( fsFile_t * f,  
             int n,  
             const void * p  
 )
```

Try to write a number of bytes to a file.

Writes upto to `n` bytes to the file. Depending on available space may not be able to write all the requested bytes. Writing on the end of the file to append is allowed.

**Parameters**

- `f` Handle to open file.
- `n` Number of bytes to write.
- `p` Buffer with data to write.

**Returns**

- Number of bytes written on success.
- `BASE_ERR_INVALID_HANDLE` if `f` is invalid.
- `BASE_ERR_ACCESS` if do not have permission to write to `f`.

**See also**

[fsWriteByte](#)

---

```
int fsWriteByte ( fsFile_t * f,  
                  int b  
 )
```

Try to write a byte to a file.

## Parameters

- f** Handle to open file.
- b** Byte value to write. Only the least significant 8-bits are used, truncating **b** to the range [0, 255].

## Returns

- BASE\_ERR\_OK if byte was written.
- BASE\_ERR\_INVALID\_HANDLE if **f** is invalid.
- BASE\_ERR\_ACCESS if do not have permission to write to **f**.
- BASE\_ERR\_RESOURCE\_PB if could not write byte.

## See also

[fsWrite](#)

# LAR Library 1.14

Main Page	Related Pages	Modules	Data Structures
Files	Data Structures   Typedefs   Functions		
<b>INI file parsing</b>			

## Data Structures

---

struct **iniElement\_t**

Descriptor of elements allowed in an INI file. [More...](#)

---

## Typedefs

---

typedef char \*(\* **iniLineReader\_t**) (void \*p, char \*line, int maxline)

Signature of function used by **iniParseLines()** to  
read next line from input file. More...

---

## Functions

---

int **iniParse** (**iniElement\_t** elements[], const char \*fileName)

Process the given INI file and calls handler functions according to the element table given. [More...](#)

int **iniParseString** (**iniElement\_t** elements[], const char \*contents)

Process a previously INI file loaded in memory. [More...](#)

int **iniParseLines** (**iniElement\_t** elements[], **iniLineReader\_t**

readLine, void \*p)

Process the lines of an INI file returned from repeated calls to readLine. [More...](#)

---

## Detailed Description

---

### Rationale

INI files are a common and simple way to specify application parameters in a user-readable and editable manner. Even if the application does not use INI files internally to store configuration data, it can use it to read external user parameters and convert to an internal format.

# Introduction

Read/parsing of MS-like INI files.

Only a subset of INI file syntax is accepted:

- Comments start with ';' and must be the first non-space char on a line. Also, comments must be on a line on their own;
- Key names may include spaces in the middle;
- Values may be literals (*value*) or strings ("value"), literals have spaces before and after trimmed, strings are "as-is";
- Values are optional, they are considered NULL if not present;
- Keys and Groups are **not** case-sensitive;
- Groups are marked using [group name] and spaces inside the [] are *important*.

For example, the pair key name = key value will be parsed as "key name" is assigned the value of "key value", whereas key name = " key value " will have a value of " key value " (including the spaces, but not the quotes).

Example:

```
{  
    int err;  
    iniElement_t elements[] = {  
        { "general", "name", handle_general_name, NULL }, // handle "name" key inside "[general]"  
        section  
        { NULL, "phone", handle_phone, NULL }, // handle "phone" key inside any section  
        { "generic", NULL, handle_generic_all, NULL }, // handle all keys inside "[generic]" section  
        { NULL, NULL, NULL, NULL } // list terminator  
    };  
    err = iniParse(elements, "myfile.ini");
```

```
if (err == BASE_ERR_OK) // success
else // fail
}
```

This is a valid INI file that matches the description above:

```
; example of INI file
phone = "handled by handle_phone"
[general]
name = "handled by handle_general_name"
[generic]
other = "handled by handle_generic_all"
[stuff]
phone = "also handled by handle_phone"
other = "this entry is ignored"
```

### Attention

Remember that the scanning is done from top to bottom, so a very generic entry near the top (i.e. with both key and group as NULL) will be matched before any more specific entries below!

# Typedef Documentation

---

**typedef char\*(\* iniLineReader\_t) (void \*p, char \*line, int maxline)**

---

Signature of function used by [iniParseLines\(\)](#) to read next line from input file.

User-defined functions *must* follow the signature and behavior defined here.

Each call to a iniLineReader\_t should read the next line of input. The p parameter is the same that is given to [iniParseLines\(\)](#) and should be used to identify the source of input.

## Parameters

- p** Opaque pointer passed as parameter to [iniParseLines\(\)](#)
- line** Where to store the line read
- maxline** Max number of chars to write to maxline

## Returns

line on success, NULL on error.

# Function Documentation

---

```
int iniParse ( iniElement_t elements[],  
               const char * fileName  
             )
```

---

Process the given INI file and calls handler functions according to the element table given.

For each key = value pair found in the input file, the list `elements` is scanned sequentially until a match is found, and the associated `handler()` function is called. See `iniElement_t` for information on how matching is done, specially when `iniElement_t::group` and `iniElement_t::key` are `NULL`.

## Parameters

**elements** List of allowed elements. An entry with `handler = NULL` must terminate the list.

**fileName** Name of INI file to parse.

## Returns

`BASE_ERR_OK` if file was processed OK.

`BASE_ERR_DATA_NOT_FOUND` if the file `fileName` was not found.

The error code returned by one of the handler functions.

```
int iniParseLines ( iniElement_t     elements[],  
                   iniLineReader_t  readLine,  
                   void *           p  
                 )
```

---

Process the lines of an INI file returned from repeated calls to

`readLine`.

This call behaves as [iniParse\(\)](#), but, instead of reading the input from a file, interprets whatever is returned by repeated calls to `readLine`.

See the documentation of [iniLineReader\\_t](#) for more information on how `readLine` should behave.

## Parameters

**elements** List of allowed elements. An entry with `handler = NULL` must terminate the list.

**readLine** Function called to read each line of input. Parsing terminates when `readLine` returns `NULL`.

**p** Passed as the first parameter to `readLine`. Not accessed in any other way.

## Returns

`BASE_ERR_OK` if file was processed OK.

`BASE_ERR_INVALID_PARAMETER` if `elements` or `readLine` are `NULL`

The error code returned by one of the handler functions.

---

```
int iniParseString ( iniElement\_t elements[],  
                      const char * contents  
)
```

---

Process a previously INI file loaded in memory.

This call behaves as [iniParse\(\)](#), but, instead of reading the input from a file, directly decodes `contents`, that should be a zero-terminated string with a valid INI file.

## Parameters

**elements** List of allowed elements. An entry with `handler = NULL` must terminate the list.

**contents** Zero-terminated string with the contents of the INI file.

## Returns

BASE\_ERR\_OK if file was processed OK.

BASE\_ERR\_INVALID\_PARAMETER if elements or contents are NULL

The error code returned by one of the handler functions.

# LAR Library 1.14

Main Page	Related Pages	Modules	Data Structures	
Files				
Data Structures	Data Fields	Data Fields		
INI file parsing			Data Fields	
<h2>iniElement_t Struct Reference</h2> <p>Descriptor of elements allowed in an INI file. <a href="#">More...</a></p>				

Descriptor of elements allowed in an INI file. [More...](#)

```
#include <ini.h>
```

## Data Fields

---

char \* **group**

Is this entry allowed inside which group? [More...](#)

char \* **key**

Name of element. [More...](#)

int(\* **handler** )(iniElement\_t \*ini, const char \***group**, const char \***key**, const char \*value)

Function that must be called to handle this element. [More...](#)

void \* **param**

An optional user-defined parameter that is not touched by the [iniParse\(\)](#) function.

---

## Detailed Description

---

Descriptor of elements allowed in an INI file.

Every element present in the INI file but not found here is ignored.

# Field Documentation

---

## **char\* iniElement\_t::group**

---

Is this entry allowed inside which group?

Entries with the same key, but different group are allowed and will be selected according to the group. If NULL this entry is allowed inside any group.

## **int(\* iniElement\_t::handler) (iniElement\_t \*ini, const char \*group, const char \*key, const char \*value)**

---

Function that must be called to handle this element.

If it returns any value different than zero, processing will terminate.

### Parameters

- ini** Pointer to which element in the `iniElements_t` table was matched.
- group** Which group we are inside (NULL if none).
- key** Key that is being set (value on the left of =).
- value** Value to be set to, will be NULL if the line had no =

### Returns

Zero on success, any other value interrupts INI file parsing.

## **char\* iniElement\_t::key**

---

Name of element.

If NULL any key that does not match any *previous* definition will

match.

---

The documentation for this struct was generated from the following file:

- [ini.h](#)
- 

Generated on Mon Mar 27 2017 15:42:53 for LAR Library by  1.8.9.1

# LAR Library 1.14

Main Page	Related Pages	Modules	Data Structures
Files	Macros   Typedefs   Enumerations   Functions		
Logging			

## Macros

---

```
#define LOG_ALL_CHANNELS 0, 256
```

This is a short-cut to define the parameters of all channels in one single call to **logSetChannels()**.

```
#define LOG_ASSERT(cond)
```

Assertion using larlib.log for output. More...

---

## Typedefs

typedef int(\* **logWriteFunction\_t**) (void \*context, **uint16\_t** channelLevel, const char \*msg, int msgLength)  
Type of the function that does the actual writing on a trace channel. [More...](#)

typedef int(\* **logDumpFunction\_t**) (**logWriteFunction\_t** write, void \*context, **uint16\_t** channelLevel, const void \*data, int dataLen)  
Type of function called to format the output on **logDump()**. [More...](#)

---

## Enumerations

---

```
enum LogLevel_t {
    LOG_DEBUG = 0x0000, LOG_INFO = 0x2000,
    LOG_WARNING = 0x4000, LOG_ERROR = 0x6000,
    LOG_CRITICAL = 0x8000, LOG_ALWAYS = 0xFF00
}
```

General tracing levels. [More...](#)

```
enum logReservedChannels_t {
    LOG_CH_BASE = 200, LOG_CH_CONV, LOG_CH_DATE,
    LOG_CH_FS,
    LOG_CH_INI, LOG_CH_MEM, LOG_CH_TAB,
    LOG_CH_STEP,
    LOG_CH_FORMAT, LOG_CH_TASK
}
```

List of reserved log channels. [More...](#)

---

## Functions

int **logWriteTeliumTrace** (void \*context, **uint16\_t** channelLevel, const char \*msg, int msgLength)

A **logWriteFunction\_t** that can be used as parameter to **logSetChannels()** and writes traces using Telium's trace() function. [More...](#)

int **logWriteTeliumRemoteDebugger** (void \*context, **uint16\_t** channelLevel, const char \*msg, int msgLength)

A **logWriteFunction\_t** that can be used as parameter to **logSetChannels()** and writes traces to the Remote Debugger console on IngeDev. [More...](#)

int **logDumpFormattedAscii** (**logWriteFunction\_t** write, void \*context, **uint16\_t** channelLevel, const void \*data, int dataLen)

dumpFormatFunction that formats the received buffer as a two-column hexadecimal and ASCII display and calls the received write function. [More...](#)

int **logSetChannels** (**uint8\_t** firstChannel, int numChannels, **uint16\_t** level, **logWriteFunction\_t** writeFunction, **logDumpFunction\_t** dumpFormatFunction, void \*context)

Configures one or more tracing channels. [More...](#)

int **logChannelsEnabled** (**uint16\_t** channelLevel)

Check if a log channel is enabled for a specific priority level.  
[More...](#)

int **logPrintf** (**uint16\_t** channelLevel, const char \*fmt,...)

sprintf() like formatted traces. [More...](#)

int **logPrintvf** (**uint16\_t** channelLevel, const char \*fmt, va\_list va)

A version of **logPrintf()** that accepts a va\_list as parameter.  
[More...](#)

`int logDump (uint16_t channelLevel, const void *buffer, int size)`

Write a block of data as a binary dump. [More...](#)

---

## Detailed Description

---

## Rationale

Tracing is one of the basic helper tools for debugging. Defining a common tracing API that can be used by all LAR libraries and modules makes easier for them to interact. Nevertheless tracing should be optional and highly customizable.

# Introduction

The `larlib.log` module provides a general logging/tracing system with assertions and other helper routines. It is designed to be easily ported to any platform.

Its mechanics are based on the concept of *tracing channels*: 256 channels are available, and each one can be separately configured to be disabled, or enabled with a given configuration. An application can use different channel IDs to different modules so to conditionally enable tracing of specific modules.

By default all channels are *disabled*.

## Note

Channel numbers 200 and higher are reserved for LAR's internal libraries and modules and should *not* be used by applications.

Given that tracing is very platform dependent, this library takes the position that all platform-dependency is moved to the *configuration* step, and after that the actual tracing code should not need to change.

# Setup

Each channel has a separated configuration, and each configuration includes, information about how to write, and the tracing level in use.

The *tracing level* of a given channel is a number that define the priority of the trace. On run-time only traces with a priority value *larger than or equal to* the configured priority of this channel are actually written.

For example, to have all tracing disabled, except for the channel number 42, which should only print errors or higher priority:

```
// disable all channels (this is the default)
logSetChannels(LOG_ALL_CHANNELS, 0, NULL, NULL,
NULL);
// enable only channel 42
logSetChannels(42, 1, LOG_ERROR, myWriteFunction,
myDumpFormatFunction, myContext);
```

For each channel a `write` function should be given, this function is unique for each platform and device, and a `format` function may also be given.

In platforms and devices where the `write` function expects binary data (Telium's `trace()` for instance) the `format` function may not be given, and all binary data written using `logDump()` will be directly sent to the `write` function.

In platforms where the `write` function only accepts ASCII chars (the printer for example) then the `format` function must be given and it should convert the binary data received by `logDump()` to a printable (ASCII-only) format. The provided function `logDumpFormattedAscii()` is an example of this formatting function, and will convert the binary data to a two-column hexadecimal/ascii format.

## Telium

The Telium platform has a built-in system for tracing, using the `trace()` function. Also, each trace may have an associated code (called its *SAP* code).

To use the built-in `trace()` function, use `logWriteTeliumTrace()` as `writeFunction` and `NULL` as `dumpFormatFunction` parameters to `logSetChannels()`.

```
logSetChannels(<firstChannel>, <channelCount>,
    <logLevel>, logWriteTeliumTrace, NULL, (void
*) <sapCode>);
```

# Tracing

After the configuration is done using `logSetChannels()`, one should use `logPrintf()`, `logPrintvf()` or `logDump()` to output traces:

```
// this trace has LOG_ERROR priority
logPrintf(42|LOG_ERROR, "Failed at %s with error
    %d", functionName, errorCode);
// But those only print if LOG_DEBUG level is
    active for channel 42:
logPrintf(42, "Result buffer:");
logDump(42, buffer, bufferSize);
```

`LOG_ASSERT()` uses channel `LOG_CH_BASE` to display the assertion failure message and resets the device. If channel `LOG_CH_BASE` is not enabled the message is not displayed, but the device will be reseted anyway. See `logReservedChannels_t`.

# Macro Definition Documentation

---

**#define LOG\_ASSERT( cond )**

---

Assertion using `larlib.log` for output.

If `cond` evaluates to zero, write an error message on channel **LOG\_CH\_BASE** and either **halt** execution or **reset** the device.

# Typedef Documentation

---

**typedef int(\* logDumpFunction\_t) ([logWriteFunction\\_t](#) write, void \*context, [uint16\\_t](#) channelLevel, const void \*data, int dataLen)**

---

Type of function called to format the output on [logDump\(\)](#).

It should call the received `write` function to actually output the formatted data.

## Parameters

<b>write</b>	The function to call to actually generate the trace
<b>context</b>	The same <code>context</code> variable that is passed as parameter to <a href="#">logSetChannels()</a>
<b>channelLevel</b>	Channel and level that triggered this trace
<b>data</b>	Pointer to binary data to write.
<b>dataLen</b>	Length of <code>data</code> , in bytes.

## Returns

Zero on success, non-zero on error.

**typedef int(\* logWriteFunction\_t) (void \*context, [uint16\\_t](#) channelLevel, const char \*msg, int msgLength)**

---

Type of the function that does the actual writing on a trace channel.

## Parameters

<b>context</b>	The same <code>context</code> variable that is passed as parameter to <a href="#">logSetChannels()</a>
<b>channelLevel</b>	Channel and level that triggered this trace
<b>msg</b>	Data to write.
<b>msgLength</b>	Length, in bytes, of <code>msg</code> .

## Returns

Zero on success, non-zero on error.

# Enumeration Type Documentation

---

## **enum logLevel\_t**

---

General tracing levels.

Applications may define intermediary values.

Enumerator	
LOG_DEBUG	Lowest-priority: only for debugging.
LOG_INFO	Informative.
LOG_WARNING	Run-time warnings.
LOG_ERROR	Run-time errors.
LOG_CRITICAL	Critical.
LOG_ALWAYS	Always write (if channel is enabled)

## **enum logReservedChannels\_t**

---

List of reserved log channels.

These channels are reserved by larlib and other system libraries.

Enumerator	
LOG_CH_BASE	<a href="#">Basic Definitions</a> and <a href="#">LOG_ASSERT()</a>
LOG_CH_CONV	<a href="#">Conversion routines</a>
LOG_CH_DATE	<a href="#">Date and time</a>
LOG_CH_FS	<a href="#">File System</a>
LOG_CH_INI	<a href="#">INI file parsing</a>
LOG_CH_MEM	<a href="#">Memory Management</a>

LOG_CH_TAB	Table Management
LOG_CH_STEP	State-machine
LOG_CH_FORMAT	Buffer formatting
LOG_CH_TASK	Multi-threading

# Function Documentation

---

## **int logChannelsEnabled ( uint16\_t channelLevel )**

---

Check if a log channel is enabled for a specific priority level.

Given a `channelLevel`, formatted exactly as would for [logPrintf\(\)](#) or [logDump\(\)](#), check if a write to this configuration would generate a trace.

### Parameters

**channelLevel** Defines both the channel and the level of this trace. The lower 8-bits define the channel to use, and the upper 8-bits define the level.

### Returns

Non-zero if a [logPrintf\(\)](#) or [logDump\(\)](#) to `channelLevel` would generate a call to the configured `writeFunction`.

Zero otherwise.

---

## **int logDump ( uint16\_t channelLevel,                  const void \* buffer,                  int            size                 )**

---

Write a block of data as a binary dump.

The output format should include both the readable part of the data pointed by `buffer` and the hexadecimal of its bytes.

### Parameters

**channelLevel** Channel and level of this trace (see [logPrintf\(\)](#))  
**buffer** Data block to dump.

**size** Number of bytes of buffer to print.

**Returns**

BASE\_ERR\_OK

**See also**

[logSetChannels](#)

```
int logDumpFormattedAscii ( logWriteFunction_t write,  
                           void * context,  
                           uint16_t channelLevel,  
                           const void * data,  
                           int dataLen  
                         )
```

---

dumpFormatFunction that formats the received buffer as a two-column hexadecimal and ASCII display and calls the received `write` function.

This function can be used as the `dumpFormatFunction` parameter of [logSetChannels\(\)](#) for channels where `write` is not ready to directly receive binary (non-ASCII) data.

**Parameters**

<b>write</b>	Function that will be used to actually write the formatted lines.
<b>context</b>	Ignored and passed as received to <code>write</code> .
<b>channelLevel</b>	Channel and level that triggered this trace
<b>data</b>	Binary data to format.
<b>dataLen</b>	Length of <code>data</code> in bytes.

**Returns**

The return of `write`.

