



User's Manual

V1.36.02

Micrium
For the Way Engineers Work

Micrium
1290 Weston Road, Suite 306
Weston, FL 33326
USA

www.Micrium.com

Designations used by companies to distinguish their products are often claimed as trademarks. In all instances where Micrium Press is aware of a trademark claim, the product name appears in initial capital letters, in all capital letters, or in accordance with the vendor's capitalization preference. Readers should contact the appropriate companies for more complete information on trademarks and trademark registrations. All trademarks and registered trademarks in this book are the property of their respective holders.

Copyright © 2011 by Micrium except where noted otherwise. All rights reserved. Printed in the United States of America. No part of this publication may be reproduced or distributed in any form or by any means, or stored in a database or retrieval system, without the prior written permission of the publisher; with the exception that the program listings may be entered, stored, and executed in a computer system, but they may not be reproduced for publication.

The programs and code examples in this book are presented for instructional value. The programs and examples have been carefully tested, but are not guaranteed to any particular purpose. The publisher does not offer any warranties and does not guarantee the accuracy, adequacy, or completeness of any information herein and is not responsible for any errors and omissions. The publisher assumes no liability for damages resulting from the use of the information in this book or for any infringement of the intellectual property rights of third parties that would result from the use of this information.

USER'S MANUAL VERSIONS

If you find any errors in this document, please inform us and we will make the appropriate corrections for future releases.

Manual Version	Date	By	Description
V1.18	2005/08/30	ITJ	Manual Created
V1.19	2006/04/25	JJL	Updated Manual
V1.20	2006/06/01	ITJ	Updated Manual
V1.21	2006/08/09	ITJ	Added memory macros
V1.22	2006/09/20	ITJ	Updated Manual
V1.23	2007/02/27	SR	Updated Manual
V1.24	2007/05/04	ITJ	Updated Manual
V1.25	2008/06/20	ITJ	Added memory management Added ASCII module
V1.26	2008/12/01	ITJ	Added heap memory Added string parse functions
V1.27	2009/01/05	ITJ	Updated Manual
V1.28	2009/02/26	EHS	Updated Manual
V1.29	2009/04/28	ITJ	Updated Manual
V1.30	2009/05/05	BAN	Added math module with pseudo-random number generator
V1.31	2009/09/28	ITJ	Updated Manual
V1.32	2010/04/02	ITJ	Updated Manual
V1.33	2010/07/28	ITJ	Updated Manual
V1.33	2010/09/03	ITJ	Converted document from Word to FrameMaker
V1.34	2010/12/17	ITJ	Updated Manual
V1.35.00	2011/06/13	ITJ	Added endian memory conversion macros
V1.36.00	2011/08/15	ITJ	Added memory remaining functions

Manual Version	Date	By	Description
V1.36.02	2012/02/21	FBJ	Added access to memory pool blocks by index functions

Table of Contents

Chapter 1	Introduction	9
1-1	Portable	9
1-2	Scalable	9
1-3	Coding Standards	9
1-4	MISRA C	10
1-5	Safety Critical Certification	10
1-6	µC/LIB Limitations	10
Chapter 2	Directories and Files	11
Chapter 3	µC/LIB Constant and Macro Library	13
3-1	Library Constants	13
3-1-1	Boolean Constants	13
3-1-2	Bit Constants	13
3-1-3	Octet Constants	13
3-1-4	Number Base Constants	14
3-1-5	Integer Constants	14
3-1-6	Time Constants	14
3-2	Common Library Macros	15
3-2-1	DEF_BITxx()	15
3-2-2	DEF_BIT_MASK_xx()	17
3-2-3	DEF_BIT_FIELD_xx()	19
3-2-4	DEF_BIT_SET_xx()	21
3-2-5	DEF_BIT_CLR_xx()	23
3-2-6	DEF_BIT_IS_SET()	25
3-2-7	DEF_BIT_IS_CLR()	27
3-2-8	DEF_BIT_IS_SET_ANY()	29
3-2-9	DEF_BIT_IS_CLR_ANY()	31
3-2-10	DEF_CHK_VAL_MIN()	33

3-2-11	DEF_CHK_VAL_MAX()	35
3-2-12	DEF_CHK_VAL()	37
3-2-13	DEF_GET_U_MAX_VAL()	39
3-2-14	DEF_MIN()	40
3-2-15	DEF_MAX()	42
3-2-16	DEF_ABS()	44
 Chapter 4	 µC/LIB Memory Library	 45
4-1	Memory Library Configuration	45
4-2	Memory Library Macros	46
4-2-1	MEM_VAL_BIG_TO_LITTLE_xx() / MEM_VAL_LITTLE_TO_BIG_xx() .	46
4-2-2	MEM_VAL_BIG_TO_HOST_xx() / MEM_VAL_HOST_TO_BIG_xx()	48
4-2-3	MEM_VAL_LITTLE_TO_HOST_xx() / MEM_VAL_HOST_TO_LITTLE_xx()	50
4-2-4	MEM_VAL_GET_xxx()	52
4-2-5	MEM_VAL_SET_xxx()	54
4-2-6	MEM_VAL_COPY_GET_xxx()	56
4-2-7	MEM_VAL_COPY_SET_xxx()	59
4-2-8	MEM_VAL_COPY_xxx()	62
4-3	Memory Library Functions	64
4-3-1	Mem_Clr()	64
4-3-2	Mem_Set()	65
4-3-3	Mem_Copy()	66
4-3-4	Mem_Cmp()	68
4-4	Memory Allocation Functions	70
4-4-1	Mem_Init()	71
4-4-2	Mem_HeapAlloc()	72
4-4-3	Mem_HeapGetSizeRem()	74
4-4-4	Mem_SegGetSizeRem()	76
4-4-5	Mem_PoolClr()	78
4-4-6	Mem_PoolCreate()	80
4-4-7	Mem_PoolBlkGetNbrAvail()	83
4-4-8	Mem_PoolBlkGet()	85
4-4-9	Mem_PoolBlkGetUsedAtIx()	87
4-4-10	Mem_PoolBlkFree()	89
4-4-11	Mem_PoolBlkIxGet()	91
4-5	Memory Library Optimization	93

Chapter 5	µC/LIB String Library	94
5-1	String Library Configuration	94
5-2	String Library Functions	95
5-2-1	Str_Len()	95
5-2-2	Str_Len_N()	96
5-2-3	Str_Copy()	98
5-2-4	Str_Copy_N()	100
5-2-5	Str_Cat()	102
5-2-6	Str_Cat_N()	104
5-2-7	Str_Cmp()	106
5-2-8	Str_Cmp_N()	108
5-2-9	Str_CmplgnoreCase()	110
5-2-10	Str_CmplgnoreCase_N()	112
5-2-11	Str_Char()	114
5-2-12	Str_Char_N()	116
5-2-13	Str_Char_Last()	118
5-2-14	Str_Char_Last_N()	120
5-2-15	Str_Str()	122
5-2-16	Str_Str_N()	124
5-2-17	Str_FmtNbr_Int32U()	126
5-2-18	Str_FmtNbr_Int32S()	130
5-2-19	Str_FmtNbr_32()	136
5-2-20	Str_ParseNbr_Int32U()	142
5-2-21	Str_ParseNbr_Int32S()	146
Chapter 6	µC/LIB ASCII Library	150
6-1	Character Value Constants	150
6-2	ASCII Library Macros and Functions	151
6-2-1	ASCII_IS_ALPHA() / ASCII_IsAlpha()	151
6-2-2	ASCII_IS_ALPHA_NUM() / ASCII_IsAlphaNum()	153
6-2-3	ASCII_IS_LOWER() / ASCII_IsLower()	155
6-2-4	ASCII_IS_UPPER() / ASCII_IsUpper()	157
6-2-5	ASCII_IS_DIG() / ASCII_IsDig()	159
6-2-6	ASCII_IS_DIG_OCT() / ASCII_IsDigOct()	161
6-2-7	ASCII_IS_DIG_HEX() / ASCII_IsDigHex()	162
6-2-8	ASCII_IS_BLANK() / ASCII_IsBlank()	164
6-2-9	ASCII_IS_SPACE() / ASCII_IsSpace()	166
6-2-10	ASCII_IS_PRINT() / ASCII_IsPrint()	168

6-2-11	ASCII_IS_GRAPH() / ASCII_IsGraph()	170
6-2-12	ASCII_IS_PUNCT() / ASCII_IsPunct()	172
6-2-13	ASCII_IS_CTRL() / ASCII_IsCtrl()	174
6-2-14	ASCII_TO_LOWER() / ASCII_ToLower()	176
6-2-15	ASCII_TO_UPPER() / ASCII_ToUpper()	178
6-2-16	ASCII_Cmp()	180
Chapter 7	µC/LIB Mathematics Library	182
7-1	Mathematics Library Functions	183
7-1-1	Math_Init()	183
7-1-2	Math_RandSetSeed()	184
7-1-3	Math_Rand()	185
7-1-4	Math_RandSeed()	186
Appendix A	µC/LIB Licensing Policy	188

Chapter

1

Introduction

Designed with Micrium's renowned quality, scalability and reliability, the purpose of μ C/LIB is to provide a clean, organized ANSI C implementation of the most common standard library functions, macros, and constants.