```
int logPrintf ( uint16_t channelLevel,
```

```
    const char * fmt,  
    ...  
)
```

sprintf() like formatted traces.

This call formats and sends the message defined by `fmt` using the parameters associated with the channel and priorities defined in `channelLevel`.

The values in `LogLevel_t` are defined such that a simple OR'ing of the values with the channel number work.

For example, to write to channel 42 with the default priority:

```
1| logPrintf(42, "this is the message %d",  
1);
```

And to write to the same channel using the `LOG_CRITICAL` priority:

```
1| logPrintf(42|LOG_CRITICAL, "this is the  
message %d", 2);
```

## Parameters

**channelLevel** Defines both the channel and the level of this trace. The lower 8-bits define the channel to use, and the upper 8-bits define the level.

**fmt** String format (as `sprintf()`).

## Returns

`BASE_ERR_OK` even if `channel` is disabled or message not written by priority

An error code returned by the `writeFunction` defined for this channel

## See also

[logSetChannels](#)

```
int logPrintvf ( uint16_t      channelLevel,  
                  const char * fmt,  
                  va_list       va  
                )
```

---

A version of [logPrintf\(\)](#) that accepts a `va_list` as parameter.

This is quite useful for creating other abstractions on top of this lib.

## Parameters

`channelLevel` Channel and level of tracing for this trace (see [logPrintf\(\)](#)).

`fmt` String format (as `sprintf()`).

`va` `va_list` with the list of parameters to be formatted.

## Returns

`BASE_ERR_OK`

## See also

[logSetChannels](#)

```
int logSetChannels ( uint8_t      firstChannel,  
                     int          numChannels,  
                     uint16_t     level,  
                     logWriteFunction_t writeFunction,  
                     logDumpFunction_t dumpFormatFunction,  
                     void *       context  
                   )
```

---

Configures one or more tracing channels.

Configures the first `numChannels` starting at `firstChannel` to use the tracing priority `level` and write output using `writeFunction` and

`dumpFormatFunction`.

For example, to configure the channels in the range [5, 10]:

```
1| logSetChannels(5, 5, myLevel,  
myWriteFunction, NULL, myContext);
```

To configure all channels, use the macro `LOG_ALL_CHANNELS`:

```
1| logSetChannels(LOG_ALL_CHANNELS, myLevel,  
myWriteFunction, NULL, myContext);
```

## Parameters

<b>firstChannel</b>	Index of the first channel to configure.
<b>numChannels</b>	How many channels starting at <code>firstChannel</code> to configure. If zero no channels are configured!
<b>level</b>	Tracing level enabled for the selected channels. Only the most significant 8-bits are used, see the values in <a href="#">logLevel_t</a> .
<b>writeFunction</b>	A function that is called with the string to write. It is called only with a zero-terminated string of chars in the ASCII range. Using <code>NULL</code> disables this channel.
<b>dumpFormatFunction</b>	A function that is called to pre-process the binary buffer received by <a href="#">logDump()</a> . If <code>NULL</code> then <a href="#">logDump()</a> will directly call <code>writeFunction</code> with the received data buffer. If not <code>NULL</code> then this function should format the input buffer and call the received <code>write</code> function with the formatted output. See <a href="#">logDumpFormattedAscii()</a> .
<b>context</b>	This value is stored and passed unchanged to <code>writeFunction</code> , and can

be used to configure its behavior. See the documentation of each logWrite function for more information.

## Returns

BASE\_ERR\_OK

BASE\_ERR\_INVALID\_PARAMETER if the range defined by firstChannel and numChannels is invalid

## See also

[logWriteFunction\\_t](#)  
[logDumpFunction\\_t](#)  
[logPrintf](#)  
[logPrintvf](#)  
[logDump](#)  
[logDumpFormattedAscii](#)  
[logWriteTeliumTrace](#)

```
int logWriteTeliumRemoteDebugger( void * context,
                                  uint16_t channelLevel,
                                  const char * msg,
                                  int msgLength
                                )
```

A [logWriteFunction\\_t](#) that can be used as parameter to [logSetChannels\(\)](#) and writes traces to the Remote Debugger console on IngeDev.

### Attention

This function is Telium-specific and may not be defined on other platforms.

## Parameters

**context** Not used

**channelLevel** Channel and level that triggered this trace

**msg** Zero-terminated string to write.

**msgLength** Length of `msg` in bytes.

### Returns

Always return `BASE_ERR_OK`.

### See also

[logSetChannels](#)

```
int logWriteTeliumTrace ( void * context,
                           uint16_t channelLevel,
                           const char * msg,
                           int msgLength
                         )
```

A `logWriteFunction_t` that can be used as parameter to `logSetChannels()` and writes traces using Telium's `trace()` function.

Traces are written using the SAP code defined by the integer value of `context`. For example, to use SAP code `0xB000`, use:

```
1| logWriteTeliumTrace((void*) 0xB000,
                      myMessage, strlen(myMessage))
```

To configure the channel 42 to use this tracing function with SAP code `0x1234`, use:

```
1| logSetChannels(42, 1, LOG_DEBUG,
                  logWriteTeliumTrace, NULL, (void*) 0x1234);
```

### Attention

This function is Telium-specific and may not be defined on other platforms.

### Parameters

**context** The *value* of this pointer indicates which SAP code to use.

**channelLevel** Channel and level that triggered this trace

**msg** Zero-terminated string to write.

**msgLength** Length of `msg` in bytes.

### Returns

Always return `BASE_ERR_OK`.

### See also

[logSetChannels](#)

# LAR Library 1.14

Main Page	Related Pages	Modules	Data Structures	
Files				Functions
<h2>Memory Management</h2>				

## Functions

---

`void * memAlloc (int nb)`

Allocate a block of memory from the heap. [More...](#)

`void * memAllocZero (int nb)`

Allocate and zero a block of memory from the heap. [More...](#)

`void * memRealloc (void *oldp, int nb)`

Resize a block of memory, or allocate a new one. [More...](#)

`void memFree (void *p)`

Release a previously allocated block. [More...](#)

---

## Detailed Description

---

## Rationale

Memory management is important, specially in an embedded environment. Besides not all platforms correctly supporting ANSI C, having our own memory-allocation functions allow for easier debugging of applications.

# Introduction

This modules includes portable memory-management functions. All modules of `larlib` use this functions. They may allow improved debugging / tracing support than any given platform's built-in implementation.

# Function Documentation

---

## **void\* memAlloc ( int nb )**

---

Allocate a block of memory from the heap.

The memory allocated by this function should only be accessed by the calling application. The pointer is not guaranteed to be valid for other applications.

### Parameters

**nb** Number of bytes to allocate.

### Returns

NULL if memory could not be allocated.

A pointer to the newly allocated, uninitialized, block.

## **void\* memAllocZero ( int nb )**

---

Allocate and zero a block of memory from the heap.

This is equivalent to calling [memAlloc\(\)](#) followed by `memset()`.

### Parameters

**nb** Number of bytes to allocate.

### Returns

NULL if memory could not be allocated.

A pointer to the newly allocated, zero-initialized, block.

## **void memFree ( void \* p )**

---

Release a previously allocated block.

## Parameters

**p** A pointer returned by **memAlloc()**

```
void* memRealloc ( void * oldp,  
                   int      nb  
                 )
```

---

Resize a block of memory, or allocate a new one.

The data pointed by **oldp** is shallow-copied (using `memcpy()`) to the new pointer on success. This is equivalent to the ANSI C function `realloc()`.

## Parameters

**oldp** Address of the memory block to resize.  
**nb** Number of bytes to resize **oldp** to.

## Returns

`NULL` if **oldp** could not be resized and new memory could not be allocated.

A pointer to the newly allocated memory, its prefix is the same as **oldp**.

# LAR Library 1.14

Main Page	Related Pages	Modules	Data Structures
Files	Data Structures   Typedefs   Functions		
<b>State-machine</b>			

## Data Structures

---

struct **stepAction\_t**

Each action in the step machine. [More...](#)

---

## Typedefs

---

typedef struct **step\_t** **step\_t**

Opaque type of a handle to a step-machine.

typedef int(\*) **stepFunction\_t** (**step\_t** \*s, **stepAction\_t** \*a,  
void \*c)

Type signature of the function associated with  
each action on the step machine. [More...](#)

---

## Functions

---

`step_t * stepCreate (stepAction_t *actions, int nactions)`

Create a new step machine. [More...](#)

`int stepRun (step_t *s, void *c)`

Execute the step machine. [More...](#)

`int stepGotoLabel (step_t *s, int label)`

Change the next action to be executed. [More...](#)

`void stepDestroy (step_t *s)`

Release the resources associated with a step-machine.

[More...](#)

---

## Detailed Description

---

### Rationale

Most of the POS financial applications have the same overall structure: each transaction is broken into distinct "steps", each one should be executed in turn, and some steps may be shared between different transactions.

# Introduction

This module provides a framework to implement a sub-set of state-machines, here called **step-machines**. A step-machine is formed by a sequence of states (steps), which are executed in sequence, except explicit change of execution flow (such as a *goto*).

Each step is defined by the contents of a **stepAction\_t** structure. Each action has an associated `label`, a numeric identifier that is useful if this action must be target for a *goto*; The function that executes the actual behavior associated with this action; And a opaque parameter that is meant to be used by the associated function.

An action may have a `NULL` function, in this case it is considered that the function has returned **BASE\_ERR\_OK**. This may be useful to insert "dummy" actions with only a valid label field, meant as *goto* targets.

Unless a **stepGotoLabel()** call is made, execution follows the steps in the order that they are given to **stepCreate()**, this is considered the *default* execution sequence.

# Example

This example is a loop that prints all numbers in the range [0..9]. Even though this is an unrealistic use of the step-machine module, it demonstrates all its features.

```
enum {
    // label for 'print' step
    LABEL_PRINT = 1
};

// print the int pointed by c
static int print(step_t *s, stepAction_t *a, void
    *c)
{
    printf(a->param, * (int *) c);
    return BASE_ERR_OK;
}

// increment the value of the integer pointed by c
static int incr(step_t *s, stepAction_t *a, void
    *c)
{
    int *n = (int *) c;
    (*n)++;
    return BASE_ERR_OK;
}

// goto LABEL_PRINT if the integer pointed by 'c'
// is < 10
static int loop(step_t *s, stepAction_t *a, void
    *c)
{
    int *n = (int *) c;
    if (*n < 10) {
        // loop back to 'print' action
        return stepGotoLabel(s, LABEL_PRINT);
    }
}
```

```
    }
    else {
        // continue to next step in default sequence
        return BASE_ERR_OK;
    }
}

static int stepExample(void)
{
    stepAction_t actions[] = {
        LABEL_PRINT, &print, "The number is %d\n",
        0,           &incr,  NULL,
        0,           &loop,   NULL,
    };

    step_t *s = stepCreate(actions,
                           BASE_LENGTH(actions));
    if (s) {
        int n = 0;
        int err = stepRun(s, &n);
        stepDestroy(s);
        return err;
    }
    else {
        return BASE_ERR_RESOURCE_PB;
    }
}
```

# Typedef Documentation

---

**typedef int(\* stepFunction\_t) (step\_t \*s, stepAction\_t \*a, void \*c)**

---

Type signature of the function associated with each action on the step machine.

## Parameters

- s** Handle to controlling step machine.
- a** Pointer to action associated with this step.
- c** Overall context parameter passed as parameter to [stepRun\(\)](#).

## Returns

BASE\_ERR\_OK on success. Any other value will stop the execution of the step machine immediately and return this value to the caller of [stepRun\(\)](#).

## Function Documentation

---

```
step_t* stepCreate ( stepAction_t * actions,
                     int               nactions
                   )
```

---

Create a new step machine.

The list of actions defined by `actions` will be copied to an internal, dynamically allocated, array.

This makes safe to create a step from an array on the stack and returning this value:

```
1 step_t *CreateMyStep(void)
2 {
3     stepAction_t actions[] = {
4         1, GetAmount,      "###.###.##0,00",
5         0, CheckAmount,   NULL,
6         2, GetCardData,   "Passe ou insira o
7             cartao",
8         0, CheckCardData, NULL,
9         3, DoTransaction, "Pedindo
10            autorizacao",
11         0, CheckResult,   NULL,
12         4, TransFail,     "Transacao Falhou",
13         5, TransOk,       "Transacao foi OK"
14     }
15     return stepCreate(actions,
16                         BASE_LENGTH(actions));
17 }
```

### Parameters

**actions** List of actions associated with this step machine.

**nactions** Number of elements in actions.

### Returns

A pointer to a new step-machine or NULL on error.

### See also

[stepRun](#)

[stepDestroy](#)

---

## void stepDestroy ( [step\\_t](#) \* s )

---

Release the resources associated with a step-machine.

### Parameters

s Handle to step-machine.

---

## int stepGotoLabel ( [step\\_t](#) \* s,                     int       label                     )

---

Change the next action to be executed.

Note that this call only marks an indicator of "next action to run", it does not cause the flow to change immediately. Multiple calls will override each other, the last one prevailing.

If more than one action of the given step-machine have the same label, the first one (the one with the smallest index) is selected.

One common idiom for using this call is associated with the return keyword:

```
1| int myActionFunction(step\_t *s,  
2|   stepAction\_t *a, void *c)  
3| {  
4|   // ... processing ...
```

```
4 |     return stepGotoLabel(s,
5 |     LABEL_MY_OTHER_ACTION);
5 | }
```

This has the side-effect of aborting the execution if `LABEL_MY_OTHER_ACTION` label is not found in the step-machine `s`.

## Parameters

**s** Handle to step-machine.  
**label** Label of target action, see [stepAction\\_t::label](#).

## Returns

`BASE_ERR_OK` if next action to execute was changed.  
`BASE_ERR_DATA_NOT_FOUND` if `label` was not found.

## See also

[stepRun](#)

---

```
int stepRun ( step_t * s,
              void *   c
            )
```

Execute the step machine.

## Parameters

**s** Handle to step-machine to run.  
**c** Opaque value that is passed to the action functions.

## Returns

`BASE_ERR_OK` if the machine was executed to the end.  
`BASE_ERR_INVALID_HANDLE` if `s` is invalid.  
The return from the first step that didn't return `BASE_ERR_OK`.

## See also

[stepGotoLabel](#)

---

Generated on Mon Mar 27 2017 15:42:53 for LAR Library by [doxygen](#) 1.8.9.1

# LAR Library 1.14

Main Page	Related Pages	Modules	Data Structures
Files			
Data Structures	Data Fields	Data Fields	
Data Fields		Data Fields	
<h2>stepAction_t Struct Reference</h2> <p>State-machine</p>			

Each action in the step machine. [More...](#)

```
#include <step.h>
```

## Data Fields

---

int **label**

Label associated with this action.

**stepFunction\_t fun**

Function that executes this action.

void \* **param**

Place-holder for parameter to `fun`.

---

## Detailed Description

---

Each action in the step machine.

---

The documentation for this struct was generated from the following file:

- **step.h**
- 

Generated on Mon Mar 27 2017 15:42:53 for LAR Library by [doxygen](#) 1.8.9.1

# LAR Library 1.14

Main Page	Related Pages	Modules	Data Structures	
Files	TypeDefs   Enumerations   Functions			
<h2>Table Management</h2>				

## Typedefs

---

`typedef struct table_t table_t`

Opaque type of a table instance.

---

## Enumerations

---

enum **tabErrors\_t** { **TAB\_ERR\_DELETED** = 1 }

Return codes specific to this module. [More...](#)

---

## Functions

int **tabOpen** (const char \*fname, **table\_t** \*\*t)  
Open an existing table. [More...](#)

int **tabCreate** (const char \*fname, int recSize, **table\_t** \*\*t)  
Create a new table or reset an existing one. [More...](#)

void **tabClose** (**table\_t** \*t)  
Close a **table\_t** instance. [More...](#)

int **tabFlush** (**table\_t** \*t)  
Force an update of the table file. [More...](#)

int **tabRecSize** (**table\_t** \*t)  
Return the record size for a table. [More...](#)

int **tabGetStamp** (**table\_t** \*t, **uint32\_t** \*stamp)  
Return the stamp value of the table. [More...](#)

int **tabSetStamp** (**table\_t** \*t, **uint32\_t** stamp)  
Modify the stamp value of the table. [More...](#)

int **tabGet** (**table\_t** \*t, int recn, void \*recp)  
Read an entry from the table. [More...](#)

int **tabPut** (**table\_t** \*t, int recn, const void \*recp)  
Write or update an entry on the table. [More...](#)

int **tabAppend** (**table\_t** \*t, const void \*recp)  
Append an entry to the table. [More...](#)

int **tabSize** (**table\_t** \*t)  
Return the total number of entries on the table. [More...](#)

int **tabCount** (**table\_t** \*t)  
Return the number of non-deleted entries on the table. [More...](#)

int **tabDelete** (**table\_t** \*t, int recn)  
Mark an entry as deleted. [More...](#)

int **tabUnDelete** (**table\_t** \*t, int recn)  
Remove the deleted mark of an entry. [More...](#)

int **tabPack** (const char \*fname)  
Permanently remove from the table all entries marked as deleted. [More...](#)

int **tabGoFirst** (**table\_t** \*t)  
Move the current entry marker to the first (zero'th) element on the table. [More...](#)

int **tabGetNext** (**table\_t** \*t, void \*recp)  
Read the current entry and move to the next. [More...](#)

int **tabGetPrev** (**table\_t** \*t, void \*recp)  
Read the previous element. [More...](#)

int **tabGoLast** (**table\_t** \*t)  
Move to the last element on the table. [More...](#)

int **tabFindNext** (**table\_t** \*t, void \*recp, const void \*keyp,  
int(\*cmpFn)(const void \*recp, const void \*keyp))  
Search incrementally the table for an entry. [More...](#)

int **tabFind** (**table\_t** \*t, void \*recp, const void \*keyp, int(\*cmpFn)  
(const void \*recp, const void \*keyp))  
Search the table for an element. [More...](#)

## Detailed Description

---

### Rationale

Simple DBF-like tables are at the core of most applications (transactions done, supported AIDs/keys, etc). Providing a common API not only eases on development (no need to reinvent the wheel) but also makes easier to replace by a version that provides custom characteristics (caching, storage access) without needing to revise the caller application.

# Introduction

This module implements a sequential storage of fixed-size records, in other words a *table*. This table is indexed by the position of each *record*, or *entry*, inside the table, with zero being the first one.

Each entry also includes a flag indicating if it was marked for deletion or not. New entries are always written to the end of the file, even if deleted entries exist. The user must manually call **tabPack()** to permanently remove all deleted entries from the file. If **tabPack()** is never called the file will grow indefinitely.

The table header has a field called *stamp*, a 32-bit unsigned integer value that can be modified by the user to control table versioning and state (no functions on this module, except for **tabGetStamp()** and **tabSetStamp()**, change this value). Upon creation the table receives a stamp value of zero.

This module does no control of table integrity (checksum, for example), if an application needs this level of integrity control, it must do so itself.

Some meta-data of the table (as number of elements) is kept in a memory cache and are only written with **tabFlush()** is called. This is done to (greatly) improve performance of table operations, specially during repeated add. Note that **tabFlush()** is internally called by **tabClose()**. See **Caching** below.

# Multi-Threading

No synchronization is done on accesses to the same `table_t` instance.  
The caller must guarantee that the same instance is not accessed  
concurrently. Simultaneous accesses to separate instances are ok.

# Caching

An implementation is allowed to cache operations in memory before committing to storage. To guarantee that all changes have been written, the user must call [tabFlush\(\)](#).

The current version caches the contents of the table header. This is done to avoid re-writing the header when adding or deleting records and changing the stamp value, and causes significant performance improvements on Telium platform.

# Enumeration Type Documentation

---

## **enum tabErrors\_t**

---

Return codes specific to this module.

Enumerator	
TAB_ERR_DELETED	Signal that the target entry is marked as deleted.

# Function Documentation

---

```
int tabAppend (table_t * t,  
               const void * recip  
)
```

---

Append an entry to the table.

This call is equivalent to `tabPut(t, tabSize(t), recip)` and has the same performance characteristics.

## Parameters

`t` A valid `table_t` instance.  
`recip` Data to write to new entry.

## Returns

`BASE_ERR_OK` if record was read correctly.  
`BASE_ERR_INVALID_HANDLE` if `t` is invalid.  
`BASE_ERR_INVALID_PARAMETER` if `recip` is invalid.

## See also

[tabPut\(\)](#)

---

**void tabClose (table\_t \* t)**

---

Close a `table_t` instance.

## Parameters

`t` A valid `table_t` instance.

---

**int tabCount (table\_t \* t)**

---

Return the number of non-deleted entries on the table.

This operation is O(1) and very quick, requiring no storage access.

### Parameters

**t** A valid `table_t` instance.

### Returns

The number non-deleted entries on the file.

`BASE_ERR_INVALID_HANDLE` if **t** is invalid.

---

```
int tabCreate ( const char * fname,
                int          recSize,
                table_t **   t
              )
```

---

Create a new table or reset an existing one.

### Parameters

**fname** File name of the table to open.

**recSize** Size of each record on the table, in bytes. Must be a positive number. There is no other hard-coded restriction other than available memory and the limitations of the `int` type.

[out] **t** On success, store the handle to the open table.

### Returns

`BASE_ERR_OK` on success.

`BASE_ERR_DATA_NOT_FOUND` if `fname` was not found.

`BASE_ERR_ACCESS` if cannot access the file `fname`.

`BASE_ERR_INVALID_PARAMETER` if `fname`, `recSize` or `t` is invalid

---

```
int tabDelete ( table_t * t,
```

```
    int      recn  
)  
}
```

Mark an entry as deleted.

This does not effectively remove the entry from the file, only mark it as deleted. The entry can be read with [tabGet\(\)](#) and un-deleted with [tabUnDelete\(\)](#), but will not be returned as a result of any iteration or search function.

This operation is O(1) and requires at most two writes (one to the *deleted* flag of the record, another to the table header).

## Parameters

- t** A valid [table\\_t](#) instance.
- recn** Index of record to delete, in the range [0, `tabSize(t) - 1`].

## Returns

- `BASE_ERR_OK` if record was read correctly.
- `BASE_ERR_DATA_NOT_FOUND` if `recn` is an invalid index.
- `BASE_ERR_INVALID_HANDLE` if `t` is invalid.

## See also

[tabUnDelete\(\)](#) [tabGet\(\)](#)

```
int tabFind ( table_t *  
             void *  
             const void *  
             int(*)(const void *recp, const void *keyp) cmpFn  
           )
```

Search the table for an element.

This is equivalent to a call to [tabGoFirst\(\)](#) followed by a call to [tabFindNext\(\)](#).

This operation is O(n).

## Parameters

- t** A valid `table_t` instance.
- [out] recip** Contents of the entry found. Modified even if entry was not found.
- keyp** Pointer to the key passed as second parameter to `cmpFn`.
- cmpFn** Comparison function, see [tabFindNext\(\)](#).

## Returns

- The index of the record read ( $\geq 0$ ) on success.
- `BASE_ERR_INVALID_HANDLE` if `t` is invalid.
- `BASE_ERR_DATA_NOT_FOUND` if no matching record was found.
- `BASE_ERR_INVALID_PARAMETER` if any required parameter is NULL.

## See also

- [tabFindNext\(\)](#)
- [tabGoFirst\(\)](#)

```
int  
tabFindNext (table_t *  
             void *  
             const void *  
             int (*)(const void *recp, const void *keyp) cmpFn  
)
```

Search incrementally the table for an entry.

Return the index of the next element `recp`, starting at the current table position, for which `cmpFn(recp, keyp)` is zero.

This is equivalent to iterating through the table using [tabGetNext\(\)](#) until an entry that satisfies the `cmpFn()` condition is found.

This operation is O(n).

## Parameters

- t** A valid `table_t` instance.
- [out] recip** Contents of the entry found. Modified even if entry was not found.
- keyp** Pointer to the key passed as second parameter to `cmpFn`.
- cmpFn** Comparison function, see the description above.

## Returns

- The index of the record read ( $\geq 0$ ) on success.
- `BASE_ERR_INVALID_HANDLE` if `t` is invalid.
- `BASE_ERR_DATA_NOT_FOUND` if no matching record was found.
- `BASE_ERR_INVALID_PARAMETER` if any required parameter is NULL.

---

## `int tabFlush (table_t * t)`

Force an update of the table file.

The implementation is allowed to cache certain operations for faster throughput, this call forces all pending changes to be actually written to storage.

Because of this, there is no guarantee that any change to the table is actually reflected on storage before this call.

## Parameters

- t** A valid `table_t` instance.

## Returns

- `BASE_ERR_INVALID_HANDLE` if `t` is invalid.
- `BASE_ERR_RESOURCE_PB` if could not commit changes.

```
int tabGet (table_t * t,
            int      recn,
            void *   recip
        )
```

Read an entry from the table.

The code to read a record and detect if it is marked as deleted:

```
1 err = tabGet(table, index, &record);
2 if (err == BASE_ERR_OK) {
3     // record was read and *NOT* marked as
4     // deleted
5 } else if (err == TAB_ERR_DELETED) {
6     // record was read but *IS* marked as
7     // deleted
8 } else {
9     // failed: could not read record
10 }
```

This operation is O(1), requiring only one read operation.

## Parameters

**t** A valid **table\_t** instance.

**recn** Index of the entry to read, in the range [0, `tabSize(t)` - 1].

**[out] recip** Where to store the entry data. Must have space for at least `tabRecSize(t)` bytes.

## Returns

`BASE_ERR_OK` if record was read correctly.

`TAB_ERR_DELETED` if the record was read, but it is marked as deleted.

`BASE_ERR_INVALID_HANDLE` if **t** is invalid.

BASE\_ERR\_DATA\_NOT\_FOUND if `recn` is an invalid index.  
BASE\_ERR\_INVALID\_PARAMETER if `recp` is invalid.

---

```
int tabGetNext( table_t * t,  
                void *      recp  
            )
```

---

Read the current entry and move to the next.

This call skips all entries marked as deleted.

### Parameters

**t** A valid `table_t` instance.  
[out] **recp** Contents of the read entry. Not modified if no entry was read.

### Returns

The index of the record read ( $\geq 0$ ) on success.

BASE\_ERR\_DATA\_NOT\_FOUND on *EOF*.

---

```
int tabGetPrev( table_t * t,  
                void *      recp  
            )
```

---

Read the previous element.

This call skips all entries marked as deleted.

### Parameters

**t** A valid `table_t` instance.  
[out] **recp** Contents of the read entry. Not modified if no entry was read.

### Returns

The index of the record read ( $\geq 0$ ) on success.

BASE\_ERR\_DATA\_NOT\_FOUND on EOF.

```
int tabGetStamp ( table_t* t,
                  uint32_t* stamp
                )
```

---

Return the stamp value of the table.

This operation is very cheap, the stamp value is cached on memory and no file reads are actually performed.

### Parameters

**t** A valid `table_t` instance.  
[out] **stamp** Value of the stamp field of this table.

### Returns

BASE\_ERR\_OK if stamp was read correctly.  
BASE\_ERR\_INVALID\_HANDLE if t is invalid.  
BASE\_ERR\_INVALID\_PARAMETER if stamp is an invalid pointer.

---

**int tabGoFirst (table\_t \* t)**

---

Move the current entry marker to the first (zero'th) element on the table.

The generic code to iterate through an entire table, start to finish, is:

```
1 tabGoFirst(t);
2 while (tabGetNext(t, recip) == BASE_ERR_OK)
{
3     // use the value on recip
4 }
```

### Parameters

**t** A valid `table_t` instance.

### Returns

`BASE_ERR_OK` on success.  
`BASE_ERR_INVALID_HANDLE` if `t` is invalid.  
`BASE_ERR_DATA_NOT_FOUND` if `t` is empty.

### See also

[tabGetNext\(\)](#)

---

**int tabGoLast ( `table_t` \* `t` )**

Move to the last element on the table.

The generic code to iterate an entire table, end to start, is:

```
1 tabGoLast(t);
2 while (tabGetPrev(t, recp) == BASE_ERR_OK)
3 {
4     // use the value on recp
5 }
```

### Parameters

**t** A valid `table_t` instance.

### Returns

`BASE_ERR_OK` on success.  
`BASE_ERR_INVALID_HANDLE` if `t` is invalid.  
`BASE_ERR_DATA_NOT_FOUND` if `t` is empty.

### See also

[tabGetPrev\(\)](#)

---

**int tabOpen ( const char \* fname,**  
**`table_t` \*\*     t**

)

---

Open an existing table.

### Parameters

**fname** File name of the table to open.

[out] **t** On success, store the handle to the open table.

### Returns

BASE\_ERR\_OK on success.

BASE\_ERR\_DATA\_NOT\_FOUND if fname was not found.

BASE\_ERR\_ACCESS if cannot access the file fname.

BASE\_ERR\_INVALID\_PARAMETER if fname or t is invalid

---

## **int tabPack ( const char \* fname )**

---

Permanently remove from the table all entries marked as deleted.

A side-effect of this call is that the record index of all entries may change (all entries after a deleted entry will "move up").

This operation is O(n) on the number of records on the file. Each record of the file is read and copied to a new temporary table, requiring one read and two writes per record. The original file is then replaced by the temporary one.

### Parameters

**fname** Name of table file. Should not be currently open.

### Returns

BASE\_ERR\_OK on success.

BASE\_ERR\_DATA\_NOT\_FOUND if fname is not found.

BASE\_ERR\_RESOURCE\_PB on errors creating the temporary table or manipulating the file system.

```
int tabPut ( table_t *      t,
             int            recn,
             const void *   recip
           )
```

Write or update an entry on the table.

This operation is O(1), but requires two writes (one for the record itself, other for the table header).

Note that this call can, exceptionally, write an entry to one past the last existing entry of the table, which is equivalent to a call to [tabAppend\(\)](#).

Calling [tabPut\(\)](#) for a deleted entry does *not* change the deleted flag for it.

### Parameters

**t** A valid [table\\_t](#) instance.

**recn** Index of the entry to write, in the range [0, `tabSize(t)`].

**recp** Data to write to entry.

### Returns

`BASE_ERR_OK` if record was read correctly.

`BASE_ERR_INVALID_HANDLE` if **t** is invalid.

`BASE_ERR_DATA_NOT_FOUND` if **recn** is an invalid index.

`BASE_ERR_INVALID_PARAMETER` if **recp** is invalid.

---

**int tabRecSize ( table\_t \* t )**

Return the record size for a table.

### Parameters

**t** A valid [table\\_t](#) instance.

### Returns

The size of each entry on this table in bytes.  
BASE\_ERR\_INVALID\_HANDLE if t is invalid.

---

```
int tabSetStamp ( table_t * t,  
                  uint32_t stamp  
                )
```

Modify the stamp value of the table.

This operation does one write to the header of the table.

#### Parameters

**t** A valid `table_t` instance.  
**stamp** Value of the stamp field to set.

#### Returns

BASE\_ERR\_OK if stamp was read correctly.  
BASE\_ERR\_INVALID\_HANDLE if t is invalid.

---

```
int tabSize ( table_t * t )
```

Return the total number of entries on the table.

This operation is O(1) and very quick, requiring no storage access.

#### Parameters

**t** A valid `table_t` instance.

#### Returns

The total number of entries on the file, including the ones marked as deleted.  
BASE\_ERR\_INVALID\_HANDLE if t is invalid.

---

```
int tabUnDelete ( table_t * t,
```

```
    int      recn  
)  
}
```

Remove the deleted mark of an entry.

This operation is O(1) and requires at most two writes (one to the *deleted* flag of the record, another to the table header).

### Parameters

- t** A valid [table\\_t](#) instance.
- recn** Index of record to un-delete, in the range [0, `tabSize(t)` - 1].

### Returns

- `BASE_ERR_OK` if record was read correctly.
- `BASE_ERR_DATA_NOT_FOUND` if `recn` is an invalid index.
- `BASE_ERR_INVALID_HANDLE` if `t` is invalid.

### See also

[tabDelete\(\)](#) [tabGet\(\)](#)

# LAR Library 1.14

Main Page	Related Pages	Modules	Data Structures	
Files				
				Macros   Typedefs   Functions

## Multi-threading

## Macros

---

```
#define TASK_INFINITE_TIMEOUT (0xFFFFFFFFLU)
```

Use this constant to wait forever when a timeout is necessary. More...

---

## Typedefs

---

`typedef struct taskThread_t taskThread_t`

A handle to a running thread.

`typedef void(* taskThreadFunction_t) (void *param)`

Type of a function pointer to be used as entry point of a new thread. [More...](#)

`typedef struct taskQueue_t taskQueue_t`

Handle to a multi-threading queue.

`typedef struct taskSemaphore_t taskSemaphore_t`

Handle to a semaphore.

`typedef struct taskEvent_t taskEvent_t`

Handle to an event.

---

## Functions

`taskThread_t * taskThreadCreate (taskThreadFunction_t fn,  
void *param, uint32_t stackSize)`  
Create a new execution thread with entry point  
fn. [More...](#)

`void taskSleep (uint32_t timeout)`  
Sleep the current running thread. [More...](#)

`taskThread_t * taskThreadCurrent (void)`  
Return the handle of the current running  
thread. [More...](#)

`taskQueue_t * taskQueueCreate (int maxItems)`  
Create a new queue with up to `maxItems`  
allowed. [More...](#)

`void taskQueueDestroy (taskQueue_t *q)`  
Destroy a queue and discards all pending  
elements. [More...](#)

`int taskQueuePost (taskQueue_t *q, void *item)`  
Insert an element on the back of the queue q.  
[More...](#)

`int taskQueuePop (taskQueue_t *q, void **item,  
uint32_t timeout)`  
Extract the element from the front of the queue.  
[More...](#)

`taskSemaphore_t * taskSemaphoreCreate (int limit)`  
Create a semaphore with a maximum entry  
count of `limit`. [More...](#)

`void taskSemaphoreDestroy (taskSemaphore_t`

`*s)`

Destroy an existing semaphore. [More...](#)

int **taskSemaphoreAcquire** (**taskSemaphore\_t**  
`*s, uint32_t timeout)`

Acquire (also called *wait* or *down*) the  
semaphore. [More...](#)

int **taskSemaphoreRelease** (**taskSemaphore\_t**  
`*s)`

Release (also called *signal* or *up*) a previously  
acquired semaphore. [More...](#)

**taskEvent\_t** \* **taskEventCreate** (`void`)

Create a new event handler. [More...](#)

**taskEvent\_t** \* **taskEventCreateSystem** (`uint32_t` bitmask)

Creates an event handler to wait for system  
peripheral events. [More...](#)

void **taskEventDestroy** (**taskEvent\_t** \*`e`)

Release the resources associated with an  
event handler. [More...](#)

int **taskEventSignal** (**taskEvent\_t** \*`e`)

Set an event to the *signaled* state. [More...](#)

int **taskEventCheck** (**taskEvent\_t** \*`e`)

Check if an event is signaled, without waiting or  
clearing the signal. [More...](#)

int **taskEventWait** (**taskEvent\_t** \*`e`, `uint32_t`  
`timeout)`

Wait for `e` to change to signaled state. [More...](#)

int **taskEventWaitAny** (**taskEvent\_t** \*`events[]`, int  
`nevents, uint32_t timeout)`

Wait for at least one among a list of events to be signaled. [More...](#)

---

int **taskEventClear** (**taskEvent\_t** \*e)  
Set an event as *non-signaled*. [More...](#)

## Detailed Description

---

### Rationale

Multi-threading capabilities are necessary for cases such as background dialing, controlling of external devices and background processing. Unfortunately OS capabilities and APIs vary widely, so a common ground is necessary to ease porting.

# Introduction

This module deals with multi-threading, or the creation of simultaneous threads of execution, and synchronization of those threads. The following objects are given:

- *Threads*: an application may create parallel threads of execution;
- *Queues*: FIFO (first-in-first-out) sequences of values designed so one thread can communicate with other in an orderly manner;
- *Events*: signals between threads, that carry no other information other than the event has been triggered;
- *Semaphores*: restrict simultaneous access to resources.

# Threads

Each thread is a parallel point of execution that has access to the same global environment (global variables, APIs) as the application as a whole, but a local stack (for local variables).

Because all threads access the same set of global variables great care must be taken so two threads do not attempt to modify the same memory location at the same time.

Each thread has an associated amount of memory for its stack, the application may "suggest" a minimum size for this stack on the call to **taskThreadCreate()**, but the size of the stack is ultimately platform-dependent.

After a thread has finished running (i.e. the user provided function has returned) its **taskThread\_t** handle becomes invalid, it is the user responsibility to guarantee that the handle is not used in this case.

This module does not provide any form of thread-local storage, if such is required the application must implement this itself.

## Attention

Each platform may have an unspecified limit on the number of concurrent tasks that may be created at each time!

See [http://en.wikipedia.org/wiki/Thread\\_\(computer\\_science\)](http://en.wikipedia.org/wiki/Thread_(computer_science)) and [http://en.wikipedia.org/wiki/Thread\\_safety](http://en.wikipedia.org/wiki/Thread_safety)

# Queues

A queue is a variable-sized array of elements with first-in-first-out semantics. The two basic operations provided are **taskQueuePost()** which adds an element to the *front* of the queue, and **taskQueuePop()** which removes an element from the *back* of the queue.

This implementation is limited to queue elements of type `void*`, they can be used to directly encode integer values (small enough to fit on an address) or point themselves to the actual value.

Queues are the recommended method of intra-thread communication, but the application must be careful to enforce *move* semantics: i.e. when a value (specially in case of pointers) is sent from one thread to another using a queue, the sender thread is in effect relinquishing ownership of this element and passing to the receiver thread. Carefully applying this methodology has the potential of reducing errors caused by simultaneous access to memory locations (see reference below).

Note that, as in the **Double-ended queue** module, the queue does no special handling of the `void*` pointers stored in it, more specifically, it will not call **memFree()** on any pointer, the application must be careful to release any memory itself (specially when **taskQueueDestroy()** is called).

See [http://en.wikipedia.org/wiki/Message\\_passing](http://en.wikipedia.org/wiki/Message_passing)

# Events

Events signal the occurrence of something. They are a binary marker with the states *signaled* and *non-signaled*. When created, a `taskEvent_t` is in *non-signaled* state, and only a call to `taskEventSignal()` will change this.

Threads may wait until an event is in *signaled* state by calling `taskEventWait()` or `taskEventWaitAny()`. The difference is that the first is a simpler API when it is necessary to wait for only one event handle, while the second allows to wait for multiple event handles on the same call.

## Device Events

Both supported platforms (Unicapt32, Telium) use a bitmap (encoded in a 32-bit unsigned integer) to mark which events have occurred. This API support interface with those OS-provided event facilities by using [taskEventCreateSystem\(\)](#).

A [taskEvent\\_t](#) created this way works as any other event when used with [taskEventWait\(\)](#) or [taskEventWaitAny\(\)](#), but will not work when used with [taskEventSignal\(\)](#), only OS APIs will work.

Note that since *which* bits are used for which peripherals changes from platform to platform, this in itself is not a completely portable solution.

### Note

On Unicapt32 only device events are supported, user events generated by `psyEventSend()` and such are *not* supported, this is because the bitmap associated with each event defines the first parameter to `psyPeripheralResultWait()`, but do not handle the case of `PSY_EVENT_RECEIVED`.

# Semaphores

Semaphores limit the number of threads that can simultaneously enter a given region of code, called a *critical-region*. The most common case being of allowing only one thread to enter a critical region at a time.

This is useful to control access to resources where a queue or an event would not apply, for example access to a file.

For example, consider the (very) constrained case where two threads are repeatedly appending information to a same file using a function `appendToFile()` that receives as parameter the name of the file and a string message to append to it:

```
taskSemaphore_t *mutex;
static void init() {
    mutex = taskSemaphoreCreate(1); // only one
        thread at a time
}
static void thread1(void *p) {
for (;;) {
    taskSemaphoreAcquire(mutex,
        TASK_INFINITE_TIMEOUT);
    appendToFile("global.txt", "from thread 1");
    taskSemaphoreRelease(mutex);
}
}
static void thread2(void *p) {
for (;;) {
    taskSemaphoreAcquire(mutex,
        TASK_INFINITE_TIMEOUT);
    appendToFile("global.txt", "from thread 2");
    taskSemaphoreRelease(mutex);
}
}
void test(void) {
```

```
    init();
taskThreadCreate(&thread1, NULL, 0);
taskThreadCreate(&thread2, NULL, 0);
taskThreadSleep(TASK_INFINITE_TIMEOUT);
}
```

The semaphore `mutex` is used to forbid the two tasks from access the file "global.txt" at the same time. (Note that this convoluted example is un-optimal in many respects. If you find yourself writing this sort of architecture on a real application, please reconsider! A more scalable approach would be having an *append-to-file* thread that receives requests of strings to write to the global file from a `taskQueue_t`).

See [http://en.wikipedia.org/wiki/Semaphore\\_\(programming\)](http://en.wikipedia.org/wiki/Semaphore_(programming))

# Macro Definition Documentation

---

**#define TASK\_INFINITE\_TIMEOUT (0xFFFFFFFFFLU)**

---

Use this constant to wait forever when a timeout is necessary.

# Typedef Documentation

---

**typedef void(\* taskThreadFunction\_t) (void \*param)**

---

Type of a function pointer to be used as entry point of a new thread.

## Parameters

**param** The user-provided param as given to  
**taskThreadCreate()**

# Function Documentation

---

## `int taskEventCheck ( taskEvent_t * e )`

---

Check if an event is signaled, without waiting or clearing the signal.

### Attention

This does *not* work with events created by [taskEventCreateSystem\(\)](#).

### Parameters

**e** Handle to event

### Returns

BASE\_ERR\_OK if e is signaled.

BASE\_ERR\_TIMEOUT if e was not signaled.

BASE\_ERR\_INVALID\_HANDLE if e is invalid or was created by [taskEventCreateSystem\(\)](#).

BASE\_ERR\_RESOURCE\_PB in case of internal error.

## `int taskEventClear ( taskEvent_t * e )`

---

Set an event as *non-signaled*.

This call may be used to "clean the slate" before other operations, discarding previous signals.

### Attention

This does *not* work with events created by [taskEventCreateSystem\(\)](#).

### Parameters

**e** Handle to event.

## Returns

BASE\_ERR\_OK on success.  
BASE\_ERR\_INVALID\_HANDLE if e is invalid.  
BASE\_ERR\_RESOURCE\_PB in case of internal error

## `taskEvent_t* taskEventCreate ( void )`

---

Create a new event handler.

The event is created in *non-signaled* state. Multiple events can be created and the limit on number of created events is not related to the platform event handling primitives.

## Returns

An event handler or NULL on error.

## `taskEvent_t* taskEventCreateSystem ( uint32_t bitmask )`

---

Creates an event handler to wait for system peripheral events.

Both Unicapt32 and Telium platforms use a bitmap to check for device events (see `psyPeripheralResultWait()` and `ttestall()`). This call provides a bridge between `taskEvent_t` and system peripheral events.

Internally any call to `taskEventWait()` or `taskEventWaitAny()` with events created this way will call the OS functionality to wait for peripherals.

For platforms where this form of event handling does not exist, this call will return NULL.

Event handlers created in this way also need to be destroyed by calling `taskEventDestroy()`.

## Note

Signaling an event created this way (by `taskEventSignal()`) is

*not* supported. Those events can only be triggered by the OS, either internally or by calling OS APIs.

## Parameters

**bitmask** Which system event bits to wait for. The 31st bit (0x80000000) is reserved for internal use, trying to create an event for it will return NULL instead.

## Returns

A valid taskEvent\_t handle, or NULL if bitmask is invalid or peripheral events are not supported in this platform.

## `void taskEventDestroy ( taskEvent_t * e )`

---

Release the resources associated with an event handler.

What happens to threads blocked waiting for this event is undefined and platform-dependent.

## Parameters

**e** Handle to event.

## `int taskEventSignal ( taskEvent_t * e )`

---

Set an event to the *signaled* state.

If the event is already signaled, nothing changes.

### Attention

This does *not* work with events created by [taskEventCreateSystem\(\)](#).

## Parameters

**e** Handle to event.

## Returns

BASE\_ERR\_OK on success.

BASE\_ERR\_INVALID\_HANDLE if e is invalid or created with [taskEventCreateSystem\(\)](#).

---

```
int taskEventWait ( taskEvent_t * e,
                    uint32_t      timeout
)
```

Wait for e to change to signaled state.

If e is already signaled, will return immediately.

Non-system events are automatically changed back to *non-signaled* state. For system events this is platform-dependent.

### Parameters

e Handle to event.

timeout Max time to wait for e to signal, in hundreds of second.

### Returns

BASE\_ERR\_OK if e was signaled.

BASE\_ERR\_TIMEOUT if e was not signaled.

BASE\_ERR\_INVALID\_HANDLE if e is invalid.

BASE\_ERR\_RESOURCE\_PB in case of internal error

---

```
int taskEventWaitAny ( taskEvent_t * events[],
                      int            nevents,
                      uint32_t       timeout
)
```

Wait for at least one among a list of events to be signaled.

Calling this function with nevents == 1 is equivalent to a call to [taskEventWait\(\)](#).

Example:

```
1 // ... events e1 and e2 created somewhere
else ...
2 taskEvent_t *events[] = { e1, e2 };
3 int which = taskEventWaitAny(events, 2,
    TASK_INFINITE_TIMEOUT);
4 if (which == 0) event_e1_happened();
5 else if (which == 1) event_e2_happened();
6 else error_happened();
```

## Parameters

**events** Array of event handles to wait for.

**nevents** Number of elements in events.

**timeout** Max time to wait for an event to signal, in hundreds of second.

## Returns

The (zero-based) index (inside events) of the *first* event to be detected.

BASE\_ERR\_TIMEOUT if no event was signaled.

BASE\_ERR\_INVALID\_HANDLE if any element in events is invalid.

BASE\_ERR\_RESOURCE\_PB in case of internal error

## [\*\*taskQueue\\_t\\* taskQueueCreate \( int maxItems \)\*\*](#)

---

Create a new queue with up to `maxItems` allowed.

## Parameters

**maxItems** Max number of items that fit on this queue.

## Returns

Handle to new queue or NULL on error.

## `void taskQueueDestroy (taskQueue_t * q)`

---

Destroy a queue and discards all pending elements.

### Parameters

**q** Queue to be discarded.

### Note

The queue elements are discarded with no further processing, which may cause resource and/or memory leaks. If a `taskQueuePop()` is waiting the behavior is undefined and platform-dependent.

## `int taskQueuePop (taskQueue_t * q,                       void **           item,                       uint32_t        timeout                       )`

---

Extract the element from the front of the queue.

Example:

```
1 taskQueue_t *q = get_queue_handle();  
2 void *value;  
3 if (taskQueuePop(q, &value,  
                  TASK_INFINITE_TIMEOUT) == BASE_ERR_OK) {  
4     process((myType_t *) value);  
5 }
```

### Parameters

**q** Handle to queue.  
**[out] item** Where to store the element extracted from **q** (note the double indirection!).  
**timeout** Time to wait for element, in hundreds of seconds.

# Returns

BASE\_ERR\_OK on success  
BASE\_ERR\_INVALID\_HANDLE if `q` is invalid  
BASE\_ERR\_INVALID\_PARAMETER if `item` is NULL  
BASE\_ERR\_TIMEOUT if no element was received in `timeout`

Insert an element on the back of the queue q.

## Example:

```
1| taskQueue_t *q = taskQueueCreate(NITEMS);
2| taskQueuePost(q, (void *) 0x12345678); // hand-crafted pointer value
3| taskQueuePost(q, (void *) &someVariable); // pointer to existing variable
```

## Parameters

**q** Handle to queue.  
**item** Item to be inserted.

## Returns

BASE\_ERR\_OK on success  
BASE\_ERR\_INVALID\_HANDLE if q is invalid.  
BASE\_ERR\_OVERFLOW if q already has the max number of elements allowed.

Acquire (also called *wait* or *down*) the semaphore.

May block if the *s* access count is full, waiting for a slot to vacant.

### Parameters

**s** Handle to semaphore

**timeout** Max time to wait for a vacancy, in hundreds of seconds.

### Returns

BASE\_ERR\_OK on success (semaphore was acquired)

BASE\_ERR\_TIMEOUT if could no acquire semaphore in timeout

BASE\_ERR\_INVALID\_HANDLE if *s* is invalid

---

## **taskSemaphore\_t\* taskSemaphoreCreate ( int **limit** )**

Create a semaphore with a maximum entry count of *limit*.

### Parameters

**limit** Number of threads that may simultaneously acquire the returned semaphore.

### Returns

Handle to semaphore or NULL on error.

---

## **void taskSemaphoreDestroy ( taskSemaphore\_t \* **s** )**

Destroy an existing semaphore.

The effects of destroying a semaphore that is being waited on is undefined and platform-dependent.

### Parameters

**s** Handle to semaphore.