1-1 PORTABLE

μ C/LIB was designed for the vast variety of embedded applications. The source code for μ C/LIB is designed to be independent of and used with any processor (CPU) and compiler.

1-2 SCALABLE

The memory footprint of μ C/LIB can be adjusted at compile time based on the features you need and the desired level of run-time performance.

1-3 CODING STANDARDS

Coding standards have been established early in the design of μ C/LIB and include:

- C coding style
- Naming convention for `#define` constants, macros, variables and functions
- Commenting
- Directory structure

1-4 MISRA C

The source code for μ C/LIB follows the Motor Industry Software Reliability Association (MISRA) C Coding Standards. These standards were created by MISRA to improve the reliability and predictability of C programs in critical automotive systems. Members of the MISRA consortium include Delco Electronics, Ford Motor Company, Jaguar Cars Ltd., Lotus Engineering, Lucas Electronics, Rolls-Royce, Rover Group Ltd., and other firms and universities dedicated to improving safety and reliability in automotive electronics. Full details of this standard can be obtained directly from the MISRA web site, <http://www.misra.org.uk>.

1-5 SAFETY CRITICAL CERTIFICATION

μ C/LIB has been designed and implemented with safety critical certification in mind. μ C/LIB is intended for use in any high-reliability, safety-critical systems including avionics RTCA DO-178B and EUROCAE ED-12B, medical FDA 510(k), IEC 61508 industrial control systems, and EN-50128 rail transportation and nuclear systems.

For example, the FAA (Federal Aviation Administration) requires that all the source code for an application be available in source form and conforming to specific software standards in order to be certified for avionics systems. Since most standard library functions are provided by compiler vendors in uncertifiable binary format, μ C/LIB provides its library functions in certifiable source-code format.

If your product is not safety critical, you should view the software and safety-critical standards as proof that μ C/LIB is a very robust and highly-reliable software module.

1-6 μ C/LIB LIMITATIONS

By design, we have limited some of the feature of μ C/LIB:

- Does not support variable argument library functions

Directories and Files

The distribution of μ C/LIB is typically included in a ZIP file called: **Micrium_uC-LIB-Vxxy.zip**. (Note: The ZIP file name might also include customer names, invoice numbers, and file creation date.) The ZIP file contains all the source code and documentation for μ C/LIB organized in a directory structure according to “AN-2002, μ C/OS-II Directory Structure.” Specifically, the files may be found in the following directories:

\Micrium\Software\uC-LIB

This is the main directory for μ C/LIB and contains source code for many standard library functions, macros, and constants including:

lib_def.h

This file defines constants for many common values such as **TRUE/FALSE**, **YES/NO**, **ENABLED/DISABLED**; as well as for integer, octet, and bit values. However, all **#defines** in this file start are prefixed with **DEF_** — **DEF_TRUE/DEF_FALSE**, **DEF_YES/DEF_NO**, **DEF_ENABLED/DEF_DISABLED**, etc. This file also contains macros for common mathematical operations like **min()/max()**, **abs()**, **bit_set()/bit_clr()**, etc. See Chapter 3, “ μ C/LIB Constant and Macro Library” on page 13 for more details.

lib_mem.c and lib_mem.h

These files contain source code to replace standard library functions **memclr()**, **memset()**, **memcpy()**, **memcmp()**, etc. with μ C/LIB equivalents **Mem_Clr()**, **Mem_Set()**, **Mem_Copy()**, and **Mem_Cmp()**, respectively. See Chapter 4, “ μ C/LIB Memory Library” on page 45 for more details.

lib_str.c and lib_str.h

These files contain source code to replace standard library functions **strlen()**, **strcpy()**, **strcmp()**, etc. with μ C/LIB equivalents **Str_Len()**, **Str_Copy()**, and **Str_Cmp()**, respectively. See Chapter 5, “ μ C/LIB String Library” on page 94 for more details.

lib_ascii.c and lib_ascii.h

These files contain source code to replace standard library functions `tolower()`, `toupper()`, `isalpha()`, `isdigit()`, etc. with μ C/LIB equivalents `ASCII_ToLower()`, `ASCII_ToUpper()`, `ASCII_IsAlpha()`, and `ASCII_IsDig()`, respectively. See Chapter 6, “ μ C/LIB ASCII Library” on page 150 for more details.

lib_math.c and lib_math.h

These files contain source code to replace standard library functions `rand()`, `srand()`, etc. with μ C/LIB equivalents `Math_Rand()`, `Math_RandSetSeed()`, respectively. See Chapter 7, “ μ C/LIB Mathematics Library” on page 182 for more details.

\Micrium\Software\uC-LIB\Doc

This directory contains all μ C/LIB documentation files.

\Micrium\Software\uC-LIB\Cfg\Template

This directory contains a template file, `lib_cfg.h`, which includes configuration for μ C/CPU features such as memory allocation, assembly optimization, and floating point support. If not specified, all μ C/LIB features are configured by default to be disabled. However, you should include the configuration from the template configuration file into your application’s `app_cfg.h` with application-specific configuration settings. See section 4-1 “Memory Library Configuration” on page 45 and section 5-1 “String Library Configuration” on page 94 for more details.

\Micrium\Software\Ports\<CPU Type>\<Compiler>

μ C/CPU also contains additional sub-directories specific to each processor/compiler combination which may include assembly-optimized files:

lib_mem_a.asm or lib_mem_a.s

These (optional) files contain assembly code to optimize certain memory library functions.

Application files which intend to make use of μ C/LIB constants, macros, or functions should `#include` the desired μ C/LIB header files. In addition, applications should configure μ C/LIB features in application-specific configuration file, `app_cfg.h`, which `#define`’s certain constants used to configure Micrium products and/or non-Micrium-related application files.

µC/LIB Constant and Macro Library

µC/CPU contains many standard constants and macros. Common constants include Boolean, bit-mask, and integer values; common macros include bit-level, minimum, maximum, and absolute value operations. All µC/LIB constants and macros are prefixed with `DEF_` to provide a consistent naming convention and to avoid namespace conflicts with other constants and macros in your application. These constants and macros are defined in `lib_def.h`.

3-1 LIBRARY CONSTANTS

3-1-1 BOOLEAN CONSTANTS

µC/LIB contains many Boolean constants such as `DEF_TRUE/DEF_FALSE`, `DEF_YES/DEF_NO`, `DEF_ON/DEF_OFF`, `DEF_ENABLED/DEF_DISABLED`, etc. These constants should be used to configure, assign, and test Boolean values or variables.

3-1-2 BIT CONSTANTS

µC/LIB contains bit constants such as `DEF_BIT_00`, `DEF_BIT_07`, `DEF_BIT_15`, etc.; which define values corresponding to specific bit positions. Currently, µC/LIB supports bit constants up to 64-bits (`DEF_BIT_63`). These constants should be used to configure, assign, and test appropriately-sized bit-field or integer values or variables.

3-1-3 OCTET CONSTANTS

µC/LIB contains octet constants such as `DEF_OCTET_NBR_BITS` and `DEF_OCTET_MASK` which define octet or octet-related values. These constants should be used to configure, assign, and test appropriately-sized, octet-related integer values or variables.

3-1-4 NUMBER BASE CONSTANTS

μC/LIB contains number base constants such as `DEF_NBR_BASE_BIN` and `DEF_NBR_BASE_HEX` which define number base values. These constants should be used to configure, assign, and test number base values or variables.

3-1-5 INTEGER CONSTANTS

μC/LIB contains octet constants such as `DEF_INT_08_MASK`, `DEF_INT_16U_MAX_VAL`, and `DEF_INT_32S_MIN_VAL` which define integer-related values. These constants should be used to configure, assign, and test appropriately-sized, octet-related integer values or variables.