```
int taskSemaphoreRelease (taskSemaphore_t * s)
```

Release (also called *signal* or *up*) a previously acquired semaphore.

## Parameters

**s** Handle to semaphore.

## Returns

BASE ERR OK on success

BASE ERR INVALID HANDLE if s is invalid

**void taskSleep( uint32 t timeout )**

Sleep the current running thread.

This will cause the current thread to stop executing for `timeout` hundreds of second, but other threads will keep running.

## Parameters

**timeout** Time to sleep the current thread, in hundreds of seconds.

## taskThread t\*

## taskThreadCreate

```
( taskThreadFunction_t fn,  
    void *                param,  
    uint32_t               stackSize  
)
```

Create a new execution thread with entry point fn.

## Parameters

**fn** Entry point for thread execution.

**param** Parameter to be passed to `fn`.

**stackSize** A hint on the minimum stack size required for this thread. Depending on the platform, different limitations on minimum and maximum stack size will apply. Use 0 for a default value that is at least 2KiB in all supported platforms.

### Returns

Handle to newly created thread, or NULL on error.

---

## `taskThread_t* taskThreadCurrent( void )`

---

Return the handle of the current running thread.

### Returns

Handle to current thread.

# LAR Library 1.14

Main Page	Related Pages	Modules	Data Structures
Files			
Data Structures	Data Fields		

## Data Structures

Here are the data structures with brief descriptions:

<a href="#"> <b>dateDate_t</b></a>	A gregorian date
<a href="#"> <b>dateTime_t</b></a>	A gregorian date with time
<a href="#"> <b>format_t</b></a>	Overall parameters of a format session
<a href="#"> <b>formatField_t</b></a>	Information about each field to be read or written
<a href="#"> <b>iniElement_t</b></a>	Descriptor of elements allowed in an INI file
<a href="#"> <b>stepAction_t</b></a>	Each action in the step machine

# LAR Library 1.14

Main Page	Related Pages	Modules	Data Structures
Files			
Data Structures	Data Fields		
All	Variables		

Here is a list of all documented struct and union fields with links to the struct/union documentation for each field:

- body : [formatField\\_t](#)
- bodySize : [formatField\\_t](#)
- data : [formatField\\_t](#)
- date : [dateTime\\_t](#)
- day : [dateDate\\_t](#)
- errorCode : [format\\_t](#)
- errorField : [format\\_t](#)
- fieldMap : [format\\_t](#)
- fields : [format\\_t](#)
- filter : [format\\_t](#)
- fun : [stepAction\\_t](#)
- group : [iniElement\\_t](#)
- handler : [iniElement\\_t](#)
- hour : [dateTime\\_t](#)
- index : [formatField\\_t](#)
- key : [iniElement\\_t](#)
- label : [stepAction\\_t](#)
- min : [dateTime\\_t](#)
- mon : [dateDate\\_t](#)
- nfields : [format\\_t](#)
- param : [iniElement\\_t](#) , [stepAction\\_t](#)
- prefix : [formatField\\_t](#)
- sec : [dateTime\\_t](#)
- size : [formatField\\_t](#)
- suffix : [formatField\\_t](#)
- year : [dateDate\\_t](#)

Generated on Mon Mar 27 2017 15:42:53 for LAR Library by [doxygen](#) 1.8.9.1

# LAR Library 1.14

Main Page	Related Pages	Modules	Data Structures
Files			
	Data Structures	Data Fields	
All	Variables		

- body : `formatField_t`
- bodySize : `formatField_t`
- data : `formatField_t`
- date : `dateTime_t`
- day : `dateDate_t`
- errorCode : `format_t`
- errorField : `format_t`
- fieldMap : `format_t`
- fields : `format_t`
- filter : `format_t`
- fun : `stepAction_t`
- group : `iniElement_t`
- handler : `iniElement_t`
- hour : `dateTime_t`
- index : `formatField_t`
- key : `iniElement_t`
- label : `stepAction_t`
- min : `dateTime_t`
- mon : `dateDate_t`
- nfields : `format_t`
- param : `iniElement_t , stepAction_t`
- prefix : `formatField_t`
- sec : `dateTime_t`
- size : `formatField_t`
- suffix : `formatField_t`
- year : `dateDate_t`

# LAR Library 1.14

Main Page	Related Pages	Modules	Data Structures
Files			
File List	Globals		

## File List

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

 <a href="#">all.h</a>	Force inclusion of all LarLib modules
 <a href="#">base.h</a>	Larlib basic definitions
 <a href="#">bits.h</a>	Bit manipulation
 <a href="#">conv.h</a>	Conversion routines
 <a href="#">date.h</a>	Date and time
 <a href="#">deque.h</a>	A double-ended queue of void* elements
 <a href="#">format.h</a>	Buffer formatting
 <a href="#">fs.h</a>	File System abstraction
 <a href="#">ini.h</a>	Read/parsing of MS-like INI files
 <a href="#">log.h</a>	General Logging System
 <a href="#">mem.h</a>	Memory Management
 <a href="#">step.h</a>	State-machine for transaction processing
 <a href="#">tab.h</a>	Table management
 <a href="#">task.h</a>	Multi-threading

# LAR Library 1.14

Main Page	Related Pages	Modules	Data Structures
<b>Files</b>			
File List			Globals
C: \ Projects \ lalarlib \ larlib \ common \ inc \ larlib \			

## all.h File Reference

Force inclusion of all LarLib modules. [More...](#)

```
#include <lарlib/base.h> #include <lарlib/bits.h>
#include <lарlib/conv.h>
#include <lарlib/date.h>
#include <lарlib/deque.h>
#include <lарlib/format.h>
#include <lарlib/fs.h>
#include <lарlib/ini.h>
#include <lарlib/log.h>
#include <lарlib/mem.h>
#include <lарlib/step.h>
#include <lарlib/tab.h>
#include <lарlib/task.h>
```

[Go to the source code of this file.](#)

## Detailed Description

---

Force inclusion of all LarLib modules.

---

Generated on Mon Mar 27 2017 15:42:52 for LAR Library by [doxygen](#) 1.8.9.1

# LAR Library 1.14

Main Page	Related Pages	Modules	Data Structures
<b>Files</b>			
File List	Globals		

C: > Projects > lalarlib > larlib > common > inc > larlib >

Macros | Typedefs | Enumerations | Functions

## base.h File Reference

Larlib basic definitions. [More...](#)

[Go to the source code of this file.](#)

## Macros

---

#define **BASE\_MAX**(a, b)

Return the largest of two elements (as defined by the comparison <).

#define **BASE\_MIN**(a, b)

Return the smallest of two elements (as defined by the comparison <).

#define **BASE\_LENGTH**(v)

Return the count of a fixed-size array. [More...](#)

#define **LARLIB\_VERSION** (0x0113)

Version of this LarLib as a 2-byte BCD number. [More...](#)

---

## Typedefs

typedef char **int8\_t**

Signed 8-bit integer. [More...](#)

typedef short **int16\_t**

Signed 16-bit integer. [More...](#)

typedef int **int32\_t**

Signed 32-bit integer. [More...](#)

typedef long long **int64\_t**

Signed 64-bit integer. [More...](#)

typedef unsigned char **uint8\_t**

Unsigned 8-bit integer. [More...](#)

typedef unsigned short **uint16\_t**

Unsigned 16-bit integer. [More...](#)

typedef unsigned int **uint32\_t**

Unsigned 32-bit integer. [More...](#)

typedef unsigned long long **uint64\_t**

Unsigned 64-bit integer. [More...](#)

## Enumerations

```
enum baseErrors_t {
    BASE_ERR_OK = 0, BASE_ERR_INVALID_PARAMETER
    = -1, BASE_ERR_DATA_NOT_FOUND = -2,
    BASE_ERR_INVALID_HANDLE = -3,
    BASE_ERR_RESOURCE_PB = -4,
    BASE_ERR_OVERFLOW = -5, BASE_ERR_ACCESS = -6,
    BASE_ERR_CANCEL = -7,
    BASE_ERR_TIMEOUT = -8
}
```

Common error values. [More...](#)

---

## Functions

---

const char \* **larlibGetVersion** (void)

Return a string with the version and revision of LarLib library. More...

---

# Detailed Description

---

Larlib basic definitions.

---

Generated on Mon Mar 27 2017 15:42:52 for LAR Library by [doxygen](#) 1.8.9.1

# LAR Library 1.14

Main Page	Related Pages	Modules	Data Structures
<b>Files</b>			
File List	Globals		

C: > Projects > lalarlib > larlib > common > inc > larlib >

[Typedefs](#) | [Enumerations](#) | [Functions](#)

## bits.h File Reference

Bit manipulation. [More...](#)

#include <[larlib/base.h](#)>

[Go to the source code of this file.](#)

## Typedefs

---

**typedef uint8\_t bitsBuffer\_t**

A bit-buffer is a sequence of 8-bits unsigned integers. More...

---

## Enumerations

---

```
enum bitsBltOp_t {
    BITS_BLT_COPY, BITS_BLT_NOT, BITS_BLT_AND,
    BITS_BLT_OR,
    BITS_BLT_XOR, BITS_BLT_NOT_AND
}
```

Possible bit operations for **bitsBlt()**. More...

---

## Functions

int **bitsSet** (**bitsBuffer\_t** \*b, int index, int value)  
Set or clear a bit in b. [More...](#)

int **bitsSetRange** (**bitsBuffer\_t** \*b, int offset, int count, int value)  
Set a range of bits on a buffer. [More...](#)

int **bitsGet** (const **bitsBuffer\_t** \*b, int index)  
Return the value of a bit in b. [More...](#)

**int64\_t** **bitsExtractInt** (const **bitsBuffer\_t** \*b, int offset, int count)  
Extract a sub-sequence of a bit buffer as an integer. [More...](#)

int **bitsAppend** (const **bitsBuffer\_t** \*src, int srcOffset, int srcCount, **bitsBuffer\_t** \*dst, int dstCount)  
Append one bit buffer to another. [More...](#)

int **bitsAppendInt** (**uint64\_t** bits, int nbits, **bitsBuffer\_t** \*dst, int dstCount)  
Append the bits extracted from an integer to a bit buffer.  
[More...](#)

int **bitsRotateLeft** (**bitsBuffer\_t** \*b, int index, int nbits, int nrotate)  
Rotate b left by nrotate bits. [More...](#)

int **bitsRotateRight** (**bitsBuffer\_t** \*b, int index, int nbits, int nrotate)  
Rotate b right by nrotate bits. [More...](#)

int **bitsShiftLeft** (**bitsBuffer\_t** \*b, int index, int nbits, int nshift)  
Shift b left by nshift bits. [More...](#)

int **bitsShiftRight** (**bitsBuffer\_t** \*b, int index, int nbits, int nshift)  
Shift b right by nshift bits. [More...](#)

int **bitsCopy** (const **bitsBuffer\_t** \*src, int srcOffset, int srcCount,  
**bitsBuffer\_t** \*dst, int dstOffset)  
Copy a sequence of bits from one buffer to another. [More...](#)

int **bitsBlt** (const **bitsBuffer\_t** \*src, int srcOffset, int srcCount,  
enum **bitsBltOp\_t** op, **bitsBuffer\_t** \*dst, int dstOffset)  
Combine two bit buffers using a bit operation. [More...](#)

---

## Detailed Description

---

Bit manipulation.

---

Generated on Mon Mar 27 2017 15:42:52 for LAR Library by [doxygen](#) 1.8.9.1

# LAR Library 1.14

Main Page	Related Pages	Modules	Data Structures
<b>Files</b>			
File List	Globals		

C: › Projects › lalarlib › larlib › common › inc › larlib ›

[Enumerations](#) | [Functions](#)

## conv.h File Reference

Conversion routines. More...

#include <[larlib/base.h](#)>

[Go to the source code of this file.](#)

## Enumerations

---

enum **convErrors\_t** { **CONV\_ERR\_INVALID\_BASE** = -101 }

Error codes specific to this module. [More...](#)

---

## Functions

int **convTxtToInt** (const char \*txt, int cnt, int base, **uint64\_t** \*n)  
Convert Text to Integer. [More...](#)

int **convTxtToInt8** (const char \*txt, int cnt, int base, **uint8\_t** \*n)  
Convert Text to an **uint8\_t**. [More...](#)

int **convTxtToInt16** (const char \*txt, int cnt, int base, **uint16\_t** \*n)  
Convert Text to an **uint16\_t**. [More...](#)

int **convTxtToInt32** (const char \*txt, int cnt, int base, **uint32\_t** \*n)  
Convert Text to an **uint32\_t**. [More...](#)

int **convTxtToInt64** (const char \*txt, int cnt, int base, **uint64\_t** \*n)  
This function is a synonym to **convTxtToInt()**. [More...](#)

int **convIntToTxt** (**uint64\_t** n, char \*txt, int maxcnt, int base)  
Convert Integer to Text. [More...](#)

int **convIntToTxtPad** (**uint64\_t** n, char \*txt, int cnt, int base)  
Convert Integer to Text padding left with zeroes to fill cnt chars. [More...](#)

int **convBufToHex** (const **uint8\_t** \*buf, int nbuf, char \*hex, int maxcnt)  
Convert buffer to hexadecimal string. [More...](#)

int **convHexToBuf** (const char \*hex, int cnt, **uint8\_t** \*buf, int maxbuf)  
Convert hexadecimal string to buffer. [More...](#)

int **convIntToBcd** (**uint64\_t** n, **uint8\_t** \*bcd, int maxcnt, int padRight)  
Convert Integer to BCD. [More...](#)

int **convIntToBcdPad** (**uint64\_t** n, **uint8\_t** \*bcd, int nibbles)  
Convert Integer to BCD with left padding. [More...](#)

int **convBcdToInt** (const **uint8\_t** \*bcd, int cnt, **uint64\_t** \*n)  
Convert BCD to Integer. [More...](#)

int **convTxtToBcd** (const char \*txt, int maxtxt, **uint8\_t** \*bcd, int maxbcd)  
Convert a string of decimal digits (or ISO PAN) to BCD.  
[More...](#)

int **convBcdToTxt** (const **uint8\_t** \*bcd, int maxbcd, char \*txt, int maxtxt)  
Convert a sequence of digits in BCD format to a decimal string. [More...](#)

int **convIntToBuf** (**uint64\_t** n, **uint8\_t** \*buf, int maxbuf)  
Convert an integer to a sequence of bytes in *big-endian* ordering. [More...](#)

int **convIntToBufPad** (**uint64\_t** n, **uint8\_t** \*buf, int maxbuf)  
Convert an integer to a sequence of bytes in *big-endian* ordering and with padding. [More...](#)

**uint64\_t** **convBufToInt** (const **uint8\_t** \*buf, int nbuf)  
Convert a sequence of bytes in *big-endian* ordering to an integer. [More...](#)

int **convBase64ToBuf** (const char \*b64, int cnt, **uint8\_t** \*buf, int maxbuf)  
Decode a base-64 encoded string into a sequence of bytes. [More...](#)

```
int convBufToBase64 (const uint8_t *buf, int nbuf, char *b64,  
int maxb64)  
Encode a sequence of bytes into a Base-64 string. More...
```

---

# Detailed Description

---

## Conversion routines.

---

Generated on Mon Mar 27 2017 15:42:52 for LAR Library by [doxygen](#) 1.8.9.1

# LAR Library 1.14

Main Page	Related Pages	Modules	Data Structures
<b>Files</b>			
File List	Globals		

C: > Projects > lalarlib > larlib > common > inc > larlib >

[Data Structures](#) | [Enumerations](#) | [Functions](#)

## date.h File Reference

Date and time. More...

#include <[larlib/base.h](#)>

[Go to the source code of this file.](#)

## Data Structures

---

struct **dateDate\_t**

A gregorian date. [More...](#)

struct **dateTime\_t**

A gregorian date with time. [More...](#)

---

## Enumerations

---

```
enum dateMonths_t {  
    DATE_JAN = 1, DATE_FEB, DATE_MAR, DATE_APR,  
    DATE_MAY, DATE_JUN, DATE_JUL, DATE_AUG,  
    DATE_SEP, DATE_OCT, DATE_NOV, DATE_DEC  
}
```

Enumeration of months. [More...](#)

```
enum dateWeekdays_t {  
    DATE_SUN = 0, DATE_MON, DATE_TUE, DATE_WED,  
    DATE_THU, DATE_FRI, DATE_SAT  
}
```

Enumeration of days of a week. [More...](#)

---

## Functions

int **dateGet (dateDate\_t \*d)**

Return the current system date. [More...](#)

int **dateTimeGet (dateTime\_t \*dt)**

Return the current system date and time. [More...](#)

**uint64\_t dateTimeGetMs (void)**

Return an increasing count of milliseconds. [More...](#)

int **dateSet (const dateDate\_t \*d)**

Update the system date. [More...](#)

int **dateTimeSet (const dateTime\_t \*dt)**

Update the system date and time. [More...](#)

int **dateToJulianDay (const dateDate\_t \*d)**

Return the Julian Day Number of d. [More...](#)

int **dateFromJulianDay (int jdn, dateDate\_t \*d)**

Convert the Julian Day Number to the Gregorian calendar.

[More...](#)

int **dateWeekday (const dateDate\_t \*d)**

Return the weekday of a given date. [More...](#)

int **dateIsLeap (uint16\_t year)**

Return non-zero if given year number is a leap year.

[More...](#)

int **dateIsValid (const dateDate\_t \*d)**

Checks if a date is valid. [More...](#)

int **dateTimeIsValid (const dateTime\_t \*dt)**

Checks if both time and date are valid. [More...](#)

int **dateDiff** (const **dateDate\_t** \*a, const **dateDate\_t** \*b)  
Return the difference, in days, between two dates. [More...](#)

int **dateAddDays** (**dateDate\_t** \*d, int ndays)  
Add a number of days to a given date. [More...](#)

int **dateCompare** (const **dateDate\_t** \*a, const **dateDate\_t** \*b)  
Compare two dates. [More...](#)

int **dateTimeAddSeconds** (**dateTime\_t** \*dt, int nsecs)  
Add seconds to a given date and time. [More...](#)

int **dateTimeCompare** (const **dateTime\_t** \*a, const **dateTime\_t** \*b)  
Compare two date times. [More...](#)

int **dateTimeDiff** (const **dateTime\_t** \*a, const **dateTime\_t** \*b)  
Return the difference, in seconds, between two date times.  
[More...](#)

---

## Detailed Description

---

Date and time.

---

Generated on Mon Mar 27 2017 15:42:52 for LAR Library by [doxygen](#) 1.8.9.1

# LAR Library 1.14

Main Page	Related Pages	Modules	Data Structures
<b>Files</b>			
File List	Globals		

C: > Projects > lalarlib > larlib > common > inc > larlib >

[Typedefs](#) | [Functions](#)

## deque.h File Reference

A double-ended queue of void\* elements. [More...](#)

#include <[larlib/base.h](#)>

[Go to the source code of this file.](#)

## TypeDefs

---

`typedef struct deque_t deque_t`

Opaque type for a double-ended queue.  
[More...](#)

---

## Functions

**deque\_t \* dequeCreate (int sizeHint)**

Create a new deque with space reserved for at least sizeHint elements. [More...](#)

**void dequeDestroy (deque\_t \*d)**

Destroy a deque created with **dequeCreate()**. [More...](#)

**int dequePushBack (deque\_t \*d, void \*p)**

Add an element to the "back" of a deque. [More...](#)

**int dequePopBack (deque\_t \*d)**

Remove the last element of a queue. [More...](#)

**int dequePushFront (deque\_t \*d, void \*p)**

Add an element on the "back" of a deque. [More...](#)

**int dequePopFront (deque\_t \*d)**

Remove the last element of a queue. [More...](#)

**int dequeLength (const deque\_t \*d)**

Return the number of elements inside a deque. [More...](#)

**void \* dequeGet (const deque\_t \*d, int i)**

Return the i'th element on a queue. [More...](#)

**void \* dequeGetFront (const deque\_t \*d)**

Return the first element on a deque. [More...](#)

**void \* dequeGetBack (const deque\_t \*d)**

Return the last element on a deque. [More...](#)

**void \* dequeSet (deque\_t \*d, int i, void \*p)**

Change the value of the i'th element on a deque. [More...](#)

int **dequeInsert** (**deque\_t** \*d, int i, void \*p)  
Insert an element at any position inside the deque.  
[More...](#)

---

int **dequeRemove** (**deque\_t** \*d, int i)  
Remove an element at any position inside the deque.  
[More...](#)

---

## Detailed Description

---

A double-ended queue of `void*` elements.

---

Generated on Mon Mar 27 2017 15:42:52 for LAR Library by [doxygen](#) 1.8.9.1

# LAR Library 1.14

Main Page	Related Pages	Modules	Data Structures
<b>Files</b>			
File List	Globals		
C: > Projects > lalarlib > larlib > common > inc > larlib >	Data Structures   Enumerations   Functions		

## format.h File Reference

Buffer formatting. [More...](#)

#include <[lalarlib/base.h](#)>

[Go to the source code of this file.](#)

## Data Structures

---

struct **format\_t**

Overall parameters of a format session. [More...](#)

struct **formatField\_t**

Information about each field to be read or written. [More...](#)

---

## Enumerations

---

enum **formatDirection\_t** { **FORMAT\_PACK**, **FORMAT\_UNPACK** }

Used as parameter to the formatting functions to inform if they should handle packing or unpacking. [More...](#)

---

## Functions

int **formatPack** (**format\_t** \*fmt, **uint8\_t** \*output, int maxOutput)

Write to output the packed values as described in fmt. [More...](#)

int **formatUnpack** (**format\_t** \*fmt, **uint8\_t** \*input, int inputSize)

Read from input and unpack the fields to fmt. [More...](#)

int **formatFilterISO8583** (**const format\_t** \*fmt, **const formatField\_t** \*field)

Use this function as the filter function for a ISO8583 buffer.

[More...](#)

int **formatBodyISO8583** (**uint8\_t** \*buffer, int limit, **formatDirection\_t** direction, **format\_t** \*fmt, **formatField\_t** \*field)

Read/write the bitmap of a ISO8583 message. [More...](#)

int **formatBodyPacked** (**uint8\_t** \*buffer, int limit, **formatDirection\_t** direction, **format\_t** \*fmt, **formatField\_t** \*field)

Format a field as a recursive call to **formatPack()** and **formatUnpack()**. [More...](#)

int **formatBodySkip** (**uint8\_t** \*buffer, int limit, **formatDirection\_t** direction, **format\_t** \*fmt, **formatField\_t** \*field)

Skip field->size bytes from message. [More...](#)

int **formatBodyMemcpy** (**uint8\_t** \*buffer, int limit, **formatDirection\_t** direction, **format\_t** \*fmt, **formatField\_t** \*field)

Copy field->size bytes of field->data to/from buffer. [More...](#)

int **formatBodyStrcpy** (**uint8\_t** \*buffer, int limit, **formatDirection\_t** direction, **format\_t** \*fmt, **formatField\_t** \*field)

Formatting for variable-sized zero-terminated strings. [More...](#)

int **formatBodyUint64Dec** (**uint8\_t** \*buffer, int limit,

**formatDirection\_t** direction, **format\_t** \*fmt, **formatField\_t** \*field)  
Format an `uint64_t` field as a decimal string. [More...](#)

int **formatBodyUint32Dec** (`uint8_t` \*buffer, int limit,  
**formatDirection\_t** direction, **format\_t** \*fmt, **formatField\_t** \*field)  
Format an `uint32_t` field as a decimal string. [More...](#)

int **formatBodyUint16Dec** (`uint8_t` \*buffer, int limit,  
**formatDirection\_t** direction, **format\_t** \*fmt, **formatField\_t** \*field)  
Format an `uint16_t` field as a decimal string. [More...](#)

int **formatBodyUint8Dec** (`uint8_t` \*buffer, int limit,  
**formatDirection\_t** direction, **format\_t** \*fmt, **formatField\_t** \*field)  
Format an `uint8_t` field as a decimal string. [More...](#)

int **formatBodyUint64Buf** (`uint8_t` \*buffer, int limit,  
**formatDirection\_t** direction, **format\_t** \*fmt, **formatField\_t** \*field)  
Format an `uint64_t` field as a sequence of bytes. [More...](#)

int **formatBodyBufHex** (`uint8_t` \*buffer, int limit, **formatDirection\_t**  
direction, **format\_t** \*fmt, **formatField\_t** \*field)  
Format an `buffer` field as a hexadecimal string. [More...](#)

int **formatSizeAsc2** (`uint8_t` \*buffer, int limit, **formatDirection\_t**  
direction, **format\_t** \*fmt, **formatField\_t** \*field)  
Read or write the value of `bodySize` as a 2-digit base-10 string.  
[More...](#)

int **formatSizeAsc3** (`uint8_t` \*buffer, int limit, **formatDirection\_t**  
direction, **format\_t** \*fmt, **formatField\_t** \*field)  
Read or write the value of `bodySize` as a 3-digit base-10 string.  
[More...](#)

int **formatSizeBcd2** (`uint8_t` \*buffer, int limit, **formatDirection\_t**  
direction, **format\_t** \*fmt, **formatField\_t** \*field)  
Read or write the value of `bodySize` as a 2-digit BCD number (1

byte). More...

int **formatSizeBcd3** (**uint8\_t** \*buffer, int limit, **formatDirection\_t** direction, **format\_t** \*fmt, **formatField\_t** \*field)

Read or write the value of bodySize as a 3-digit BCD number (2 bytes). More...

int **formatPadSpaces** (**uint8\_t** \*buffer, int limit, **formatDirection\_t** direction, **format\_t** \*fmt, **formatField\_t** \*field)

Write or skip field->size - field->bodySize spaces. More...

int **formatPadZeroes** (**uint8\_t** \*buffer, int limit, **formatDirection\_t** direction, **format\_t** \*fmt, **formatField\_t** \*field)

Write or skip field->size - field->bodySize ASCII '0' chars.

More...

int **formatPadZeroBytes** (**uint8\_t** \*buffer, int limit, **formatDirection\_t** direction, **format\_t** \*fmt, **formatField\_t** \*field)

Write or skip field->size - field->bodySize 0x00 bytes. More...

int **formatPadSkip** (**uint8\_t** \*buffer, int limit, **formatDirection\_t** direction, **format\_t** \*fmt, **formatField\_t** \*field)

Skip exactly field->size - field->bodySize bytes. More...

## Detailed Description

---

Buffer formatting.

---

Generated on Mon Mar 27 2017 15:42:52 for LAR Library by [doxygen](#) 1.8.9.1

# LAR Library 1.14

Main Page	Related Pages	Modules	Data Structures
<b>Files</b>			
File List	Globals		

C: › Projects › lalarlib › larlib › common › inc › larlib ›

[Typedefs](#) | [Enumerations](#) | [Functions](#)

## fs.h File Reference

File System abstraction. [More...](#)

#include <[larlib/base.h](#)>

[Go to the source code of this file.](#)

## Typedefs

---

```
typedef struct fsFile_t fsFile_t  
    Opaque type of a file.
```

---

## Enumerations

---

```
enum fsOpenFlags_t {  
    FS_OPEN_READ = 1, FS_OPEN_WRITE = 2,  
    FS_OPEN_CREATE = 4, FS_OPEN_RESET = 8,  
    FS_OPEN_APPEND = 16  
}  
Flags for fsOpen(). More...
```

```
enum fsSeekWhence_t { FS_WHENCE_SET, FS_WHENCE_END,  
    FS_WHENCE_CUR }  
Possible seek positions, see fsSeek(). More...
```

---

## Functions

int **fsSetFileNamePrefix** (const char \*prefix)

Set the common file name prefix. [More...](#)

int **fsPushFileNamePrefix** (const char \*prefix)

Push a new item on the file name prefix stack. [More...](#)

int **fsPopFileNamePrefix** (void)

Remove the top item on the file name prefix stack.

[More...](#)

const char \* **fsGetFileNamePrefix** (void)

Return the current prefix being used. [More...](#)

int **fsOpen** (const char \*name, int flags, **fsFile\_t** \*\*file)

Open the selected file to input and/or output. [More...](#)

int **fsClose** (**fsFile\_t** \*f)

Close a file handle. [More...](#)

int **fsRead** (**fsFile\_t** \*f, int n, void \*p)

Try to read a number of bytes from a file. [More...](#)

int **fs.ReadByte** (**fsFile\_t** \*f)

Try to read the next byte from the file. [More...](#)

int **fsReadUntil** (**fsFile\_t** \*f, const **uint8\_t** \*delims, int ndelim, int n, void \*p)

Read bytes from the input file until either one of delims bytes is found, or n bytes have been read. [More...](#)

int **fsWrite** (**fsFile\_t** \*f, int n, const void \*p)

Try to write a number of bytes to a file. [More...](#)

int **fsWriteByte** (**fsFile\_t** \*f, int b)  
Try to write a byte to a file. [More...](#)

int **fsTell** (**fsFile\_t** \*f)  
Return the current read/write position of a file. [More...](#)

int **fsSeek** (**fsFile\_t** \*f, int whence, int offset)  
Set the read/write position for a file. [More...](#)

int **fsLength** (**fsFile\_t** \*f)  
Return the length of an open file in bytes. [More...](#)

int **fsRemove** (const char \*name)  
Delete file name from the file system. [More...](#)

int **fsRename** (const char \*oldname, const char \*newname)  
Rename the file oname to newname. [More...](#)

int **fsExist** (const char \*name)  
Check if file name exists. [More...](#)

---

# Detailed Description

---

File System abstraction.

---

Generated on Mon Mar 27 2017 15:42:52 for LAR Library by [doxygen](#) 1.8.9.1

# LAR Library 1.14

Main Page	Related Pages	Modules	Data Structures
<b>Files</b>			
File List	Globals		

C: > Projects > lalarlib > larlib > common > inc > larlib >

[Data Structures](#) | [Typedefs](#) | [Functions](#)

## ini.h File Reference

Read/parsing of MS-like INI files. More...

#include <[larlib/base.h](#)>

[Go to the source code of this file.](#)

## Data Structures

---

struct **iniElement\_t**

Descriptor of elements allowed in an INI file. [More...](#)

---

## Typedefs

---

typedef char \*(\* **iniLineReader\_t**) (void \*p, char \*line, int maxline)

Signature of function used by **iniParseLines()** to  
read next line from input file. More...

---

## Functions

---

int **iniParse** (**iniElement\_t** elements[], const char \*fileName)

Process the given INI file and calls handler functions according to the element table given. [More...](#)

int **iniParseString** (**iniElement\_t** elements[], const char \*contents)

Process a previously INI file loaded in memory. [More...](#)

int **iniParseLines** (**iniElement\_t** elements[], **iniLineReader\_t**

readLine, void \*p)

Process the lines of an INI file returned from repeated calls to readLine. [More...](#)

---

## Detailed Description

---

Read/parsing of MS-like INI files.

---

Generated on Mon Mar 27 2017 15:42:52 for LAR Library by [doxygen](#) 1.8.9.1

# LAR Library 1.14

Main Page	Related Pages	Modules	Data Structures
<b>Files</b>			
File List	Globals		

C: > Projects > lalarlib > larlib > common > inc > larlib >

Macros | Typedefs | Enumerations | Functions

## log.h File Reference

General Logging System. [More...](#)

```
#include <larlib/base.h> #include <stdarg.h>
```

[Go to the source code of this file.](#)

## Macros

---

```
#define LOG_ALL_CHANNELS 0, 256
```

This is a short-cut to define the parameters of all channels in one single call to **logSetChannels()**.

```
#define LOG_ASSERT(cond)
```

Assertion using larlib.log for output. More...

---

## Typedefs

typedef int(\* **logWriteFunction\_t**) (void \*context, **uint16\_t** channelLevel, const char \*msg, int msgLength)  
Type of the function that does the actual writing on a trace channel. [More...](#)

typedef int(\* **logDumpFunction\_t**) (**logWriteFunction\_t** write, void \*context, **uint16\_t** channelLevel, const void \*data, int dataLen)  
Type of function called to format the output on **logDump()**. [More...](#)

---

## Enumerations

---

```
enum LogLevel_t {
    LOG_DEBUG = 0x0000, LOG_INFO = 0x2000,
    LOG_WARNING = 0x4000, LOG_ERROR = 0x6000,
    LOG_CRITICAL = 0x8000, LOG_ALWAYS = 0xFF00
}
```

General tracing levels. [More...](#)

```
enum logReservedChannels_t {
    LOG_CH_BASE = 200, LOG_CH_CONV, LOG_CH_DATE,
    LOG_CH_FS,
    LOG_CH_INI, LOG_CH_MEM, LOG_CH_TAB,
    LOG_CH_STEP,
    LOG_CH_FORMAT, LOG_CH_TASK
}
```

List of reserved log channels. [More...](#)

---

## Functions

int **logWriteTeliumTrace** (void \*context, **uint16\_t** channelLevel, const char \*msg, int msgLength)

A **logWriteFunction\_t** that can be used as parameter to **logSetChannels()** and writes traces using Telium's trace() function. [More...](#)

int **logWriteTeliumRemoteDebugger** (void \*context, **uint16\_t** channelLevel, const char \*msg, int msgLength)

A **logWriteFunction\_t** that can be used as parameter to **logSetChannels()** and writes traces to the Remote Debugger console on IngeDev. [More...](#)

int **logDumpFormattedAscii** (**logWriteFunction\_t** write, void \*context, **uint16\_t** channelLevel, const void \*data, int dataLen)

dumpFormatFunction that formats the received buffer as a two-column hexadecimal and ASCII display and calls the received write function. [More...](#)

int **logSetChannels** (**uint8\_t** firstChannel, int numChannels, **uint16\_t** level, **logWriteFunction\_t** writeFunction, **logDumpFunction\_t** dumpFormatFunction, void \*context)

Configures one or more tracing channels. [More...](#)

int **logChannelsEnabled** (**uint16\_t** channelLevel)

Check if a log channel is enabled for a specific priority level.  
[More...](#)

int **logPrintf** (**uint16\_t** channelLevel, const char \*fmt,...)

sprintf() like formatted traces. [More...](#)

int **logPrintvf** (**uint16\_t** channelLevel, const char \*fmt, va\_list va)

A version of **logPrintf()** that accepts a va\_list as parameter.  
[More...](#)

`int logDump (uint16_t channelLevel, const void *buffer, int size)`

Write a block of data as a binary dump. [More...](#)

---

# Detailed Description

---

## General Logging System.

---

Generated on Mon Mar 27 2017 15:42:52 for LAR Library by [doxygen](#) 1.8.9.1

# LAR Library 1.14

Main Page	Related Pages	Modules	Data Structures
<b>Files</b>			
File List	Globals		

C: > Projects > lalarlib > larlib > common > inc > larlib >

Functions

## mem.h File Reference

Memory Management. [More...](#)

#include <[larlib/base.h](#)>

[Go to the source code of this file.](#)

## Functions

---

`void * memAlloc (int nb)`

Allocate a block of memory from the heap. [More...](#)

`void * memAllocZero (int nb)`

Allocate and zero a block of memory from the heap. [More...](#)

`void * memRealloc (void *oldp, int nb)`

Resize a block of memory, or allocate a new one. [More...](#)

`void memFree (void *p)`

Release a previously allocated block. [More...](#)

---

# Detailed Description

---

Memory Management.

---

Generated on Mon Mar 27 2017 15:42:52 for LAR Library by [doxygen](#) 1.8.9.1

# LAR Library 1.14

Main Page	Related Pages	Modules	Data Structures
<b>Files</b>			
File List	Globals		

C: > Projects > lalarlib > larlib > common > inc > larlib >

[Data Structures](#) | [Typedefs](#) | [Functions](#)

## step.h File Reference

State-machine for transaction processing. [More...](#)

#include <[larlib/base.h](#)>

[Go to the source code of this file.](#)

## Data Structures

---

struct **stepAction\_t**

Each action in the step machine. [More...](#)

---

## Typedefs

---

typedef struct **step\_t** **step\_t**

Opaque type of a handle to a step-machine.

typedef int(\*) **stepFunction\_t** (**step\_t** \*s, **stepAction\_t** \*a,  
void \*c)

Type signature of the function associated with  
each action on the step machine. [More...](#)

---

## Functions

---

`step_t * stepCreate (stepAction_t *actions, int nactions)`

Create a new step machine. [More...](#)

`int stepRun (step_t *s, void *c)`

Execute the step machine. [More...](#)

`int stepGotoLabel (step_t *s, int label)`

Change the next action to be executed. [More...](#)

`void stepDestroy (step_t *s)`

Release the resources associated with a step-machine.

[More...](#)

---

## Detailed Description

---

State-machine for transaction processing.

---

Generated on Mon Mar 27 2017 15:42:52 for LAR Library by [doxygen](#) 1.8.9.1

# LAR Library 1.14

Main Page	Related Pages	Modules	Data Structures
<b>Files</b>			
File List	Globals		

C: > Projects > lalarlib > larlib > common > inc > larlib >

[Typedefs](#) | [Enumerations](#) | [Functions](#)

## tab.h File Reference

Table management. More...

#include <[larlib/base.h](#)>

[Go to the source code of this file.](#)

## Typedefs

---

`typedef struct table_t table_t`

Opaque type of a table instance.

---

## Enumerations

---

enum **tabErrors\_t** { **TAB\_ERR\_DELETED** = 1 }

Return codes specific to this module. [More...](#)

---

## Functions

int **tabOpen** (const char \*fname, **table\_t** \*\*t)  
Open an existing table. [More...](#)

int **tabCreate** (const char \*fname, int recSize, **table\_t** \*\*t)  
Create a new table or reset an existing one. [More...](#)

void **tabClose** (**table\_t** \*t)  
Close a **table\_t** instance. [More...](#)

int **tabFlush** (**table\_t** \*t)  
Force an update of the table file. [More...](#)

int **tabRecSize** (**table\_t** \*t)  
Return the record size for a table. [More...](#)

int **tabGetStamp** (**table\_t** \*t, **uint32\_t** \*stamp)  
Return the stamp value of the table. [More...](#)

int **tabSetStamp** (**table\_t** \*t, **uint32\_t** stamp)  
Modify the stamp value of the table. [More...](#)

int **tabGet** (**table\_t** \*t, int recn, void \*recp)  
Read an entry from the table. [More...](#)

int **tabPut** (**table\_t** \*t, int recn, const void \*recp)  
Write or update an entry on the table. [More...](#)

int **tabAppend** (**table\_t** \*t, const void \*recp)  
Append an entry to the table. [More...](#)

int **tabSize** (**table\_t** \*t)  
Return the total number of entries on the table. [More...](#)

int **tabCount** (**table\_t** \*t)  
Return the number of non-deleted entries on the table. [More...](#)

int **tabDelete** (**table\_t** \*t, int recn)  
Mark an entry as deleted. [More...](#)

int **tabUnDelete** (**table\_t** \*t, int recn)  
Remove the deleted mark of an entry. [More...](#)

int **tabPack** (const char \*fname)  
Permanently remove from the table all entries marked as deleted. [More...](#)

int **tabGoFirst** (**table\_t** \*t)  
Move the current entry marker to the first (zero'th) element on the table. [More...](#)

int **tabGetNext** (**table\_t** \*t, void \*recp)  
Read the current entry and move to the next. [More...](#)

int **tabGetPrev** (**table\_t** \*t, void \*recp)  
Read the previous element. [More...](#)

int **tabGoLast** (**table\_t** \*t)  
Move to the last element on the table. [More...](#)

int **tabFindNext** (**table\_t** \*t, void \*recp, const void \*keyp,  
int(\*cmpFn)(const void \*recp, const void \*keyp))  
Search incrementally the table for an entry. [More...](#)

int **tabFind** (**table\_t** \*t, void \*recp, const void \*keyp, int(\*cmpFn)  
(const void \*recp, const void \*keyp))  
Search the table for an element. [More...](#)

## Detailed Description

---

Table management.

---

Generated on Mon Mar 27 2017 15:42:52 for LAR Library by [doxygen](#) 1.8.9.1

# LAR Library 1.14

Main Page	Related Pages	Modules	Data Structures
<b>Files</b>			
File List	Globals		

C: > Projects > lalarlib > larlib > common > inc > larlib >

Macros | Typedefs | Functions

## task.h File Reference

Multi-threading. [More...](#)

#include <[larlib/base.h](#)>

[Go to the source code of this file.](#)

## Macros

---