3-1-6 TIME CONSTANTS

μC/LIB contains time constants such as `DEF_TIME_NBR_HR_PER_DAY`, `DEF_TIME_NBR_SEC_PER_MIN`, `DEF_TIME_NBR_mS_PER_SEC`, etc.; which define time or time-related values. These constants should be used to configure, assign, and test time-related values or variables.

3-2 COMMON LIBRARY MACROS

μC/LIB contains many common bit and arithmetic macros. Bit macros modify or test values based on bit masks. Arithmetic macros perform simple mathematical operations or tests.

3-2-1 DEF_BITxx()

Creates a bit mask based on a single bit-number position.

FILES

`lib_def.h`

PROTOTYPES

```
DEF_BIT(bit);  
  
DEF_BIT08(bit);  
DEF_BIT16(bit);  
DEF_BIT32(bit);  
DEF_BIT64(bit);
```

ARGUMENTS

bit Bit number of the bit mask to set.

RETURNED VALUE

Bit mask with the single **bit** number position set.

REQUIRED CONFIGURATION

None.

NOTES / WARNINGS

bit should be a non-negative integer.

bit values that overflow the target CPU and/or compiler environment (e.g. negative or greater-than-CPU-data-size values) may generate compiler warnings and/or errors.

To avoid overflowing any target CPU and/or compiler's integer data type, unsigned bit constant **1** is either cast to specified integer data type size or suffixed with **long** integer modifier, **'L'**. This may still be insufficient for CPUs and/or compilers that support **long long** integer data types, in which case **'LL'** integer modifier should be suffixed. However, since almost all 16- and 32-bit CPUs and compilers support **long** integer data types but many may not support **long long** integer data types, only **long** integer data types and modifiers are supported.

EXAMPLE USAGE

```
CPU_INT16U  mask_16;
CPU_INT32U  mask_32;

mask_16 = DEF_BIT(12u);
mask_16 = DEF_BIT16(15u);
mask_32 = DEF_BIT(19u);
mask_32 = DEF_BIT16(23u);  /* 16-bit shift macro overflows; sets mask_32 = 0          */
mask_32 = DEF_BIT32(28u);  /* 32-bit shift macro correctly sets mask_32 = 0x10000000 */
```


3-2-2 DEF_BIT_MASK_xx()

Shifts a bit mask.

FILES

lib_def.h

PROTOTYPES

```
DEF_BIT_MASK(bit_mask, bit_shift);

DEF_BIT_MASK_08(bit_mask, bit_shift);
DEF_BIT_MASK_16(bit_mask, bit_shift);
DEF_BIT_MASK_32(bit_mask, bit_shift);
DEF_BIT_MASK_64(bit_mask, bit_shift);
```

ARGUMENTS

bit_mask Bit mask to shift.

bit_shift Number of bit positions to left-shift the bit mask.

RETURNED VALUE

bit_mask left-shifted by **bit_shift** number of bits.

REQUIRED CONFIGURATION

None.

NOTES / WARNINGS

bit_mask should be an unsigned integer. **bit_shift** should be a non-negative integer.

bit_shift values that overflow the target CPU and/or compiler environment (e.g. negative or greater-than-CPU-data-size values) may generate compiler warnings and/or errors.

EXAMPLE USAGE

```
CPU_INT16U  mask;
CPU_INT16U  mask_hi;
CPU_INT32U  mask_32;

mask        = 0x0065u;
mask_hi = DEF_BIT_MASK(mask, 8u);
mask_32 = DEF_BIT_MASK_16(mask, 10u); /* 16-bit shift macro overflows; sets mask_32 = 0x00009400 */
mask_32 = DEF_BIT_MASK_16(mask, 20u); /* 16-bit shift macro overflows; sets mask_32 = 0          */
mask_32 = DEF_BIT_MASK_32(mask, 20u); /* 32-bit shift macro correctly sets mask_32 = 0x06500000 */
```

3-2-3 DEF_BIT_FIELD_xx()

Creates a contiguous, multi-bit bit field.

FILES

lib_def.h

PROTOTYPES

```
DEF_BIT_FIELD(bit_field, bit_shift);

DEF_BIT_FIELD_08(bit_field, bit_shift);
DEF_BIT_FIELD_16(bit_field, bit_shift);
DEF_BIT_FIELD_32(bit_field, bit_shift);
DEF_BIT_FIELD_64(bit_field, bit_shift);
```

ARGUMENTS

bit_field Number of contiguous bits to set in the bit field.

bit_shift Number of bit positions to left-shift the bit field.

RETURNED VALUE

Contiguous bit field of **bit_field** number of bits left-shifted by **bit_shift** number of bits.

REQUIRED CONFIGURATION

None.

NOTES / WARNINGS

bit_field and **bit_shift** should be non-negative integers.

bit_field/bit_shift values that overflow the target CPU and/or compiler environment (e.g. negative or greater-than-CPU-data-size values) may generate compiler warnings and/or errors.

To avoid overflowing any target CPU and/or compiler's integer data type, unsigned bit constant 1 is either cast to specified integer data type size or suffixed with **long** integer modifier, 'L'. This may still be insufficient for CPUs and/or compilers that support **long long** integer data types, in which case 'LL' integer modifier should be suffixed. However, since almost all 16- and 32-bit CPUs and compilers support **long** integer data types but many may not support **long long** integer data types, only **long** integer data types and modifiers are supported.

EXAMPLE USAGE

```
CPU_INT08U upper_nibble;
CPU_INT32U mask_32;

upper_nibble = DEF_BIT_FIELD(4u, 4u);

mask_32 = DEF_BIT_FIELD_16(7u, 13u); /* 16-bit shift macro overflows; sets mask_32 = 0x0000E000 */
mask_32 = DEF_BIT_FIELD_16(7u, 23u); /* 16-bit shift macro overflows; sets mask_32 = 0          */
mask_32 = DEF_BIT_FIELD_32(7u, 23u); /* 32-bit shift macro correctly sets mask_32 = 0x3F800000 */
```

3-2-4 DEF_BIT_SET_xx()

Sets the appropriate bits in a value according to a specified bit mask.

FILES

lib_def.h

PROTOTYPE

```
DEF_BIT_SET(val, mask);

DEF_BIT_SET_08(val, mask);
DEF_BIT_SET_16(val, mask);
DEF_BIT_SET_32(val, mask);
DEF_BIT_SET_64(val, mask);
```

ARGUMENTS

val Value to modify by setting the specified bits.

mask Mask of bits to set in the value.

RETURNED VALUE

Modified value with specified bits set.

REQUIRED CONFIGURATION

None.

NOTES / WARNINGS

val and **mask** should be unsigned integers.

EXAMPLE USAGE

```
CPU_INT16U  flags;  
CPU_INT16U  flags_alarm;  
CPU_INT32U  events;  
  
flags        = Get_current_flags;  
flags_alarm = DEF_BIT_00 | DEF_BIT_03;  
DEF_BIT_SET(flags, flags_alarm);  
DEF_BIT_SET_32(events, flags);
```

3-2-5 DEF_BIT_CLR_xx()

Clears the appropriate bits in a value according to a specified bit mask.

FILES

lib_def.h

PROTOTYPE

```
DEF_BIT_CLR(val, mask);  
  
DEF_BIT_CLR_08(val, mask);  
DEF_BIT_CLR_16(val, mask);  
DEF_BIT_CLR_32(val, mask);  
DEF_BIT_CLR_64(val, mask);
```

ARGUMENTS

val Value to modify by clearing the specified bits.

mask Mask of bits to clear in the value.

RETURNED VALUE

Modified value with specified bits clear.

REQUIRED CONFIGURATION

None.

NOTES / WARNINGS

val and **mask** should be unsigned integers.

EXAMPLE USAGE

```
CPU_INT16U  flags;  
CPU_INT16U  flags_alarm;  
  
flags        = Get current flags;  
flags_alarm = DEF_BIT_00 | DEF_BIT_03;  
DEF_BIT_CLR(flags, flags_alarm);  
DEF_BIT_CLR_32(events, flags);
```


3-2-6 DEF_BIT_IS_SET()

Determines if all the specified bits in a value are set according to a specified bit mask.

FILES

`lib_def.h`

PROTOTYPE

```
DEF_BIT_IS_SET(val, mask);
```

ARGUMENTS

val Value to test if the specified bits are set.

mask Mask of bits to check if set in the value.