```
#define TASK_INFINITE_TIMEOUT (0xFFFFFFFFLU)
```

Use this constant to wait forever when a timeout is necessary. More...

---

## Typedefs

---

typedef struct **taskThread\_t** **taskThread\_t**

A handle to a running thread.

typedef void(\*) **taskThreadFunction\_t** (void  
\*param)

Type of a function pointer to be  
used as entry point of a new  
thread. [More...](#)

typedef struct **taskQueue\_t** **taskQueue\_t**

Handle to a multi-threading queue.

typedef struct **taskSemaphore\_t** **taskSemaphore\_t**

Handle to a semaphore.

typedef struct **taskEvent\_t** **taskEvent\_t**

Handle to an event.

---

## Functions

`taskThread_t * taskThreadCreate (taskThreadFunction_t fn,  
void *param, uint32_t stackSize)`  
Create a new execution thread with entry point  
fn. [More...](#)

`void taskSleep (uint32_t timeout)`  
Sleep the current running thread. [More...](#)

`taskThread_t * taskThreadCurrent (void)`  
Return the handle of the current running  
thread. [More...](#)

`taskQueue_t * taskQueueCreate (int maxItems)`  
Create a new queue with up to `maxItems`  
allowed. [More...](#)

`void taskQueueDestroy (taskQueue_t *q)`  
Destroy a queue and discards all pending  
elements. [More...](#)

`int taskQueuePost (taskQueue_t *q, void *item)`  
Insert an element on the back of the queue q.  
[More...](#)

`int taskQueuePop (taskQueue_t *q, void **item,  
uint32_t timeout)`  
Extract the element from the front of the queue.  
[More...](#)

`taskSemaphore_t * taskSemaphoreCreate (int limit)`  
Create a semaphore with a maximum entry  
count of `limit`. [More...](#)

`void taskSemaphoreDestroy (taskSemaphore_t`

`*s)`

Destroy an existing semaphore. [More...](#)

int **taskSemaphoreAcquire** (**taskSemaphore\_t**  
`*s, uint32_t timeout)`

Acquire (also called *wait* or *down*) the  
semaphore. [More...](#)

int **taskSemaphoreRelease** (**taskSemaphore\_t**  
`*s)`

Release (also called *signal* or *up*) a previously  
acquired semaphore. [More...](#)

**taskEvent\_t** \* **taskEventCreate** (`void`)

Create a new event handler. [More...](#)

**taskEvent\_t** \* **taskEventCreateSystem** (`uint32_t bitmask`)

Creates an event handler to wait for system  
peripheral events. [More...](#)

void **taskEventDestroy** (**taskEvent\_t** \*`e`)

Release the resources associated with an  
event handler. [More...](#)

int **taskEventSignal** (**taskEvent\_t** \*`e`)

Set an event to the *signaled* state. [More...](#)

int **taskEventCheck** (**taskEvent\_t** \*`e`)

Check if an event is signaled, without waiting or  
clearing the signal. [More...](#)

int **taskEventWait** (**taskEvent\_t** \*`e`, `uint32_t`  
`timeout`)

Wait for `e` to change to signaled state. [More...](#)

int **taskEventWaitAny** (**taskEvent\_t** \*`events[]`, int  
`nevents`, `uint32_t timeout`)

Wait for at least one among a list of events to be signaled. [More...](#)

---

int **taskEventClear** (**taskEvent\_t** \*e)  
Set an event as *non-signaled*. [More...](#)

## Detailed Description

---

Multi-threading.

---

Generated on Mon Mar 27 2017 15:42:52 for LAR Library by [doxygen](#) 1.8.9.1

# LAR Library 1.14

The screenshot shows a navigation bar with tabs for Main Page, Related Pages, Modules, and Data Structures. Below this is a 'Files' section with a 'File List' tab and a 'Globals' tab, which is currently active. A search bar below the tabs contains the letters 'b', 'c', 'd', 'f', 'i', 'l', 'm', 's', 't', 'u'. To the right of the search bar are tabs for All, Functions, Typedefs, Enumerations, Enumerator, and Macros.

Here is a list of all documented functions, variables, defines, enums, and typedefs with links to the documentation:

## - b -

- BASE\_ERR\_ACCESS : [base.h](#)
- BASE\_ERR\_CANCEL : [base.h](#)
- BASE\_ERR\_DATA\_NOT\_FOUND : [base.h](#)
- BASE\_ERR\_INVALID\_HANDLE : [base.h](#)
- BASE\_ERR\_INVALID\_PARAMETER : [base.h](#)
- BASE\_ERR\_OK : [base.h](#)
- BASE\_ERR\_OVERFLOW : [base.h](#)
- BASE\_ERR\_RESOURCE\_PB : [base.h](#)
- BASE\_ERR\_TIMEOUT : [base.h](#)
- BASE\_LENGTH : [base.h](#)
- BASE\_MAX : [base.h](#)
- BASE\_MIN : [base.h](#)
- baseErrors\_t : [base.h](#)
- BITS\_BLT\_AND : [bits.h](#)
- BITS\_BLT\_COPY : [bits.h](#)
- BITS\_BLT\_NOT : [bits.h](#)
- BITS\_BLT\_NOT\_AND : [bits.h](#)
- BITS\_BLT\_OR : [bits.h](#)
- BITS\_BLT\_XOR : [bits.h](#)
- bitsAppend() : [bits.h](#)
- bitsAppendInt() : [bits.h](#)
- bitsBlt() : [bits.h](#)
- bitsBltOp\_t : [bits.h](#)
- bitsBuffer\_t : [bits.h](#)

- bitsCopy() : **bits.h**
  - bitsExtractInt() : **bits.h**
  - bitsGet() : **bits.h**
  - bitsRotateLeft() : **bits.h**
  - bitsRotateRight() : **bits.h**
  - bitsSet() : **bits.h**
  - bitsSetRange() : **bits.h**
  - bitsShiftLeft() : **bits.h**
  - bitsShiftRight() : **bits.h**
- 

Generated on Mon Mar 27 2017 15:42:53 for LAR Library by  doxygen 1.8.9.1

# LAR Library 1.14

The screenshot shows a navigation bar with tabs for Main Page, Related Pages, Modules, and Data Structures. Below this is a 'Files' tab, which is selected. Under 'Files', there are two tabs: File List and Globals, with Globals being the active one. A secondary navigation bar below shows categories: All, Functions, Typedefs, Enumerations, Enumerator, and Macros. The 'All' category is selected. At the bottom, a horizontal menu lists letters: b, c, d, f, i, l, m, s, t, u. The letter 'c' is highlighted in a darker shade.

Here is a list of all documented functions, variables, defines, enums, and typedefs with links to the documentation:

## - C -

- CONV\_ERR\_INVALID\_BASE : [conv.h](#)
- convBase64ToBuf() : [conv.h](#)
- convBcdToInt() : [conv.h](#)
- convBcdToTxt() : [conv.h](#)
- convBufToBase64() : [conv.h](#)
- convBufToHex() : [conv.h](#)
- convBufToInt() : [conv.h](#)
- convErrors\_t : [conv.h](#)
- convHexToBuf() : [conv.h](#)
- convIntToBcd() : [conv.h](#)
- convIntToBcdPad() : [conv.h](#)
- convIntToBuf() : [conv.h](#)
- convIntToBufPad() : [conv.h](#)
- convIntToTxt() : [conv.h](#)
- convIntToTxtPad() : [conv.h](#)
- convTxtToBcd() : [conv.h](#)
- convTxtToInt() : [conv.h](#)
- convTxtToInt16() : [conv.h](#)
- convTxtToInt32() : [conv.h](#)
- convTxtToInt64() : [conv.h](#)
- convTxtToInt8() : [conv.h](#)

# LAR Library 1.14

The screenshot shows a navigation bar with tabs: Main Page, Related Pages, Modules, Data Structures, Files, File List, and Globals. Below the navigation bar is a sidebar with categories: All, Functions, Typedefs, Enumerations, Enumerator, and Macros. The 'd' category is highlighted in the sidebar.

Here is a list of all documented functions, variables, defines, enums, and typedefs with links to the documentation:

## - d -

- DATE\_APR : [date.h](#)
- DATE\_AUG : [date.h](#)
- DATE\_DEC : [date.h](#)
- DATE\_FEB : [date.h](#)
- DATE\_FRI : [date.h](#)
- DATE\_JAN : [date.h](#)
- DATE\_JUL : [date.h](#)
- DATE\_JUN : [date.h](#)
- DATE\_MAR : [date.h](#)
- DATE\_MAY : [date.h](#)
- DATE\_MON : [date.h](#)
- DATE\_NOV : [date.h](#)
- DATE\_OCT : [date.h](#)
- DATE\_SAT : [date.h](#)
- DATE\_SEP : [date.h](#)
- DATE\_SUN : [date.h](#)
- DATE\_THU : [date.h](#)
- DATE\_TUE : [date.h](#)
- DATE\_WED : [date.h](#)
- dateAddDays() : [date.h](#)
- dateCompare() : [date.h](#)
- dateDiff() : [date.h](#)
- dateFromJulianDay() : [date.h](#)
- dateGet() : [date.h](#)

- dateIsLeap() : **date.h**
- dateIsValid() : **date.h**
- dateMonths\_t : **date.h**
- dateSet() : **date.h**
- dateTimeAddSeconds() : **date.h**
- dateTimeCompare() : **date.h**
- dateTimeDiff() : **date.h**
- dateTimeGet() : **date.h**
- dateTimeGetMs() : **date.h**
- dateTimeIsValid() : **date.h**
- dateTimeSet() : **date.h**
- dateToJulianDay() : **date.h**
- dateWeekday() : **date.h**
- dateWeekdays\_t : **date.h**
- deque\_t : **deque.h**
- dequeCreate() : **deque.h**
- dequeDestroy() : **deque.h**
- dequeGet() : **deque.h**
- dequeGetBack() : **deque.h**
- dequeGetFront() : **deque.h**
- dequeInsert() : **deque.h**
- dequeLength() : **deque.h**
- dequePopBack() : **deque.h**
- dequePopFront() : **deque.h**
- dequePushBack() : **deque.h**
- dequePushFront() : **deque.h**
- dequeRemove() : **deque.h**
- dequeSet() : **deque.h**

# LAR Library 1.14

The screenshot shows a navigation bar with tabs for Main Page, Related Pages, Modules, and Data Structures. Below this is a sub-navigation bar for 'Files' with tabs for File List and Globals. The Globals tab is selected. A search bar contains the letter 'f'. Below the search bar is a filter bar with tabs for All, Functions, Typedefs, Enumerations, Enumerator, and Macros. The 'All' tab is selected. A scrollable list of symbols starts with:

b	c	d	f	i	l	m	s	t	u
---	---	---	---	---	---	---	---	---	---

Here is a list of all documented functions, variables, defines, enums, and typedefs with links to the documentation:

- f -

- FORMAT\_PACK : [format.h](#)
- FORMAT\_UNPACK : [format.h](#)
- formatBodyBufHex() : [format.h](#)
- formatBodyISO8583() : [format.h](#)
- formatBodyMemcpy() : [format.h](#)
- formatBodyPacked() : [format.h](#)
- formatBodySkip() : [format.h](#)
- formatBodyStrcpy() : [format.h](#)
- formatBodyUint16Dec() : [format.h](#)
- formatBodyUint32Dec() : [format.h](#)
- formatBodyUint64Buf() : [format.h](#)
- formatBodyUint64Dec() : [format.h](#)
- formatBodyUint8Dec() : [format.h](#)
- formatDirection\_t : [format.h](#)
- formatFilterISO8583() : [format.h](#)
- formatPack() : [format.h](#)
- formatPadSkip() : [format.h](#)
- formatPadSpaces() : [format.h](#)
- formatPadZeroBytes() : [format.h](#)
- formatPadZeroes() : [format.h](#)
- formatSizeAsc2() : [format.h](#)
- formatSizeAsc3() : [format.h](#)
- formatSizeBcd2() : [format.h](#)
- formatSizeBcd3() : [format.h](#)

- `formatUnpack()` : **format.h**
- `FS_OPEN_APPEND` : **fs.h**
- `FS_OPEN_CREATE` : **fs.h**
- `FS_OPEN_READ` : **fs.h**
- `FS_OPEN_RESET` : **fs.h**
- `FS_OPEN_WRITE` : **fs.h**
- `FS_WHENCE_CUR` : **fs.h**
- `FS_WHENCE_END` : **fs.h**
- `FS_WHENCE_SET` : **fs.h**
- `fsClose()` : **fs.h**
- `fsExist()` : **fs.h**
- `fsFile_t` : **fs.h**
- `fsGetFileNamePrefix()` : **fs.h**
- `fsLength()` : **fs.h**
- `fsOpen()` : **fs.h**
- `fsOpenFlags_t` : **fs.h**
- `fsPopFileNamePrefix()` : **fs.h**
- `fsPushFileNamePrefix()` : **fs.h**
- `fsRead()` : **fs.h**
- `fs.ReadByte()` : **fs.h**
- `fsReadUntil()` : **fs.h**
- `fsRemove()` : **fs.h**
- `fsRename()` : **fs.h**
- `fsSeek()` : **fs.h**
- `fsSeekWhence_t` : **fs.h**
- `fsSetFileNamePrefix()` : **fs.h**
- `fsTell()` : **fs.h**
- `fsWrite()` : **fs.h**
- `fsWriteByte()` : **fs.h**

# LAR Library 1.14

The screenshot shows a navigation bar with tabs for Main Page, Related Pages, Modules, and Data Structures. Below this is a sub-navigation bar for Files, with tabs for File List and Globals, where Globals is selected. A third-level navigation bar shows categories: All, Functions, Typedefs, Enumerations, Enumerator, and Macros. Under All, there is a list of letters: b, c, d, f, i, l, m, s, t, u, with 'i' being the active category. The background has a light blue gradient.

Here is a list of all documented functions, variables, defines, enums, and typedefs with links to the documentation:

- i -

- iniLineReader\_t : [ini.h](#)
- iniParse() : [ini.h](#)
- iniParseLines() : [ini.h](#)
- iniParseString() : [ini.h](#)
- int16\_t : [base.h](#)
- int32\_t : [base.h](#)
- int64\_t : [base.h](#)
- int8\_t : [base.h](#)

# LAR Library 1.14

The screenshot shows a navigation bar with tabs for Main Page, Related Pages, Modules, and Data Structures. Below this is a 'Files' tab, which is selected. Under 'Files', there are two tabs: File List and Globals, with 'Globals' being the active one. A sub-navigation bar below 'Globals' includes tabs for All, Functions, Typedefs, Enumerations, Enumerator, and Macros. The 'All' tab is selected. At the bottom of this bar are links for b, c, d, f, i, l, m, s, t, u. The letter 'l' is highlighted in a dark blue box, indicating it is the current page.

Here is a list of all documented functions, variables, defines, enums, and typedefs with links to the documentation:

- l -

- LARLIB\_VERSION : [base.h](#)
- larlibGetVersion() : [base.h](#)
- LOG\_ALL\_CHANNELS : [log.h](#)
- LOG\_ALWAYS : [log.h](#)
- LOG\_ASSERT : [log.h](#)
- LOG\_CH\_BASE : [log.h](#)
- LOG\_CH\_CONV : [log.h](#)
- LOG\_CH\_DATE : [log.h](#)
- LOG\_CH\_FORMAT : [log.h](#)
- LOG\_CH\_FS : [log.h](#)
- LOG\_CH\_INI : [log.h](#)
- LOG\_CH\_MEM : [log.h](#)
- LOG\_CH\_STEP : [log.h](#)
- LOG\_CH\_TAB : [log.h](#)
- LOG\_CH\_TASK : [log.h](#)
- LOG\_CRITICAL : [log.h](#)
- LOG\_DEBUG : [log.h](#)
- LOG\_ERROR : [log.h](#)
- LOG\_INFO : [log.h](#)
- LOG\_WARNING : [log.h](#)
- logChannelsEnabled() : [log.h](#)
- logDump() : [log.h](#)
- logDumpFormattedAscii() : [log.h](#)
- logDumpFunction\_t : [log.h](#)

- `logLevel_t` : **log.h**
  - `logPrintf()` : **log.h**
  - `logPrintvf()` : **log.h**
  - `logReservedChannels_t` : **log.h**
  - `logSetChannels()` : **log.h**
  - `logWriteFunction_t` : **log.h**
  - `logWriteTeliumRemoteDebugger()` : **log.h**
  - `logWriteTeliumTrace()` : **log.h**
- 

Generated on Mon Mar 27 2017 15:42:53 for LAR Library by [doxygen](#) 1.8.9.1

# LAR Library 1.14

The screenshot shows a navigation bar with tabs: Main Page, Related Pages, Modules, Data Structures, Files, File List, and Globals. The Globals tab is selected. Below this is a sub-navigation bar with tabs: All, Functions, Typedefs, Enumerations, Enumerator, and Macros. The 'All' tab is selected. A list of letters (b, c, d, f, i, l, m, s, t, u) is shown, with 'm' being the active selection. The main content area displays a list of global symbols starting with 'm'.

Here is a list of all documented functions, variables, defines, enums, and typedefs with links to the documentation:

## - m -

- memAlloc() : [mem.h](#)
- memAllocZero() : [mem.h](#)
- memFree() : [mem.h](#)
- memRealloc() : [mem.h](#)

# LAR Library 1.14

The screenshot shows a navigation bar with tabs: Main Page, Related Pages, Modules, Data Structures, Files, File List, and Globals. Below this is a search bar with dropdowns for All, Functions, Typedefs, Enumerations, Enumerator, and Macros. A sidebar lists categories: b, c, d, f, i, l, m, s, t, u. The 's' category is highlighted.

Here is a list of all documented functions, variables, defines, enums, and typedefs with links to the documentation:

## - S -

- step\_t : [step.h](#)
- stepCreate() : [step.h](#)
- stepDestroy() : [step.h](#)
- stepFunction\_t : [step.h](#)
- stepGotoLabel() : [step.h](#)
- stepRun() : [step.h](#)

# LAR Library 1.14

The screenshot shows a navigation bar with tabs: Main Page, Related Pages, Modules, and Data Structures. Below this is a 'Files' tab, which is active. Under 'Files', there are two tabs: File List and Globals, with Globals being the active one. A sub-navigation bar below Globals has tabs for All, Functions, Typedefs, Enumerations, Enumerator, and Macros. The 'All' tab is active. Below this is a horizontal menu with letters b, c, d, f, i, l, m, s, t, u. The letter 't' is highlighted in a dark blue box.

Here is a list of all documented functions, variables, defines, enums, and typedefs with links to the documentation:

- t -

- TAB\_ERR\_DELETED : [tab.h](#)
- tabAppend() : [tab.h](#)
- tabClose() : [tab.h](#)
- tabCount() : [tab.h](#)
- tabCreate() : [tab.h](#)
- tabDelete() : [tab.h](#)
- tabErrors\_t : [tab.h](#)
- tabFind() : [tab.h](#)
- tabFindNext() : [tab.h](#)
- tabFlush() : [tab.h](#)
- tabGet() : [tab.h](#)
- tabGetNext() : [tab.h](#)
- tabGetPrev() : [tab.h](#)
- tabGetStamp() : [tab.h](#)
- tabGoFirst() : [tab.h](#)
- tabGoLast() : [tab.h](#)
- table\_t : [tab.h](#)
- tabOpen() : [tab.h](#)
- tabPack() : [tab.h](#)
- tabPut() : [tab.h](#)
- tabRecSize() : [tab.h](#)
- tabSetStamp() : [tab.h](#)
- tabSize() : [tab.h](#)
- tabUnDelete() : [tab.h](#)

- TASK\_INFINITE\_TIMEOUT : **task.h**
- taskEvent\_t : **task.h**
- taskEventCheck() : **task.h**
- taskEventClear() : **task.h**
- taskEventCreate() : **task.h**
- taskEventCreateSystem() : **task.h**
- taskEventDestroy() : **task.h**
- taskEventSignal() : **task.h**
- taskEventWait() : **task.h**
- taskEventWaitAny() : **task.h**
- taskQueue\_t : **task.h**
- taskQueueCreate() : **task.h**
- taskQueueDestroy() : **task.h**
- taskQueuePop() : **task.h**
- taskQueuePost() : **task.h**
- taskSemaphore\_t : **task.h**
- taskSemaphoreAcquire() : **task.h**
- taskSemaphoreCreate() : **task.h**
- taskSemaphoreDestroy() : **task.h**
- taskSemaphoreRelease() : **task.h**
- taskSleep() : **task.h**
- taskThread\_t : **task.h**
- taskThreadCreate() : **task.h**
- taskThreadCurrent() : **task.h**
- taskThreadFunction\_t : **task.h**

# LAR Library 1.14

The screenshot shows a navigation bar with tabs: Main Page, Related Pages, Modules, Data Structures, Files, File List, and Globals. The Globals tab is currently active. Below the navigation bar is a search bar with the placeholder "Search" and a "Search" button. Underneath the search bar is a filter bar with categories: All, Functions, Typedefs, Enumerations, Enumerator, and Macros. The "All" category is selected. At the bottom of the filter bar is a list of letters: b, c, d, f, i, l, m, s, t, u, where 'u' is also highlighted.

Here is a list of all documented functions, variables, defines, enums, and typedefs with links to the documentation:

## - u -

- `uint16_t` : [base.h](#)
- `uint32_t` : [base.h](#)
- `uint64_t` : [base.h](#)
- `uint8_t` : [base.h](#)

# LAR Library 1.14

The screenshot shows a navigation bar with tabs: Main Page, Related Pages, Modules, and Data Structures. Below this is a 'Files' section with tabs: File List and Globals (selected). Under Globals, there are tabs: All, Functions (selected), Typedefs, Enumerations, Enumerator, and Macros. A horizontal menu below these tabs includes links for b, c, d, f, i, l, m, s, and t.

## - b -

- bitsAppend() : [bits.h](#)
- bitsAppendInt() : [bits.h](#)
- bitsBlt() : [bits.h](#)
- bitsCopy() : [bits.h](#)
- bitsExtractInt() : [bits.h](#)
- bitsGet() : [bits.h](#)
- bitsRotateLeft() : [bits.h](#)
- bitsRotateRight() : [bits.h](#)
- bitsSet() : [bits.h](#)
- bitsSetRange() : [bits.h](#)
- bitsShiftLeft() : [bits.h](#)
- bitsShiftRight() : [bits.h](#)

## - c -

- convBase64ToBuf() : [conv.h](#)
- convBcdToInt() : [conv.h](#)
- convBcdToTxt() : [conv.h](#)
- convBufToBase64() : [conv.h](#)
- convBufToHex() : [conv.h](#)
- convBufToInt() : [conv.h](#)
- convHexToBuf() : [conv.h](#)
- convIntToBcd() : [conv.h](#)
- convIntToBcdPad() : [conv.h](#)

- convIntToBuf() : **conv.h**
- convIntToBufPad() : **conv.h**
- convIntToTxt() : **conv.h**
- convIntToTxtPad() : **conv.h**
- convTxtToBcd() : **conv.h**
- convTxtToInt() : **conv.h**
- convTxtToInt16() : **conv.h**
- convTxtToInt32() : **conv.h**
- convTxtToInt64() : **conv.h**
- convTxtToInt8() : **conv.h**

- **d** -

- dateAddDays() : **date.h**
- dateCompare() : **date.h**
- dateDiff() : **date.h**
- dateFromJulianDay() : **date.h**
- dateGet() : **date.h**
- dateIsLeap() : **date.h**
- dateIsValid() : **date.h**
- dateSet() : **date.h**
- dateTimeAddSeconds() : **date.h**
- dateTimeCompare() : **date.h**
- dateTimeDiff() : **date.h**
- dateTimeGet() : **date.h**
- dateTimeGetMs() : **date.h**
- dateTimeIsValid() : **date.h**
- dateTimeSet() : **date.h**
- dateToJulianDay() : **date.h**
- dateWeekday() : **date.h**
- dequeCreate() : **deque.h**
- dequeDestroy() : **deque.h**
- dequeGet() : **deque.h**
- dequeGetBack() : **deque.h**
- dequeGetFront() : **deque.h**
- dequeInsert() : **deque.h**
- dequeLength() : **deque.h**
- dequePopBack() : **deque.h**
- dequePopFront() : **deque.h**
- dequePushBack() : **deque.h**

- dequePushFront() : **deque.h**
- dequeRemove() : **deque.h**
- dequeSet() : **deque.h**

- f -

- formatBodyBufHex() : **format.h**
- formatBodyISO8583() : **format.h**
- formatBodyMemcpy() : **format.h**
- formatBodyPacked() : **format.h**
- formatBodySkip() : **format.h**
- formatBodyStrcpy() : **format.h**
- formatBodyUint16Dec() : **format.h**
- formatBodyUint32Dec() : **format.h**
- formatBodyUint64Buf() : **format.h**
- formatBodyUint64Dec() : **format.h**
- formatBodyUint8Dec() : **format.h**
- formatFilterISO8583() : **format.h**
- formatPack() : **format.h**
- formatPadSkip() : **format.h**
- formatPadSpaces() : **format.h**
- formatPadZeroBytes() : **format.h**
- formatPadZeroes() : **format.h**
- formatSizeAsc2() : **format.h**
- formatSizeAsc3() : **format.h**
- formatSizeBcd2() : **format.h**
- formatSizeBcd3() : **format.h**
- formatUnpack() : **format.h**
- fsClose() : **fs.h**
- fsExist() : **fs.h**
- fsGetFileNamePrefix() : **fs.h**
- fsLength() : **fs.h**
- fsOpen() : **fs.h**
- fsPopFileNamePrefix() : **fs.h**
- fsPushFileNamePrefix() : **fs.h**
- fsRead() : **fs.h**
- fs.ReadByte() : **fs.h**
- fsReadUntil() : **fs.h**
- fsRemove() : **fs.h**
- fsRename() : **fs.h**

- fsSeek() : **fs.h**
- fsSetFileNamePrefix() : **fs.h**
- fsTell() : **fs.h**
- fsWrite() : **fs.h**
- fsWriteByte() : **fs.h**

- i -

- iniParse() : **ini.h**
- iniParseLines() : **ini.h**
- iniParseString() : **ini.h**

- l -

- larlibGetVersion() : **base.h**
- logChannelsEnabled() : **log.h**
- logDump() : **log.h**
- logDumpFormattedAscii() : **log.h**
- logPrintf() : **log.h**
- logPrintvf() : **log.h**
- logSetChannels() : **log.h**
- logWriteTeliumRemoteDebugger() : **log.h**
- logWriteTeliumTrace() : **log.h**

- m -

- memAlloc() : **mem.h**
- memAllocZero() : **mem.h**
- memFree() : **mem.h**
- memRealloc() : **mem.h**

- s -

- stepCreate() : **step.h**
- stepDestroy() : **step.h**
- stepGotoLabel() : **step.h**
- stepRun() : **step.h**

- t -

- tabAppend() : **tab.h**
- tabClose() : **tab.h**
- tabCount() : **tab.h**
- tabCreate() : **tab.h**
- tabDelete() : **tab.h**
- tabFind() : **tab.h**
- tabFindNext() : **tab.h**
- tabFlush() : **tab.h**
- tabGet() : **tab.h**
- tabGetNext() : **tab.h**
- tabGetPrev() : **tab.h**
- tabGetStamp() : **tab.h**
- tabGoFirst() : **tab.h**
- tabGoLast() : **tab.h**
- tabOpen() : **tab.h**
- tabPack() : **tab.h**
- tabPut() : **tab.h**
- tabRecSize() : **tab.h**
- tabSetStamp() : **tab.h**
- tabSize() : **tab.h**
- tabUnDelete() : **tab.h**
- taskEventCheck() : **task.h**
- taskEventClear() : **task.h**
- taskEventCreate() : **task.h**
- taskEventCreateSystem() : **task.h**
- taskEventDestroy() : **task.h**
- taskEventSignal() : **task.h**
- taskEventWait() : **task.h**
- taskEventWaitAny() : **task.h**
- taskQueueCreate() : **task.h**
- taskQueueDestroy() : **task.h**
- taskQueuePop() : **task.h**
- taskQueuePost() : **task.h**
- taskSemaphoreAcquire() : **task.h**
- taskSemaphoreCreate() : **task.h**
- taskSemaphoreDestroy() : **task.h**
- taskSemaphoreRelease() : **task.h**
- taskSleep() : **task.h**
- taskThreadCreate() : **task.h**
- taskThreadCurrent() : **task.h**

---

Generated on Mon Mar 27 2017 15:42:53 for LAR Library by [doxygen](#) 1.8.9.1

# LAR Library 1.14

Main Page	Related Pages	Modules	Data Structures		
Files					
File List	Globals				
All	Functions	Typedefs	Enumerations	Enumerator	Macros

- bitsBuffer\_t : [bits.h](#)
- deque\_t : [deque.h](#)
- fsFile\_t : [fs.h](#)
- iniLineReader\_t : [ini.h](#)
- int16\_t : [base.h](#)
- int32\_t : [base.h](#)
- int64\_t : [base.h](#)
- int8\_t : [base.h](#)
- logDumpFunction\_t : [log.h](#)
- logWriteFunction\_t : [log.h](#)
- step\_t : [step.h](#)
- stepFunction\_t : [step.h](#)
- table\_t : [tab.h](#)
- taskEvent\_t : [task.h](#)
- taskQueue\_t : [task.h](#)
- taskSemaphore\_t : [task.h](#)
- taskThread\_t : [task.h](#)
- taskThreadFunction\_t : [task.h](#)
- uint16\_t : [base.h](#)
- uint32\_t : [base.h](#)
- uint64\_t : [base.h](#)
- uint8\_t : [base.h](#)

# LAR Library 1.14

Main Page	Related Pages	Modules	Data Structures
Files			
File List	Globals		
All	Functions	Typedefs	Enumerations
			Enumerator
			Macros

- baseErrors\_t : [base.h](#)
- bitsBltOp\_t : [bits.h](#)
- convErrors\_t : [conv.h](#)
- dateMonths\_t : [date.h](#)
- dateWeekdays\_t : [date.h](#)
- formatDirection\_t : [format.h](#)
- fsOpenFlags\_t : [fs.h](#)
- fsSeekWhence\_t : [fs.h](#)
- logLevel\_t : [log.h](#)
- logReservedChannels\_t : [log.h](#)
- tabErrors\_t : [tab.h](#)

# LAR Library 1.14

The screenshot shows a navigation bar with tabs: Main Page, Related Pages, Modules, Data Structures, Files, File List, and Globals. The Globals tab is selected. Below this is a search bar with the placeholder 'Search...'. A sidebar on the left lists categories: All, Functions, Typedefs, Enumerations, Enumerator (which is selected and highlighted in blue), and Macros. At the bottom of the sidebar are links for base.h, bits.h, conv.h, date.h, and misc.h. The main content area displays a table of contents for the letter 't', with items like `BITSET_IS_SET`, `BITSET_SET`, `BITSET_UNSET`, and `BITSET_TOGGLE`.

## - b -

- `BASE_ERR_ACCESS` : [base.h](#)
- `BASE_ERR_CANCEL` : [base.h](#)
- `BASE_ERR_DATA_NOT_FOUND` : [base.h](#)
- `BASE_ERR_INVALID_HANDLE` : [base.h](#)
- `BASE_ERR_INVALID_PARAMETER` : [base.h](#)
- `BASE_ERR_OK` : [base.h](#)
- `BASE_ERR_OVERFLOW` : [base.h](#)
- `BASE_ERR_RESOURCE_PB` : [base.h](#)
- `BASE_ERR_TIMEOUT` : [base.h](#)
- `BITS_BLT_AND` : [bits.h](#)
- `BITS_BLT_COPY` : [bits.h](#)
- `BITS_BLT_NOT` : [bits.h](#)
- `BITS_BLT_NOT_AND` : [bits.h](#)
- `BITS_BLT_OR` : [bits.h](#)
- `BITS_BLT_XOR` : [bits.h](#)

## - c -

- `CONV_ERR_INVALID_BASE` : [conv.h](#)

## - d -

- `DATE_APR` : [date.h](#)
- `DATE_AUG` : [date.h](#)

- DATE\_DEC : **date.h**
- DATE\_FEB : **date.h**
- DATE\_FRI : **date.h**
- DATE\_JAN : **date.h**
- DATE\_JUL : **date.h**
- DATE\_JUN : **date.h**
- DATE\_MAR : **date.h**
- DATE\_MAY : **date.h**
- DATE\_MON : **date.h**
- DATE\_NOV : **date.h**
- DATE\_OCT : **date.h**
- DATE\_SAT : **date.h**
- DATE\_SEP : **date.h**
- DATE\_SUN : **date.h**
- DATE\_THU : **date.h**
- DATE\_TUE : **date.h**
- DATE\_WED : **date.h**

- f -

- FORMAT\_PACK : **format.h**
- FORMAT\_UNPACK : **format.h**
- FS\_OPEN\_APPEND : **fs.h**
- FS\_OPEN\_CREATE : **fs.h**
- FS\_OPEN\_READ : **fs.h**
- FS\_OPEN\_RESET : **fs.h**
- FS\_OPEN\_WRITE : **fs.h**
- FS\_WHENCE\_CUR : **fs.h**
- FS\_WHENCE\_END : **fs.h**
- FS\_WHENCE\_SET : **fs.h**

- I -

- LOG\_ALWAYS : **log.h**
- LOG\_CH\_BASE : **log.h**
- LOG\_CH\_CONV : **log.h**
- LOG\_CH\_DATE : **log.h**
- LOG\_CH\_FORMAT : **log.h**
- LOG\_CH\_FS : **log.h**
- LOG\_CH\_INI : **log.h**

- LOG\_CH\_MEM : **log.h**
- LOG\_CH\_STEP : **log.h**
- LOG\_CH\_TAB : **log.h**
- LOG\_CH\_TASK : **log.h**
- LOG\_CRITICAL : **log.h**
- LOG\_DEBUG : **log.h**
- LOG\_ERROR : **log.h**
- LOG\_INFO : **log.h**
- LOG\_WARNING : **log.h**

- t -

- TAB\_ERR\_DELETED : **tab.h**

# LAR Library 1.14

Main Page	Related Pages	Modules	Data Structures		
Files					
File List	Globals				
All	Functions	Typedefs	Enumerations	Enumerator	Macros

- BASE\_LENGTH : [base.h](#)
- BASE\_MAX : [base.h](#)
- BASE\_MIN : [base.h](#)
- LARLIB\_VERSION : [base.h](#)
- LOG\_ALL\_CHANNELS : [log.h](#)
- LOG\_ASSERT : [log.h](#)
- TASK\_INFINITE\_TIMEOUT : [task.h](#)

# LAR Library 1.14

Main Page

Related Pages

Modules

Data Structures

Files

## Related Pages

Here is a list of all related documentation pages:

[Todo List](#)

Generated on Mon Mar 27 2017 15:42:53 for LAR Library by  1.8.9.1

# LAR Library 1.14

[Main Page](#)[Related Pages](#)[Modules](#)[Data Structures](#)[Files](#)[File List](#)[Globals](#)

C: &gt; Projects &gt; lalarlib &gt; larlib &gt; common &gt; inc &gt; larlib &gt;

## date.h

Go to the documentation of this file.

```
1 #ifndef LARLIB_DATE_H
2 #define LARLIB_DATE_H
3
25 #include <larlib/base.h>
26
27 /*
28  * Typedef for structures below
29  */
30 typedef struct dateDate_t dateDate_t;
31 typedef struct dateDateTime_t dateDateTime_t;
32
37 enum dateMonths_t {
38     DATE_JAN = 1,
39     DATE_FEB,
40     DATE_MAR,
41     DATE_APR,
42     DATE_MAY,
43     DATE_JUN,
44     DATE_JUL,
45     DATE_AUG,
46     DATE_SEP,
47     DATE_OCT,
48     DATE_NOV,
49     DATE_DEC
```

```
50 };
```

```
51
```

```
55 enum dateWeekdays_t {
```

```
56     DATE_SUN = 0,
```

```
57     DATE_MON,
```

```
58     DATE_TUE,
```

```
59     DATE_WED,
```

```
60     DATE_THU,
```

```
61     DATE_FRI,
```

```
62     DATE_SAT
```

```
63 };
```

```
64
```

```
68 struct dateDate_t {
```

```
69     uint8_t day;
```

```
70     uint8_t mon;
```

```
71     uint16_t year;
```

```
72 };
```

```
73
```

```
77 struct dateTime_t {
```

```
78     dateDate_t date;
```

```
79     uint8_t hour;
```

```
80     uint8_t min;
```

```
81     uint8_t sec;
```

```
82 };
```

```
83
```

```
92 int dateGet(dateDate_t *d);
```

```
93
```

```
102 int dateTimeGet(dateTime_t *dt);
```

```
103
```

```
123 uint64_t dateTimeGetMs(void);
```

```
124
```

```
136 int dateSet(const dateDate_t *d);
```

```
137
```

```
149 int dateTimeSet(const dateTime_t *dt);
```

```
150
```

```
163 int dateToJulianDay(const dateDate_t *d);
```

```
164
```

```
176 int dateFromJulianDay(int jdn, dateDate_t
 *d);
177
189 int dateWeekday(const dateDate_t *d);
190
199 int dateIsLeap(uint16_t year);
200
210 int dateIsValid(const dateDate_t *d);
211
220 int dateTimeIsValid(const dateTime_t *dt);
221
244 int dateDiff(const dateDate_t *a, const
 dateDate_t *b);
245
256 int dateAddDays(dateDate_t *d, int ndays);
257
271 int dateCompare(const dateDate_t *a, const
 dateDate_t *b);
272
284 int dateTimeAddSeconds(dateTime_t *dt, int
 nsecs);
285
298 int dateTimeCompare(const dateTime_t *a,
 const dateTime_t *b);
299
326 int dateTimeDiff(const dateTime_t *a, const
 dateTime_t *b);
327
328 /* @} */
329
330 #endif
```

# LAR Library 1.14

Main Page

Related Pages

Modules

Data Structures

Files

File List

Globals

C: > Projects > lalarlib > larlib > common > inc > larlib >

## format.h

Go to the documentation of this file.

```
1 #ifndef LARLIB_FORMAT_H
2 #define LARLIB_FORMAT_H
3
323 #include <larlib/base.h>
324
325 /*
326  * Typedefs
327 */
328 typedef struct format_t format_t;
329 typedef struct formatField_t formatField_t;
330
335 enum formatDirection_t {
336     FORMAT_PACK,
337     FORMAT_UNPACK
338 };
339
340 typedef enum formatDirection_t
341     formatDirection_t;
345 struct format_t {
347     formatField_t *fields;
348
350     int nfields;
351 }
```

```
364     int (*filter)(const format_t *fmt, const
365                     formatField_t *field);
366
370     void *fieldMap;
371
373     int errorCode;
374
380     formatField_t *errorField;
381 };
382
386 struct formatField_t {
387
392     int index;
393
397     int (*prefix)(uint8_t *buffer, int
398                   limit, formatDirection_t direction, format_t
402                   *fmt, formatField_t *field);
403
407     int (*body)(uint8_t *buffer, int limit,
408                 formatDirection_t direction, format_t *fmt,
412                 formatField_t *field);
413
417     int (*suffix)(uint8_t *buffer, int
418                   limit, formatDirection_t direction, format_t
422                   *fmt, formatField_t *field);
423
427     int size;
428
432     void *data;
433
437     int bodySize;
438 };
439
443 int formatPack(format_t *fmt, uint8_t
447             *output, int maxOutput);
448
452 int formatUnpack(format_t *fmt, uint8_t
```

```
*input, int inputSize);  
449  
473 int formatFilterISO8583(const format_t *fmt,  
    const formatField_t *field);  
474  
490 int formatBodyISO8583(uint8_t *buffer, int  
    limit, formatDirection_t direction, format_t  
    *fmt, formatField_t *field);  
491  
508 int formatBodyPacked(uint8_t *buffer, int  
    limit, formatDirection_t direction, format_t  
    *fmt, formatField_t *field);  
509  
522 int formatBodySkip (uint8_t *buffer, int  
    limit, formatDirection_t direction, format_t  
    *fmt, formatField_t *field);  
523  
536 int formatBodyMemcpy(uint8_t *buffer, int  
    limit, formatDirection_t direction, format_t  
    *fmt, formatField_t *field);  
537  
550 int formatBodyStrcpy(uint8_t *buffer, int  
    limit, formatDirection_t direction, format_t  
    *fmt, formatField_t *field);  
551  
571 int formatBodyUint64Dec(uint8_t *buffer, int  
    limit, formatDirection_t direction, format_t  
    *fmt, formatField_t *field);  
572  
592 int formatBodyUint32Dec(uint8_t *buffer, int  
    limit, formatDirection_t direction, format_t  
    *fmt, formatField_t *field);  
593  
613 int formatBodyUint16Dec(uint8_t *buffer, int  
    limit, formatDirection_t direction, format_t  
    *fmt, formatField_t *field);  
614
```

```
634 int formatBodyUint8Dec(uint8_t *buffer, int
    limit, formatDirection_t direction, format_t
    *fmt, formatField_t *field);
635
651 int formatBodyUint64Buf(uint8_t *buffer, int
    limit, formatDirection_t direction, format_t
    *fmt, formatField_t *field);
652
671 int formatBodyBufHex(uint8_t *buffer, int
    limit, formatDirection_t direction, format_t
    *fmt, formatField_t *field);
672
685 int formatSizeAsc2(uint8_t *buffer, int
    limit, formatDirection_t direction, format_t
    *fmt, formatField_t *field);
686
699 int formatSizeAsc3(uint8_t *buffer, int
    limit, formatDirection_t direction, format_t
    *fmt, formatField_t *field);
700
713 int formatSizeBcd2(uint8_t *buffer, int
    limit, formatDirection_t direction, format_t
    *fmt, formatField_t *field);
714
727 int formatSizeBcd3(uint8_t *buffer, int
    limit, formatDirection_t direction, format_t
    *fmt, formatField_t *field);
728
741 int formatPadSpaces(uint8_t *buffer, int
    limit, formatDirection_t direction, format_t
    *fmt, formatField_t *field);
742
755 int formatPadZeroes(uint8_t *buffer, int
    limit, formatDirection_t direction, format_t
    *fmt, formatField_t *field);
756
769 int formatPadZeroBytes(uint8_t *buffer, int
```

```
    limit, formatDirection_t direction, format_t
    *fmt, formatField_t *field);
770
784 int formatPadSkip(uint8_t *buffer, int
    limit, formatDirection_t direction, format_t
    *fmt, formatField_t *field);
785 /* @} */
786 /* @} */
787
788
789
790 #endif
```

# LAR Library 1.14

[Main Page](#)[Related Pages](#)[Modules](#)[Data Structures](#)[Files](#)[File List](#)[Globals](#)

C: &gt; Projects &gt; lalarlib &gt; larlib &gt; common &gt; inc &gt; larlib &gt;

## ini.h

Go to the documentation of this file.

```
1 #ifndef LARLIB_INIPARSE_H
2 #define LARLIB_INIPARSE_H
3
77 #include <larlib/base.h>
78
79 /*
80  * Typedef for struct iniElement_t
81  */
82 typedef struct iniElement_t iniElement_t;
83
88 struct iniElement_t {
96     char *group;
97
102     char *key;
103
116     int (*handler)(iniElement_t *ini, const
124         char *group, const char *key,
125                         const char *value);
126
123     void *param;
124 };
125
139 typedef char* (*iniLineReader_t)(void *p,
140     char *line, int maxlen);
```

```
140
160 int iniParse(iniElement_t elements[], const
    char *fileName);
161
177 int iniParseString(iniElement_t elements[],
    const char *contents);
178
198 int iniParseLines(iniElement_t elements[],
    iniLineReader_t readLine, void *p);
199
200 /* @} */
201
202 #endif
```

# LAR Library 1.14

Main Page	Related Pages	Modules	Data Structures
Files			
<a href="#">File List</a>	<a href="#">Globals</a>		
C: \ Projects \ lalarlib \ larlib \ common \ inc \ larlib \			

## step.h

Go to the documentation of this file.

```
1 #ifndef LARLIB_STEP_H
2 #define LARLIB_STEP_H
3
102 #include <larlib/base.h>
103
107 typedef struct step_t step_t;
108
109 /*
110  * Forward declaration of a step action
111  */
112 typedef struct stepAction_t stepAction_t;
113
126 typedef int (*stepFunction_t)(step_t *s,
     stepAction_t *a, void *c);
127
131 struct stepAction_t {
132     int label;
133     stepFunction_t fun;
134     void *param;
135 };
136
169 step_t *stepCreate(stepAction_t *actions,
     int nactions);
170
```

```
182 int stepRun(step_t *s, void *c);
183
212 int stepGotoLabel(step_t *s, int label);
213
218 void stepDestroy(step_t *s);
219
220 /* @} */
221
222 #endif
```

---

Generated on Mon Mar 27 2017 15:42:52 for LAR Library by [doxygen](#) 1.8.9.1

# LAR Library 1.14

Main Page	Related Pages	Modules	Data Structures
Files			
File List	Globals		

C: › Projects › lalarlib › larlib › common › inc › larlib ›

## all.h

Go to the documentation of this file.