RETURNED VALUE

DEF_YES If all the bits in the bit **mask** are set in **val**;

DEF_NO if all the bits in the bit **mask** are not set in **val**.

REQUIRED CONFIGURATION

None.

NOTES / WARNINGS

val and **mask** should be unsigned integers. **NULL mask** (i.e., mask of value 0) allowed; returns **DEF_NO** since no mask bits specified.

EXAMPLE USAGE

```
CPU_INT16U  flags;  
CPU_INT16U  flags_mask;  
CPU_INT16U  flags_set;  
  
flags       = 0x0369u;  
flags_mask = DEF_BIT_08 | DEF_BIT_09;  
flags_set  = DEF_BIT_IS_SET(flags, flags_mask);
```

3-2-7 DEF_BIT_IS_CLR()

Determines if all the specified bits in a value are clear according to a specified bit mask.

FILES

lib_def.h

PROTOTYPE

```
DEF_BIT_IS_CLR(val, mask);
```

ARGUMENTS

val Value to test if the specified bits are clear.

mask Mask of bits to check if clear in the value.

RETURNED VALUE

DEF_YES If all the bits in the bit **mask** are clear in **val**;

DEF_NO if all the bits in the bit **mask** are not clear in **val**.

REQUIRED CONFIGURATION

None.

NOTES / WARNINGS

val and **mask** should be unsigned integers. **NULL mask** (i.e., mask of value 0) allowed; returns **DEF_NO** since no mask bits specified.

EXAMPLE USAGE

```
CPU_INT16U  alarms;  
CPU_INT16U  alarms_mask;  
CPU_INT16U  alarms_clr;  
  
alarms      = 0x07F0u;  
alarms_mask = DEF_BIT_04 | DEF_BIT_03;  
alarms_clr  = DEF_BIT_IS_CLR(alarms, alarms_mask);
```

3-2-8 DEF_BIT_IS_SET_ANY()

Determines if any of the specified bits in a value are set according to a specified bit mask.

FILES

`lib_def.h`

PROTOTYPE

```
DEF_BIT_IS_SET_ANY(val, mask);
```

ARGUMENTS

val Value to test if any of the specified bits are set.

mask Mask of bits to check if set in the value.

RETURNED VALUE

DEF_YES If any of the bits in the bit **mask** are set in **val**;

DEF_NO if all the bits in the bit **mask** are clear in **val**.

REQUIRED CONFIGURATION

None.

NOTES / WARNINGS

val and **mask** should be unsigned integers. **NULL mask** (i.e., mask of value 0) allowed; returns **DEF_NO** since no mask bits specified.

EXAMPLE USAGE

```
CPU_INT16U  flags;  
CPU_INT16U  flags_mask;  
CPU_INT16U  flags_set;  
  
flags       = 0x0369u;  
flags_mask = DEF_BIT_08 | DEF_BIT_09;  
flags_set  = DEF_BIT_IS_SET_ANY(flags, flags_mask);
```

3-2-9 DEF_BIT_IS_CLR_ANY()

Determines if any of the specified bits in a value are clear according to a specified bit mask.

FILES

`lib_def.h`

PROTOTYPE

```
DEF_BIT_IS_CLR_ANY(val, mask);
```

ARGUMENTS

val Value to test if any of the specified bits are clear.

mask Mask of bits to check if clear in the value.

RETURNED VALUE

DEF_YES If any of the bits in the bit **mask** are clear in **val**;

DEF_NO if all the bits in the bit **mask** are set in **val**.

REQUIRED CONFIGURATION

None.

NOTES / WARNINGS

val and **mask** should be unsigned integers. **NULL mask** (i.e., mask of value 0) allowed; returns **DEF_NO** since no mask bits specified.

EXAMPLE USAGE

```
CPU_INT16U  alarms;  
CPU_INT16U  alarms_mask;  
CPU_INT16U  alarms_clr;  
  
alarms      = 0x07F0u;  
alarms_mask = DEF_BIT_04 | DEF_BIT_03;  
alarms_clr  = DEF_BIT_IS_CLR_ANY(alarms, alarms_mask);
```


3-2-10 DEF_CHK_VAL_MIN()

Validates a value as greater than or equal to a specified minimum value.

FILES

lib_def.h

PROTOTYPE

```
DEF_CHK_VAL_MIN(val, val_min);
```

ARGUMENTS

val Value to validate.

val_min Minimum value to test.

RETURNED VALUE

DEF_OK Value is greater than or equal to minimum value;

DEF_FAIL otherwise.

REQUIRED CONFIGURATION

None.

NOTES / WARNINGS

DEF_CHK_VAL_MIN() avoids directly comparing any two values if only one of the values is negative since the negative value might be incorrectly promoted to an arbitrary unsigned value if the other value to compare is unsigned.

Validation of values is limited to the range supported by the compiler and/or target environment. All other values that underflow/overflow the supported range will modulo/wrap into the supported range as arbitrary signed or unsigned values. Therefore, any values that underflow the most negative signed value or overflow the most positive unsigned value supported by the compiler and/or target environment cannot be validated:

$$\begin{aligned} & (\quad N-1 \quad \quad \quad N \quad \quad] \\ & (-(2^{\quad}) \quad , \quad 2^{\quad} - 1 \quad] \\ & (\quad \quad \quad \quad \quad] \end{aligned}$$

where **N** is the number of data word bits supported by the compiler and/or target environment. Note that the most negative value, $-2^{(N-1)}$, is not included in the supported range since many compilers do not always correctly handle this value.

EXAMPLE USAGE

```
#define CFG_VAL          -1

#if (DEF_CHK_VAL_MIN(CFG_VAL, 0u) != DEF_OK) /* Signed CFG_VAL NOT promoted to unsigned. */
#error "CFG_VAL must be >= 0"
#endif
```

3-2-11 DEF_CHK_VAL_MAX()

Validates a value as less than or equal to a specified maximum value.

FILES

lib_def.h

PROTOTYPE

```
DEF_CHK_VAL_MAX(val, val_max);
```

ARGUMENTS

val Value to validate.

val_max Maximum value to test.

RETURNED VALUE

DEF_OK Value is less than or equal to maximum value;

DEF_FAIL otherwise.

REQUIRED CONFIGURATION

None.

NOTES / WARNINGS

DEF_CHK_VAL_MAX() avoids directly comparing any two values if only one of the values is negative since the negative value might be incorrectly promoted to an arbitrary unsigned value if the other value to compare is unsigned.

Validation of values is limited to the range supported by the compiler and/or target environment. All other values that underflow/overflow the supported range will modulo/wrap into the supported range as arbitrary signed or unsigned values. Therefore, any values that underflow the most negative signed value or overflow the most positive unsigned value supported by the compiler and/or target environment cannot be validated:

$$\begin{aligned} & (\quad N-1 \quad \quad \quad N \quad \quad] \\ & (-(2^{\quad}) \quad , \quad 2^{\quad} - 1 \quad] \\ & (\quad \quad \quad \quad \quad] \end{aligned}$$

where **N** is the number of data word bits supported by the compiler and/or target environment. Note that the most negative value, $-2^{(N-1)}$, is not included in the supported range since many compilers do not always correctly handle this value.

EXAMPLE USAGE

```
#define CFG_VAL          -1

#if (DEF_CHK_VAL_MAX(CFG_VAL, 1000u) != DEF_OK) /* Signed CFG_VAL NOT promoted to unsigned. */
#error "CFG_VAL must be <= 100"
#endif
```

3-2-12 DEF_CHK_VAL()

Validates a value as greater than or equal to a specified minimum value and less than or equal to a specified maximum value.

FILES

lib_def.h

PROTOTYPE

```
DEF_CHK_VAL(val, val_min, val_max);
```

ARGUMENTS

val Value to validate.

val_min Minimum value to test.

val_max Maximum value to test.

RETURNED VALUE

DEF_OK Value is greater than or equal to minimum value AND less than or equal to maximum value;

DEF_FAIL otherwise.

REQUIRED CONFIGURATION

None.

NOTES / WARNINGS

DEF_CHK_VAL() avoids directly comparing any two values if only one of the values is negative since the negative value might be incorrectly promoted to an arbitrary unsigned value if the other value to compare is unsigned.