```
1 #ifndef LARLIB_ALL_H
2 #define LARLIB_ALL_H
3
9 #include <lарlib/base.h>
10 #include <lарlib/bits.h>
11 #include <lарlib/conv.h>
12 #include <lарlib/date.h>
13 #include <lарlib/deque.h>
14 #include <lарlib/format.h>
15 #include <lарlib/fs.h>
16 #include <lарlib/ini.h>
17 #include <lарlib/log.h>
18 #include <lарlib/mem.h>
19 #include <lарlib/step.h>
20 #include <lарlib/tab.h>
21 #include <lарlib/task.h>
22
23 #endif
```

# LAR Library 1.14

[Main Page](#)[Related Pages](#)[Modules](#)[Data Structures](#)[Files](#)[File List](#)[Globals](#)

C: &gt; Projects &gt; lalarlib &gt; larlib &gt; common &gt; inc &gt; larlib &gt;

## base.h

Go to the documentation of this file.

```
1 #ifndef LARLIB_BASE_H
2 #define LARLIB_BASE_H
3
4 #ifdef __TELIUM3__
5 #define HAS_STDINT
6 #endif
7
8 /*
9  * If not available, redefine some C99's
10 stdint types.
11 */
12 #if !defined(HAVE_STDINT) &&
13 !defined(HAS_STDINT)
14
15 typedef char int8_t;
16
17 typedef short int16_t;
18
19 typedef int int32_t;
20
21 typedef long long int64_t;
22
23 typedef unsigned char uint8_t;
24
```

```
165 typedef unsigned short uint16_t;
166
168 typedef unsigned int uint32_t;
169
171 typedef unsigned long long uint64_t;
172
173 #else
174 #include <stdint.h>
175 #endif
176
177 /* ----- */
178
182 enum baseErrors_t {
183     BASE_ERR_OK = 0,
184     BASE_ERR_INVALID_PARAMETER = -1,
185     BASE_ERR_DATA_NOT_FOUND = -2,
186     BASE_ERR_INVALID_HANDLE = -3,
187     BASE_ERR_RESOURCE_PB = -4,
188     BASE_ERR_OVERFLOW = -5,
189     BASE_ERR_ACCESS = -6,
190     BASE_ERR_CANCEL = -7,
191     BASE_ERR_TIMEOUT = -8
192 };
193
194 /* ----- */
195
196 /*
197 * Helpful defines.
198 */
199
203 #define BASE_MAX(a,b) (((a) < (b)) ? (b) :
204 (a))
204
208 #define BASE_MIN(a,b) (((a) < (b)) ? (a) :
209 (b))
```

```
209  
220 #define BASE_LENGTH(v)  (sizeof(v) /  
    sizeof((v)[0]))  
221  
222 /* ----- */  
223  
224 /*  
225 * Library version information  
226 */  
227  
234 #define LARLIB_VERSION      (0x0113)  
235  
251 const char *larlibGetVersion(void);  
252  
253 /* @} */  
254  
255 #endif
```

# LAR Library 1.14

[Main Page](#)[Related Pages](#)[Modules](#)[Data Structures](#)[Files](#)[File List](#)[Globals](#)

C: › Projects › lalarlib › larlib › common › inc › larlib ›

## bits.h

Go to the documentation of this file.

```
1 #ifndef LARLIB_BITS_H
2 #define LARLIB_BITS_H
3
39 #include <larlib/base.h>
40
46 typedef uint8_t bitsBuffer_t;
47
51 enum bitsBltOp_t {
52     BITS_BLT_COPY,
53     BITS_BLT_NOT,
54     BITS_BLT_AND,
55     BITS_BLT_OR,
56     BITS_BLT_XOR,
57     BITS_BLT_NOT_AND
58 };
59
74 int bitsSet(bitsBuffer_t *b, int index, int
    value);
75
89 int bitsSetRange(bitsBuffer_t *b, int
    offset, int count, int value);
90
104 int bitsGet(const bitsBuffer_t *b, int
    index);
```

```
105
126 int64_t bitsExtractInt(const bitsBuffer_t
   *b, int offset, int count);
127
149 int bitsAppend(const bitsBuffer_t *src, int
   srcOffset, int srcCount,
150               bitsBuffer_t *dst, int
   dstCount);
151
175 int bitsAppendInt(uint64_t bits, int nbits,
176                      bitsBuffer_t *dst, int
   dstCount);
177
198 int bitsRotateLeft(bitsBuffer_t *b, int
   index, int nbits, int nrotate);
199
220 int bitsRotateRight(bitsBuffer_t *b, int
   index, int nbits, int nrotate);
221
241 int bitsShiftLeft(bitsBuffer_t *b, int
   index, int nbits, int nshift);
242
262 int bitsShiftRight(bitsBuffer_t *b, int
   index, int nbits, int nshift);
263
282 int bitsCopy(const bitsBuffer_t *src, int
   srcOffset, int srcCount,
283               bitsBuffer_t *dst, int
   dstOffset);
284
306 int bitsBlt(const bitsBuffer_t *src, int
   srcOffset, int srcCount,
307               enum bitsBltOp_t op,
308               bitsBuffer_t *dst, int
   dstOffset);
309
310 /* @} */
```

```
311  
312 #endif
```

# LAR Library 1.14

Main Page

Related Pages

Modules

Data Structures

Files

File List

Globals

C: > Projects > lalarlib > larlib > common > inc > larlib >

## conv.h

Go to the documentation of this file.

```
1 #ifndef LARLIB_CONV_H
2 #define LARLIB_CONV_H
3
68 #include <larlib/base.h>
69
73 enum convErrors_t {
74     CONV_ERR_INVALID_BASE = -101
75 };
76
95 int convTxtToInt(const char *txt, int cnt,
    int base, uint64_t *n);
96
115 int convTxtToInt8(const char *txt, int cnt,
    int base, uint8_t *n);
116
135 int convTxtToInt16(const char *txt, int cnt,
    int base, uint16_t *n);
136
155 int convTxtToInt32(const char *txt, int cnt,
    int base, uint32_t *n);
156
175 int convTxtToInt64(const char *txt, int cnt,
    int base, uint64_t *n);
176
```

```
200 int convIntToTxt(uint64_t n, char *txt, int
    maxcnt, int base);
201
223 int convIntToTxtPad(uint64_t n, char *txt,
    int cnt, int base);
224
239 int convBufToHex(const uint8_t *buf, int
    nbuf, char *hex, int maxcnt);
240
268 int convHexToBuf(const char *hex, int cnt,
    uint8_t *buf, int maxbuf);
269
288 int convIntToBcd(uint64_t n, uint8_t *bcd,
    int maxcnt, int padRight);
289
321 int convIntToBcdPad(uint64_t n, uint8_t
    *bcd, int nibbles);
322
335 int convBcdToInt(const uint8_t *bcd, int
    cnt, uint64_t *n);
336
357 int convTxtToBcd(const char *txt, int
    maxtxt, uint8_t *bcd, int maxbcd);
358
381 int convBcdToTxt(const uint8_t *bcd, int
    maxbcd, char *txt, int maxtxt);
382
401 int convIntToBuf(uint64_t n, uint8_t *buf,
    int maxbuf);
402
424 int convIntToBufPad(uint64_t n, uint8_t
    *buf, int maxbuf);
425
436 uint64_t convBufToInt(const uint8_t *buf,
    int nbuf);
437
461 int convBase64ToBuf(const char *b64, int
```

```
    cnt, uint8_t *buf, int maxbuf);
462
479 int convBufToBase64(const uint8_t *buf, int
480     nbuf, char *b64, int maxb64);
481 /* @} */
482
483 #endif
484
```

# LAR Library 1.14

[Main Page](#)[Related Pages](#)[Modules](#)[Data Structures](#)[Files](#)[File List](#)[Globals](#)

C: › Projects › lalarlib › larlib › common › inc › larlib ›

## deque.h

Go to the documentation of this file.

```
1 #ifndef LARLIB_DEQUE_H
2 #define LARLIB_DEQUE_H
3
23 #include <larlib/base.h>
24
29 typedef struct deque_t deque_t;
30
44 deque_t *dequeCreate(int sizeHint);
45
51 void dequeDestroy(deque_t *d);
52
69 int dequePushBack(deque_t *d, void *p);
70
80 int dequePopBack(deque_t *d);
81
98 int dequePushFront(deque_t *d, void *p);
99
109 int dequePopFront(deque_t *d);
110
124 int dequeLength(const deque_t *d);
125
144 void *dequeGet(const deque_t *d, int i);
145
159 void *dequeGetFront(const deque_t *d);
```

```
160
175 void *dequeGetBack(const deque_t *d);
176
194 void *dequeSet(deque_t *d, int i, void *p);
195
218 int dequeInsert(deque_t *d, int i, void *p);
219
234 int dequeRemove(deque_t *d, int i);
235
236 /* @} */
237
238 #endif
```

# LAR Library 1.14

[Main Page](#)[Related Pages](#)[Modules](#)[Data Structures](#)[Files](#)[File List](#)[Globals](#)

C: › Projects › lalarlib › larlib › common › inc › larlib ›

## fs.h

Go to the documentation of this file.

```
1 #ifndef LARLIB_FS_H
2 #define LARLIB_FS_H
3
60 #include <larlib/base.h>
61
65 typedef struct fsFile_t fsFile_t;
66
75 enum fsOpenFlags_t {
76     FS_OPEN_READ = 1,
77     FS_OPEN_WRITE = 2,
78     FS_OPEN_CREATE = 4,
79     FS_OPEN_RESET = 8,
80     FS_OPEN_APPEND = 16
81 };
82
86 enum fsSeekWhence_t {
87     FS_WHENCE_SET,
88     FS_WHENCE_END,
89     FS_WHENCE_CUR
90 };
91
105 int fsSetFileNamePrefix(const char *prefix);
106
120 int fsPushFileNamePrefix(const char
```

```
*prefix);  
121 int fsPopFileNamePrefix(void);  
132  
142 const char *fsGetFileNamePrefix(void);  
143  
174 int fsOpen(const char *name, int flags,  
    fsFile_t **file);  
175  
186 int fsClose(fsFile_t *f);  
187  
206 int fsRead(fsFile_t *f, int n, void *p);  
207  
221 int fs.ReadByte(fsFile_t *f);  
222  
255 int fsReadUntil(fsFile_t *f, const uint8_t  
    *delims, int ndelim, int n, void *p);  
256  
274 int fsWrite(fsFile_t *f, int n, const void  
    *p);  
275  
290 int fsWriteByte(fsFile_t *f, int b);  
291  
304 int fsTell(fsFile_t *f);  
305  
328 int fsSeek(fsFile_t *f, int whence, int  
    offset);  
329  
351 int fsLength(fsFile_t *f);  
352  
365 int fsRemove(const char *name);  
366  
382 int fsRename(const char *oldname, const char  
    *newname);  
383  
395 int fsExist(const char *name);  
396
```

```
397 /* @} */  
398  
399 #endif
```

---

Generated on Mon Mar 27 2017 15:42:52 for LAR Library by [doxygen](#) 1.8.9.1

# LAR Library 1.14

[Main Page](#)[Related Pages](#)[Modules](#)[Data Structures](#)[Files](#)[File List](#)[Globals](#)

C: › Projects › lalarlib › larlib › common › inc › larlib ›

## log.h

Go to the documentation of this file.

```
1 #ifndef LARLIB_LOG_H
2 #define LARLIB_LOG_H
3
112 #include <larlib/base.h>
113 #include <stdarg.h>
114
118 enum LogLevel_t {
119     LOG_DEBUG      = 0x0000,
120     LOG_INFO       = 0x2000,
121     LOG_WARNING    = 0x4000,
122     LOG_ERROR      = 0x6000,
123     LOG_CRITICAL   = 0x8000,
124     LOG_ALWAYS     = 0xFF00
125 };
126
131 enum logReservedChannels_t {
132     LOG_CH_BASE = 200,
133     LOG_CH_CONV,
134     LOG_CH_DATE,
135     LOG_CH_FS,
136     LOG_CH_INI,
137     LOG_CH_MEM,
138     LOG_CH_TAB,
139     LOG_CH_STEP,
```

```
140     LOG_CH_FORMAT,
141     LOG_CH_TASK
142 };
143
148 #define LOG_ALL_CHANNELS 0, 256
149
162 typedef int (*logWriteFunction_t)(void
    *context, uint16_t channelLevel,
163                                     const char
    *msg, int msgLength);
164
179 typedef int (*logDumpFunction_t)
    (logWriteFunction_t write,
180      void
    *context, uint16_t channelLevel,
181      const void
    *data, int dataLen);
182
191 #ifndef NDEBUG
192 #define LOG_ASSERT(cond) ((void) (!!!(cond))
    || (logAssertImpl(#cond, __LINE__, __FILE__),
    0)))
193 #else
194 #define LOG_ASSERT(cond)
195 #endif
196
197 /*
198  * Implementation detail -- this function
199  * prints the assertion message
200  * \p condStr plus the line number \p line
201  * and the file name \p file
202  * and resets the device.
203  */
204 void logAssertImpl(const char *condStr, int
    line, const char *file);
205
206 int logWriteTeliumTrace(void *context,
```

```
    uint16_t channelLevel, const char *msg, int
    msgLength);
234
252 int logWriteTeliumRemoteDebugger(void
    *context, uint16_t channelLevel, const char
    *msg, int msgLength);
253
270 int logDumpFormattedAscii(logWriteFunction_t
    write,
271                                     void *context,
    uint16_t channelLevel,
272                                     const void *data,
    int dataLen);
273
323 int logSetChannels(uint8_t firstChannel, int
    numChannels, uint16_t level,
324                                     logWriteFunction_t
    writeFunction,
325                                     logDumpFunction_t
    dumpFormatFunction,
326                                     void *context);
327
343 int logChannelIsEnabled(uint16_t
    channelLevel);
344
374 int logPrintf(uint16_t channelLevel, const
    char *fmt, ...);
375
390 int logPrintvf(uint16_t channelLevel, const
    char *fmt, va_list va);
391
406 int logDump(uint16_t channelLevel, const
    void *buffer, int size);
407
408 /* @} */
409
410 #endif
```



# LAR Library 1.14

Main Page

Related Pages

Modules

Data Structures

Files

File List

Globals

C: › Projects › lalarlib › larlib › common › inc › larlib ›

## mem.h

Go to the documentation of this file.

```
1 #ifndef LARLIB_MEM_H
2 #define LARLIB_MEM_H
3
26 #include <larlib/base.h>
27
40 void *memAlloc(int nb);
41
53 void *memAllocZero(int nb);
54
70 void *memRealloc(void *oldp, int nb);
71
77 void memFree(void *p);
78
79 /* @} */
80
81#endif
```

# LAR Library 1.14

[Main Page](#)[Related Pages](#)[Modules](#)[Data Structures](#)[Files](#)[File List](#)[Globals](#)

C: › Projects › lalarlib › larlib › common › inc › larlib ›

## tab.h

Go to the documentation of this file.

```
1 #ifndef LARLIB_TAB_H
2 #define LARLIB_TAB_H
3
69 #include <larlib/base.h>
70
74 typedef struct table_t table_t;
75
79 enum tabErrors_t {
80     TAB_ERR_DELETED = 1
81 };
82
94 int tabOpen(const char *fname, table_t **t);
95
110 int tabCreate(const char *fname, int
recSize, table_t **t);
111
117 void tabClose(table_t *t);
118
134 int tabFlush(table_t *t);
135
144 int tabRecSize(table_t *t);
145
159 int tabGetStamp(table_t *t, uint32_t
*stamp);
```

```
160
172 int tabSetStamp(table_t *t, uint32_t stamp);
173
206 int tabGet(table_t *t, int recn, void
 *recp);
207
231 int tabPut(table_t *t, int recn, const void
 *recp);
232
248 int tabAppend(table_t *t, const void *recp);
249
261 int tabSize(table_t *t);
262
273 int tabCount(table_t *t);
274
296 int tabDelete(table_t *t, int recn);
297
314 int tabUnDelete(table_t *t, int recn);
315
334 int tabPack(const char *fname);
335
356 int tabGoFirst(table_t *t);
357
370 int tabGetNext(table_t *t, void *recp);
371
384 int tabGetPrev(table_t *t, void *recp);
385
405 int tabGoLast(table_t *t);
406
432 int tabFindNext(table_t *t, void *recp,
 const void *keyp,
433 int (*cmpFn)(const void
 *recp, const void *keyp));
434
458 int tabFind(table_t *t, void *recp, const
 void *keyp,
459 int (*cmpFn)(const void *recp,
```

```
    const void *keyp));  
460| /* @} */  
462|  
463#endif
```

---

Generated on Mon Mar 27 2017 15:42:52 for LAR Library by [doxygen](#) 1.8.9.1

# LAR Library 1.14

[Main Page](#)[Related Pages](#)[Modules](#)[Data Structures](#)[Files](#)[File List](#)[Globals](#)

C: &gt; Projects &gt; lalarlib &gt; larlib &gt; common &gt; inc &gt; larlib &gt;

## task.h

Go to the documentation of this file.

```
1 #ifndef LARLIB_TASK_H
2 #define LARLIB_TASK_H
3
167 #include <larlib/base.h>
168
170 #define TASK_INFINITE_TIMEOUT (0xFFFFFFFFLU)
171
175 typedef struct taskThread_t taskThread_t;
176
181 typedef void (*taskThreadFunction_t)(void
 *param);
182
186 typedef struct taskQueue_t taskQueue_t;
187
191 typedef struct taskSemaphore_t
 taskSemaphore_t;
192
196 typedef struct taskEvent_t taskEvent_t;
197
210 taskThread_t
 *taskThreadCreate(taskThreadFunction_t fn,
 void *param, uint32_t stackSize);
211
220 void taskSleep(uint32_t timeout);
```

```
221
226 taskThread_t *taskThreadCurrent(void);
227
233 taskQueue_t *taskQueueCreate(int maxItems);
234
244 void taskQueueDestroy(taskQueue_t *q);
245
264 int taskQueuePost(taskQueue_t *q, void
 *item);
265
288 int taskQueuePop(taskQueue_t *q, void
 **item, uint32_t timeout);
289
298 taskSemaphore_t *taskSemaphoreCreate(int
 limit);
299
308 void taskSemaphoreDestroy(taskSemaphore_t
 *s);
309
323 int taskSemaphoreAcquire(taskSemaphore_t *s,
 uint32_t timeout);
324
333 int taskSemaphoreRelease(taskSemaphore_t
 *s);
334
344 taskEvent_t *taskEventCreate(void);
345
375 taskEvent_t *taskEventCreateSystem(uint32_t
 bitmask);
376
383 void taskEventDestroy(taskEvent_t *e);
384
398 int taskEventSignal(taskEvent_t *e);
399
414 int taskEventCheck(taskEvent_t *e);
415
431 int taskEventWait(taskEvent_t *e, uint32_t
```

```
    timeout);
432
458 int taskEventWaitAny(taskEvent_t *events[],
459     int nevents, uint32_t timeout);
460
475 int taskEventClear(taskEvent_t *e);
476
477 /* @} */
478
479 #endif
```

# LAR Library 1.14

Main Page	Related Pages	Modules	Data Structures
Files			
C: >			

## C: Directory Reference

# Directories

---

directory **Projects**

---

Generated on Mon Mar 27 2017 15:42:53 for LAR Library by **doxygen** 1.8.9.1

# LAR Library 1.14

Main Page	Related Pages	Modules	Data Structures
Files			
C: > Projects >			

## Projects Directory Reference

## Directories

---

directory **lalarlib**

---

Generated on Mon Mar 27 2017 15:42:53 for LAR Library by **[doxygen](#)** 1.8.9.1

# LAR Library 1.14

Main Page	Related Pages	Modules	Data Structures
Files			
C: > Projects > lalarlib >			

## **lalarlib Directory Reference**

## Directories

---

directory **larlib**

---

Generated on Mon Mar 27 2017 15:42:53 for LAR Library by **doxygen** 1.8.9.1

# LAR Library 1.14

Main Page	Related Pages	Modules	Data Structures
Files			
C: > Projects > lalarlib > larlib >			

## larlib Directory Reference

# Directories

---

directory **common**

---

Generated on Mon Mar 27 2017 15:42:53 for LAR Library by **doxygen** 1.8.9.1

# LAR Library 1.14

Main Page	Related Pages	Modules	Data Structures
Files			
C: > Projects > lalarlib > larlib > common >			

## common Directory Reference

# Directories

---

directory **inc**

---

Generated on Mon Mar 27 2017 15:42:53 for LAR Library by **doxygen** 1.8.9.1

# LAR Library 1.14

Main Page	Related Pages	Modules	Data Structures
Files			
C: > Projects > lalarlib > larlib > common > inc >			

## inc Directory Reference

## Directories

---

directory **larlib**

---

Generated on Mon Mar 27 2017 15:42:53 for LAR Library by **doxygen** 1.8.9.1

# LAR Library 1.14

Main Page	Related Pages	Modules	Data Structures
Files			
C: > Projects > lalarlib > larlib > common > inc > larlib >			

## larlib Directory Reference

## Files

file **all.h [code]**

Force inclusion of all LarLib modules.

file **base.h [code]**

Larlib basic definitions.

file **bits.h [code]**

Bit manipulation.

file **conv.h [code]**

Conversion routines.

file **date.h [code]**

Date and time.

file **deque.h [code]**

A double-ended queue of `void*` elements.

file **format.h [code]**

Buffer formatting.

file **fs.h [code]**

File System abstraction.

file **ini.h [code]**

Read/parsing of MS-like INI files.

file **log.h [code]**

General Logging System.

file **mem.h [code]**

Memory Management.

file [\*\*step.h\*\*](#) [code]

State-machine for transaction processing.

file [\*\*tab.h\*\*](#) [code]

Table management.

file [\*\*task.h\*\*](#) [code]

Multi-threading.