Validation of values is limited to the range supported by the compiler and/or target environment. All other values that underflow/overflow the supported range will modulo/wrap into the supported range as arbitrary signed or unsigned values. Therefore, any values that underflow the most negative signed value or overflow the most positive unsigned value supported by the compiler and/or target environment cannot be validated:

$$\begin{aligned} & (\quad N-1 \quad \quad \quad N \quad \quad] \\ & (-(2 \quad) \quad , \quad 2 \quad - 1 \quad] \\ & (\quad \quad \quad \quad \quad \quad] \end{aligned}$$

where **N** is the number of data word bits supported by the compiler and/or target environment. Note that the most negative value, $-2^{(N-1)}$, is not included in the supported range since many compilers do not always correctly handle this value.

DEF_CHK_VAL() does not validate that the maximum value (**val_max**) is greater than or equal to the minimum value (**val_min**).

EXAMPLE USAGE

```
#define CFG_VAL          -1

#if (DEF_CHK_VAL_MAX(CFG_VAL, 0u, 1000u) != DEF_OK) /* Signed CFG_VAL NOT promoted to unsigned. */
#error "CFG_VAL must be >= 0 and <= 100"
#endif
```

3-2-13 DEF_GET_U_MAX_VAL()

Gets the maximum unsigned value that can be represented in an unsigned integer variable of the same data type size as an object.

FILES

lib_def.h

PROTOTYPE

```
DEF_GET_U_MAX_VAL(obj);
```

ARGUMENTS

obj Object or data type to return maximum unsigned value.

RETURNED VALUE

Maximum unsigned integer value that can be represented by the object.

REQUIRED CONFIGURATION

None.

NOTES / WARNINGS

obj should be an integer object or data type but could also be a character or pointer object or data type.

EXAMPLE USAGE

```
CPU_DATA val;

val = Some val;
if (val == DEF_GET_U_MAX_VAL(val)) {
    handle max val condition;
}
```

3-2-14 DEF_MIN()

Determines the minimum of two values.

FILES

`lib_def.h`

PROTOTYPE

```
DEF_MIN(a, b);
```

ARGUMENTS

a First value in minimum comparison.

b Second value in minimum comparison.

RETURNED VALUE

The lesser of the two values, **a** or **b**.

REQUIRED CONFIGURATION

None.

NOTES / WARNINGS

Ideally, `DEF_MIN()` should be defined in the custom mathematics library, `lib_math.*`. However, to maintain backwards compatibility with previously-released modules, `DEF_MIN()` is still defined in `lib_def.h`.

EXAMPLE USAGE

```
CPU_INT16S  x;  
CPU_INT16S  y;  
CPU_INT16S  z;  
  
x = 100;  
y = -101;  
z = DEF_MIN(x, y);
```

3-2-15 DEF_MAX()

Determines the maximum of two values.

FILES

`lib_def.h`

PROTOTYPE

```
DEF_MAX(a, b);
```

ARGUMENTS

a First value in maximum comparison.

b Second value in maximum comparison.

RETURNED VALUE

The greater of the two values, **a** or **b**.

REQUIRED CONFIGURATION

None.

NOTES / WARNINGS

Ideally, `DEF_MAX()` should be defined in the custom mathematics library, `lib_math.*`. However, to maintain backwards compatibility with previously-released modules, `DEF_MAX()` is still defined in `lib_def.h`.

EXAMPLE USAGE

```
CPU_INT16S  x;  
CPU_INT16S  y;  
CPU_INT16S  z;  
  
x = 100;  
y = -101;  
z = DEF_MAX(x, y);
```

3-2-16 DEF_ABS()

Determines the absolute value of a value.

FILES

`lib_def.h`

PROTOTYPE

```
DEF_ABS(a);
```

ARGUMENTS

a Value to calculate absolute value.

RETURNED VALUE

The absolute value of **a**.

REQUIRED CONFIGURATION

None.

NOTES / WARNINGS

Ideally, `DEF_ABS()` should be defined in the custom mathematics library, `lib_math.*`. However, to maintain backwards compatibility with previously-released modules, `DEF_ABS()` is still defined in `lib_def.h`.

EXAMPLE USAGE

```
CPU_INT16S  x;  
CPU_INT16S  z;  
  
x = -101;  
z = DEF_ABS(x);
```

µC/LIB Memory Library

µC/LIB contains library functions that replace standard library memory functions such as `memclr()`, `memset()`, `memcpy()`, `memcmp()`, etc; as well as generic versions of network functions, `ntohl()`, `ntohs()`, `htonl()`, `htons()`. These functions and macros are defined in `lib_mem.c` and `lib_mem.h`.

4-1 MEMORY LIBRARY CONFIGURATION

The following µC/LIB memory library configurations may be optionally configured in `app_cfg.h` :

<code>LIB_MEM_CFG_OPTIMIZE_ASM_EN</code>	Implement certain memory library functionality in assembly-optimized files (see section 4-4-11). This feature may be configured to either <code>DEF_DISABLED</code> or <code>DEF_ENABLED</code> .
<code>LIB_MEM_CFG_ARG_CHK_EXT_EN</code>	Includes code to check external arguments for functions called by the user. This feature may be configured to either <code>DEF_DISABLED</code> or <code>DEF_ENABLED</code> .
<code>LIB_MEM_CFG_ALLOC_EN</code>	Include memory allocation functionality (see section 4-4). This feature may be configured to either <code>DEF_DISABLED</code> or <code>DEF_ENABLED</code> .
<code>LIB_MEM_CFG_HEAP_SIZE</code>	Heap size, in octets (see section 4-4).
<code>LIB_MEM_CFG_HEAP_BASE_ADDR</code>	Heap base address (see section 4-4).

4-2 MEMORY LIBRARY MACROS

4-2-1 MEM_VAL_BIG_TO_LITTLE_xx() / MEM_VAL_LITTLE_TO_BIG_xx()

These macros convert data values to and to/from big-endian to/from little-endian word order.

FILES

lib_mem.h

PROTOTYPES

```
MEM_VAL_BIG_TO_LITTLE_16(val);  
MEM_VAL_BIG_TO_LITTLE_32(val);  
  
MEM_VAL_LITTLE_TO_BIG_16(val);  
MEM_VAL_LITTLE_TO_BIG_32(val);
```

ARGUMENTS

val Data value to convert.

RETURNED VALUE

Converted data value.

REQUIRED CONFIGURATION

None.

NOTES / WARNINGS

Convert data values to the desired data-word order:

MEM_VAL_BIG_TO_LITTLE_xx() Convert big-endian data values to little-endian data values

MEM_VAL_LITTLE_TO_BIG_xx() Convert little-endian data values to big-endian data values

val data value to convert and any variable to receive the returned conversion must start on appropriate CPU word-aligned addresses.

MEM_VAL_COPY_GET_XXX()/**MEM_VAL_COPY_SET_XXX()** macros (see section 4-2-6 and section 4-2-7) are more efficient than **MEM_VAL_BIG_TO_LITTLE_XX()**/**MEM_VAL_LITTLE_TO_BIG_XX()** macros and are also fully independent of CPU data-word-alignment and should be used whenever possible.

MEM_VAL_BIG_TO_LITTLE_XX()/**MEM_VAL_LITTLE_TO_BIG_XX()** macros are not atomic operations and must not be used on any non-static (i.e., volatile) variables, registers, hardware, etc.; without the caller of the macros providing some form of additional protection (e.g. mutual exclusion).

EXAMPLE USAGE

```
CPU_INT32U  val_32_little;
CPU_INT32U  val_32_big;

val_32_big  = SomeBigEndianVal;
val_32_little = MEM_VAL_BIG_TO_LITTLE_32(val_32_big);
```

4-2-2 MEM_VAL_BIG_TO_HOST_xx() / MEM_VAL_HOST_TO_BIG_xx()

These macros convert data values to and to/from big-endian to/from host-endian CPU word order.

FILES

lib_mem.h

PROTOTYPES

```
MEM_VAL_BIG_TO_HOST_16(val);  
MEM_VAL_BIG_TO_HOST_32(val);  
  
MEM_VAL_HOST_TO_BIG_16(val);  
MEM_VAL_HOST_TO_BIG_32(val);
```

ARGUMENTS

val Data value to convert.

RETURNED VALUE

Converted data value.

REQUIRED CONFIGURATION

None.

NOTES / WARNINGS

Convert data values to the desired data-word order:

MEM_VAL_BIG_TO_HOST_xx() Convert big-endian data values to host-endian data values

MEM_VAL_HOST_TO_BIG_xx() Convert host-endian data values to big-endian data values

val data value to convert and any variable to receive the returned conversion must start on appropriate CPU word-aligned addresses.

`MEM_VAL_COPY_GET_xxx()`/`MEM_VAL_COPY_SET_xxx()` macros (see section 4-2-6 and section 4-2-7) are more efficient than `MEM_VAL_BIG_TO_HOST_xx()`/`MEM_VAL_HOST_TO_BIG_xx()` macros and are also fully independent of CPU data-word-alignment and should be used whenever possible.

`MEM_VAL_BIG_TO_HOST_xx()`/`MEM_VAL_HOST_TO_BIG_xx()` macros are not atomic operations and must not be used on any non-static (i.e., volatile) variables, registers, hardware, etc.; without the caller of the macros providing some form of additional protection (e.g. mutual exclusion).

EXAMPLE USAGE

```
CPU_INT32U val_32_host;
CPU_INT32U val_32_big;

val_32_host = SomeHostEndianVal;
val_32_big = MEM_VAL_HOST_TO_BIG_32(val_32_host);
```

4-2-3 MEM_VAL_LITTLE_TO_HOST_xx() / MEM_VAL_HOST_TO_LITTLE_xx()

These macros convert data values to and to/from little-endian to/from host-endian CPU word order.

FILES

lib_mem.h

PROTOTYPES

```
MEM_VAL_LITTLE_TO_HOST_16(val);  
MEM_VAL_LITTLE_TO_HOST_32(val);  
  
MEM_VAL_HOST_TO_LITTLE_16(val);  
MEM_VAL_HOST_TO_LITTLE_32(val);
```

ARGUMENTS

val Data value to convert.

RETURNED VALUE

Converted data value.

REQUIRED CONFIGURATION

None.

NOTES / WARNINGS

Convert data values to the desired data-word order:

MEM_VAL_LITTLE_TO_HOST_xx() Convert little-endian data values to host-endian data values

MEM_VAL_HOST_TO_LITTLE_xx() Convert host-endian data values to little-endian data values

val data value to convert and any variable to receive the returned conversion must start on appropriate CPU word-aligned addresses.

`MEM_VAL_COPY_GET_xxx()`/`MEM_VAL_COPY_SET_xxx()` macros (see section 4-2-6 and section 4-2-7) are more efficient than `MEM_VAL_LITTLE_TO_HOST_xx()`/`MEM_VAL_HOST_TO_LITTLE_xx()` macros and are also fully independent of CPU data-word-alignment and should be used whenever possible.

`MEM_VAL_LITTLE_TO_HOST_xx()`/`MEM_VAL_HOST_TO_LITTLE_xx()` macros are not atomic operations and must not be used on any non-static (i.e., volatile) variables, registers, hardware, etc.; without the caller of the macros providing some form of additional protection (e.g. mutual exclusion).

EXAMPLE USAGE

```
CPU_INT16U val_16_host;
CPU_INT16U val_16_little;

val_16_little = SomeLittleEndianVal;
val_16_host   = MEM_VAL_LITTLE_TO_HOST_16(val_16_little);
```

4-2-4 MEM_VAL_GET_XXX()

These macros decode data values from any CPU memory address.

FILES

lib_mem.h

PROTOTYPES

```
MEM_VAL_GET_INT08U_BIG(addr);  
MEM_VAL_GET_INT16U_BIG(addr);  
MEM_VAL_GET_INT32U_BIG(addr);  
  
MEM_VAL_GET_INT08U_LITTLE(addr);  
MEM_VAL_GET_INT16U_LITTLE(addr);  
MEM_VAL_GET_INT32U_LITTLE(addr);  
  
MEM_VAL_GET_INT08U(addr);  
MEM_VAL_GET_INT16U(addr);  
MEM_VAL_GET_INT32U(addr);
```

ARGUMENTS

addr Lowest CPU memory address of the data value to decode.

RETURNED VALUE

Decoded data value from CPU memory address.

REQUIRED CONFIGURATION

None.

NOTES / WARNINGS

CPU memory addresses/pointers not checked for NULL.

Decode data values based on the values' data-word order in CPU memory:

<code>MEM_VAL_GET_XXX_BIG()</code>	Decode big- endian data values — data words' most significant octet at lowest memory address
<code>MEM_VAL_GET_XXX_LITTLE()</code>	Decode little-endian data values — data words' least significant octet at lowest memory address
<code>MEM_VAL_GET_XXX()</code>	Decode data values using CPU's native or configured data-word order

`MEM_VAL_GET_XXX()` macros decode data values without regard to CPU word-aligned addresses. Thus for processors that require data word alignment, data words can be decoded from any CPU address, word-aligned or not, without generating data-word-alignment exceptions/faults. However, any variable to receive the returned data value must start on an appropriate CPU word-aligned address.

`MEM_VAL_COPY_GET_XXX()` macros (see section 4-2-6) are more efficient than `MEM_VAL_GET_XXX()` macros and are also fully independent of CPU data-word-alignment and should be used whenever possible.

`MEM_VAL_GET_XXX()` macros are not atomic operations and must not be used on any non-static (i.e., volatile) variables, registers, hardware, etc.; without the caller of the macros providing some form of additional protection (e.g. mutual exclusion).

EXAMPLE USAGE

```
CPU_INT08U *pval;
CPU_INT16U val;

pval = &SomeAddr;          /* Any CPU address */
val = MEM_VAL_GET_INT16U(pval);
```

4-2-5 MEM_VAL_SET_xxx()

These macros encode data values to any CPU memory address.

FILES

lib_mem.h

PROTOTYPES

```
MEM_VAL_SET_INT08U_BIG(addr);  
MEM_VAL_SET_INT16U_BIG(addr);  
MEM_VAL_SET_INT32U_BIG(addr);  
  
MEM_VAL_SET_INT08U_LITTLE(addr);  
MEM_VAL_SET_INT16U_LITTLE(addr);  
MEM_VAL_SET_INT32U_LITTLE(addr);  
  
MEM_VAL_SET_INT08U(addr);  
MEM_VAL_SET_INT16U(addr);  
MEM_VAL_SET_INT32U(addr);
```

ARGUMENTS

addr Lowest CPU memory address to encode the data value.

val Data value to encode.

RETURNED VALUE

None.

REQUIRED CONFIGURATION

None.

NOTES / WARNINGS

CPU memory addresses/pointers not checked for **NULL**.

Encode data values based on the values' data-word order in CPU memory:

<code>MEM_VAL_SET_XXX_BIG()</code>	Encode big- endian data values — data words' most significant octet at lowest memory address
<code>MEM_VAL_SET_XXX_LITTLE()</code>	Encode little-endian data values — data words' least significant octet at lowest memory address
<code>MEM_VAL_SET_XXX()</code>	Encode data values using CPU's native or configured data-word order

`MEM_VAL_SET_XXX()` macros encode data values without regard to CPU word-aligned addresses. Thus for processors that require data word alignment, data words can be encoded to any CPU address, word-aligned or not, without generating data-word-alignment exceptions/faults. However, **val** data value to encode must start on appropriate CPU word-aligned address.

`MEM_VAL_COPY_SET_XXX()` macros (see section 4-2-7) are more efficient than `MEM_VAL_SET_XXX()` macros and are also fully independent of CPU data-word-alignment and should be used whenever possible.

`MEM_VAL_SET_XXX()` macros are not atomic operations and must not be used on any non-static (i.e., volatile) variables, registers, hardware, etc.; without the caller of the macros providing some form of additional protection (e.g. mutual exclusion).

EXAMPLE USAGE

```
CPU_INT08U  *pval;
CPU_INT16U  val;

pval = &SomeAddr;          /* Any CPU address */
val = 0xABCDu;
MEM_VAL_SET_INT16U(pval, val);
```

4-2-6 MEM_VAL_COPY_GET_XXX()

These macros copy and decode data values from any CPU memory address to any other memory address.

FILES

lib_mem.h

PROTOTYPES

```
MEM_VAL_COPY_GET_INT08U_BIG(addr_dest, addr_src);
MEM_VAL_COPY_GET_INT16U_BIG(addr_dest, addr_src);
MEM_VAL_COPY_GET_INT32U_BIG(addr_dest, addr_src);
MEM_VAL_COPY_GET_INTU_BIG(addr_dest, addr_src, val_size);

MEM_VAL_COPY_GET_INT08U_LITTLE(addr_dest, addr_src);
MEM_VAL_COPY_GET_INT16U_LITTLE(addr_dest, addr_src);
MEM_VAL_COPY_GET_INT32U_LITTLE(addr_dest, addr_src);
MEM_VAL_COPY_GET_INTU_LITTLE(addr_dest, addr_src, val_size);

MEM_VAL_COPY_GET_INT08U(addr_dest, addr_src);
MEM_VAL_COPY_GET_INT16U(addr_dest, addr_src);
MEM_VAL_COPY_GET_INT32U(addr_dest, addr_src);
MEM_VAL_COPY_GET_INTU(addr_dest, addr_src, val_size);
```

ARGUMENTS

addr_dest Lowest CPU memory address to copy/decode source address's data value.

addr_src Lowest CPU memory address of the data value to copy/decode.

val_size Number of data value octets to copy/decode.

RETURNED VALUE

None.

REQUIRED CONFIGURATION

None.

NOTES / WARNINGS

CPU memory addresses/pointers not checked for **NULL** nor overlapping memory addresses which may result in undefined copy behavior.

Copy/decode data values based on the values' data-word order in CPU memory:

MEM_VAL_COPY_GET_XXX_BIG() Decode big- endian data values — data words' most significant octet at lowest memory address

MEM_VAL_COPY_GET_XXX_LITTLE() Decode little-endian data values — data words' least significant octet at lowest memory address

MEM_VAL_COPY_GET_XXX() Decode data values using CPU's native or configured data-word order

MEM_VAL_COPY_GET_XXX() macros copy/decode data values without regard to CPU word-aligned addresses. Thus for processors that require data word alignment, data words can be copied/decoded to/from any CPU addresses, word-aligned or not, without generating data-word-alignment exceptions/faults.

MEM_VAL_COPY_GET_XXX() macros are more efficient than **MEM_VAL_GET_XXX()** macros (see section 4-2-4) and are also fully independent of CPU data-word-alignment and should be used whenever possible. Fixed-size copy **MEM_VAL_COPY_GET_INTxxU_XXX()** macros are more efficient than dynamic-size copy **MEM_VAL_COPY_GET_INTU_XXX()** macros and should be used whenever possible.

MEM_VAL_COPY_GET_XXX() macros are not atomic operations and must not be used on any non-static (i.e., volatile) variables, registers, hardware, etc.; without the caller of the macros providing some form of additional protection (e.g. mutual exclusion).

Since octet-order copy/conversion are inverse operations, **MEM_VAL_COPY_GET_XXX()** and **MEM_VAL_COPY_SET_XXX()** memory data-copy get/set macros are inverse, but identical, operations and are provided in both forms for semantics and consistency. See also section 4-2-7.

EXAMPLE USAGE

```
CPU_INT16U  *pmem;
CPU_INT16U  *pval;
CPU_INT08U  buf[SIZE];

pmem = &SomeAddr;           /* Any CPU address */
pval = &SomeVal;             /* Any CPU address */
MEM_VAL_COPY_GET_INT16U(pmem, pval);
MEM_VAL_COPY_GET_INTU(&buf[0], pmem, sizeof(buf));
```

4-2-7 MEM_VAL_COPY_SET_XXX()

These macros copy and encode data values from any CPU memory address to any other memory address.

FILES

lib_mem.h

PROTOTYPES

```
MEM_VAL_COPY_SET_INT08U_BIG(addr_dest, addr_src);
MEM_VAL_COPY_SET_INT16U_BIG(addr_dest, addr_src);
MEM_VAL_COPY_SET_INT32U_BIG(addr_dest, addr_src);
MEM_VAL_COPY_SET_INTU_BIG(addr_dest, addr_src, val_size);

MEM_VAL_COPY_SET_INT08U_LITTLE(addr_dest, addr_src);
MEM_VAL_COPY_SET_INT16U_LITTLE(addr_dest, addr_src);
MEM_VAL_COPY_SET_INT32U_LITTLE(addr_dest, addr_src);
MEM_VAL_COPY_SET_INTU_LITTLE(addr_dest, addr_src, val_size);

MEM_VAL_COPY_SET_INT08U(addr_dest, addr_src);
MEM_VAL_COPY_SET_INT16U(addr_dest, addr_src);
MEM_VAL_COPY_SET_INT32U(addr_dest, addr_src);
MEM_VAL_COPY_SET_INTU(addr_dest, addr_src, val_size);
```

ARGUMENTS

addr_dest Lowest CPU memory address to copy/encode source address's data value.

addr_src Lowest CPU memory address of the data value to copy/encode.

val_size Number of data value octets to copy/encode.

RETURNED VALUE

None.

REQUIRED CONFIGURATION

None.

NOTES / WARNINGS

CPU memory addresses/pointers not checked for **NULL** nor overlapping memory addresses which may result in undefined copy behavior.

Copy/encode data values based on the values' data-word order in CPU memory:

MEM_VAL_COPY_SET_XXX_BIG() Encode big- endian data values — data words' most significant octet at lowest memory address

MEM_VAL_COPY_SET_XXX_LITTLE() Encode little-endian data values — data words' least significant octet at lowest memory address

MEM_VAL_COPY_SET_XXX() Encode data values using CPU's native or configured data-word order

MEM_VAL_COPY_SET_XXX() macros copy/encode data values without regard to CPU word-aligned addresses. Thus for processors that require data word alignment, data words can be copied/encoded to/from any CPU addresses, word-aligned or not, without generating data-word-alignment exceptions/faults.

MEM_VAL_COPY_SET_XXX() macros are more efficient than **MEM_VAL_SET_XXX()** macros (see section 4-2-5) and are also fully independent of CPU data-word-alignment and should be used whenever possible. Fixed-size copy **MEM_VAL_COPY_SET_INTxxU_XXX()** macros are more efficient than dynamic-size copy **MEM_VAL_COPY_SET_INTU_XXX()** macros and should be used whenever possible.

MEM_VAL_COPY_SET_XXX() macros are not atomic operations and must not be used on any non-static (i.e., volatile) variables, registers, hardware, etc.; without the caller of the macros providing some form of additional protection (e.g. mutual exclusion).

Since octet-order copy/conversion are inverse operations, **MEM_VAL_COPY_GET_XXX()** and **MEM_VAL_COPY_SET_XXX()** memory data-copy get/set macros are inverse, but identical, operations and are provided in both forms for semantics and consistency. See also section 4-2-6.

EXAMPLE USAGE

```
CPU_INT16U  *pmem;
CPU_INT16U  *pval;
CPU_INT08U  buf[SIZE];

pmem = &SomeAddr;           /* Any CPU address */
pval = &SomeVal;             /* Any CPU address */
MEM_VAL_COPY_SET_INT16U(pmem, pval);
MEM_VAL_COPY_SET_INTU(&buf[0], pmem, sizeof(buf));
```

4-2-8 MEM_VAL_COPY_xxx()

These macros copy data values from any CPU memory address to any other memory address.

FILES

lib_mem.h

PROTOTYPES

```
MEM_VAL_COPY_08(addr_dest, addr_src);  
MEM_VAL_COPY_16(addr_dest, addr_src);  
MEM_VAL_COPY_32(addr_dest, addr_src);  
  
MEM_VAL_COPY(addr_dest, addr_src, val_size);
```

ARGUMENTS

addr_dest Lowest CPU memory address to copy source address's data value.

addr_src Lowest CPU memory address of the data value to copy.

val_size Number of data value octets to copy.

RETURNED VALUE

None.

REQUIRED CONFIGURATION

None.

NOTES / WARNINGS

CPU memory addresses/pointers not checked for **NULL** nor overlapping memory addresses which may result in undefined copy behavior.

MEM_VAL_COPY_xxx() macros copy data values based on CPU's native data-word order.

MEM_VAL_COPY_XXX() macros copy data values without regard to CPU word-aligned addresses. Thus for processors that require data word alignment, data words can be copied to/from any CPU addresses, word-aligned or not, without generating data-word-alignment exceptions/faults.

Fixed-size copy **MEM_VAL_COPY_XXX()** macros are more efficient than dynamic-size copy **MEM_VAL_COPY()** macro and should be used whenever possible.

MEM_VAL_COPY_XXX() macros are not atomic operations and must not be used on any non-static (i.e. volatile) variables, registers, hardware, etc; without the caller of the macros providing some form of additional protection (e.g. mutual exclusion).

EXAMPLE USAGE

```
CPU_INT16U  *pmem;
CPU_INT16U  *pval;

pmem = &SomeAddr;      /* Any CPU address */
pval = &SomeVal;        /* Any CPU address */
MEM_VAL_COPY_16(pmem, pval);
```

4-3 MEMORY LIBRARY FUNCTIONS

4-3-1 Mem_Clr()

Clears a memory buffer. In other words, set all octets in the memory buffer to a value of '0'.

FILES

lib_mem.h/lib_mem.c

PROTOTYPE

```
void Mem_Clr (void      *pmem,  
              CPU_SIZE_T size);
```

ARGUMENTS

pmem Pointer to the memory buffer to be clear.

size Number of memory buffer octets to clear.

RETURNED VALUE

None.

REQUIRED CONFIGURATION

None.

NOTES / WARNINGS

Zero-sized clears allowed.

EXAMPLE USAGE

```
CPU_CHAR AppBuf[10];  
  
Mem_Clr((void      *)&AppBuf[0],  
        (CPU_SIZE_T) sizeof(AppBuf));
```


4-3-2 Mem_Set()

Fills a memory buffer with a specific value. In other words, set all octets in the memory buffer to the specific value.

FILES

lib_mem.h/lib_mem.c

PROTOTYPE

```
void Mem_Set (void      *pmem,  
              CPU_INT08U data_val,  
              CPU_SIZE_T size);
```

ARGUMENTS

pmem Pointer to the memory buffer to be set with a specific value.

data_val Data value to set.

size Number of memory buffer octets to set.

RETURNED VALUE

None.

REQUIRED CONFIGURATION

None.

NOTES / WARNINGS

Zero-sized sets allowed.

EXAMPLE USAGE

```
CPU_CHAR AppBuf[10];  
  
Mem_Set((void *) &AppBuf[0],  
        (CPU_INT08U) 0x64,  
        (CPU_SIZE_T) sizeof(AppBuf));
```

4-3-3 Mem_Copy()

Copies values from one memory buffer to another memory buffer.

FILES

lib_mem.h/lib_mem.c

PROTOTYPE

```
void Mem_Copy (    void      *pdest,  
                  const void  *psrc,  
                  CPU_SIZE_T  size);
```

ARGUMENTS

pdest Pointer to the memory buffer to copy octets into.

psrc Pointer to the memory buffer to copy octets from.

size Number of memory buffer octets to copy.

RETURNED VALUE

None.

REQUIRED CONFIGURATION

None.

NOTES / WARNINGS

Zero-sized copies allowed.

Memory buffers not checked for overlapping. However, data octets from a source memory buffer at a higher address value should successfully copy to a destination memory buffer at a lower address value even if any octets of the memory buffers overlap as long as no individual copy overlaps. Since **Mem_Copy()** performs the data octet copy via **CPU_ALIGN**-sized words and/or octets; and since **CPU_ALIGN**-sized words must be accessed

on word-aligned addresses, neither **CPU_ALIGN**-sized words nor octets at unique addresses can ever overlap. Therefore, **Mem_Copy()** **should** be able to successfully copy overlapping memory buffers as long as the source memory buffer is at a higher address value than the destination memory buffer.

This function can be configured to build an assembly-optimized version (see section 4-4-11)

EXAMPLE USAGE

```
CPU_INT08U AppBuf[10];
CPU_INT08U DataBuf[20];

/* Set data buffer with value. */
Mem_Set ((void *)&DataBuf[0],
         (CPU_INT08U) 0x64,
         (CPU_SIZE_T) sizeof(DataBuf));

/* Copy data buffer to app buffer. */
Mem_Copy((void *)&AppBuf[0],
         (void *)&DataBuf[0],
         (CPU_SIZE_T) sizeof(AppBuf));
```

4-3-4 Mem_Cmp()

Compares values from two memory buffers.

FILES

lib_mem.h/lib_mem.c

PROTOTYPE

```
CPU_BOOLEAN Mem_Cmp (const void    *p1_mem,  
                     const void    *p2_mem,  
                     CPU_SIZE_T    size);
```

ARGUMENTS

p1_mem Pointer to the first memory buffer to compare.

p2_mem Pointer to the second memory buffer to compare.

size Number of memory buffer octets to compare.

RETURNED VALUE

DEF_YES, if size number of octets are identical in both memory buffers;

DEF_NO, otherwise.

REQUIRED CONFIGURATION

None.

NOTES / WARNINGS

Zero-sized compares allowed; **DEF_YES** returned for identical **NULL** compare.

EXAMPLE USAGE

```
CPU_INT08U   DataBuf_1[10];
CPU_INT08U   DataBuf_2[20];
CPU_SIZE_T   size;
CPU_BOOLEAN  cmp;

/* Set data buffers with values. */
Mem_Set((void *) &DataBuf_1[0],
        (CPU_INT08U) 0x64,
        (CPU_SIZE_T) sizeof(DataBuf_1));
Mem_Set((void *) &DataBuf_2[0],
        (CPU_INT08U) 0x33,
        (CPU_SIZE_T) sizeof(DataBuf_2));

/* Compare data buffers' values. */
size = DEF_MIN(sizeof(DataBuf_1),
               sizeof(DataBuf_2));
cmp = Mem_Cmp((void *) &DataBuf_1[0],
              (void *) &DataBuf_2[0],
              (CPU_SIZE_T) cmp_size);
```

4-4 MEMORY ALLOCATION FUNCTIONS

µC/LIB memory allocation functions provide for the allocation of memory from a general purpose-heap or the creation of memory pools. Single memory blocks may be allocated directly from the heap. However, in order to prevent fragmentation, these memory blocks cannot be freed back to the heap. Memory pool blocks can be allocated from either the general purpose-heap or from dedicated memory specified by the application. Memory pool blocks can be dynamically allocated and freed during application execution because memory pool blocks are fixed-size which prevents possible fragmentation.

The following µC/LIB memory library configurations must be configured in `app_cfg.h` to include memory allocation functionality:

<code>LIB_MEM_CFG_ALLOC_EN</code>	Must be configured to <code>DEF_ENABLED</code> to include memory allocation functionality and heap.
<code>LIB_MEM_CFG_HEAP_SIZE</code>	Must be configured to sufficient heap size, in octets. Memory pool pointers to memory blocks are always allocated from this heap. A memory pool can optionally have its memory blocks allocated from the heap as well. In addition, single memory blocks may be allocated directly from the heap. This configuration is required if memory allocation functionality is <code>DEF_ENABLED</code> .
<code>LIB_MEM_CFG_HEAP_BASE_ADDR</code>	May be optionally configured to specify the base address of heap memory. May be configured to any additional and/or dedicated memory (RAM). If configured, it is the developer's responsibility to ensure that the configured heap memory base address and size do not overlap any other system memory-linker- or memory-mapped.

4-4-1 Mem_Init()

Initializes the memory management module.

FILES

lib_mem.h/lib_mem.c

PROTOTYPE

```
void Mem_Init (void);
```

ARGUMENTS

None.

RETURNED VALUE

None.

REQUIRED CONFIGURATION

None.

NOTES / WARNINGS

Mem_Init() must be called by the application prior to calling any other memory allocation functions.

4-4-2 Mem_HeapAlloc()

Gets a single memory block from the heap.

FILES

lib_mem.h/lib_mem.c

PROTOTYPE

```
void *Mem_HeapAlloc (CPU_SIZE_T  size,
                    CPU_SIZE_T  align,
                    CPU_SIZE_T  *poctets_reqd,
                    LIB_ERR      *perr);
```

ARGUMENTS

size Size of requested memory block (in octets).

align Alignment of requested memory block (in octets).

poctets_reqd Pointer to a variable to ...

Return the number of octets required to successfully allocate the memory block, if any errors;

Return 0, otherwise.

perr Pointer to variable that will receive the return error code from this function:

```
LIB_MEM_ERR_NONE
LIB_MEM_ERR_INVALID_MEM_SIZE
LIB_MEM_ERR_INVALID_MEM_ALIGN
LIB_MEM_ERR_HEAP_EMPTY
LIB_MEM_ERR_HEAP_OVF
```


RETURNED VALUE

Pointer to memory block, if no errors;

Pointer to `NULL`, otherwise.

REQUIRED CONFIGURATION

Available only if `LIB_MEM_CFG_ALLOC_EN` is `DEF_ENABLED` in `app_cfg.h` (see section 4-4).

NOTES / WARNINGS

None.

EXAMPLE USAGE

```
void      *pmem_blk;
CPU_SIZE_T  octets_reqd;
LIB_ERR    err;

pmem_blk = Mem_HeapAlloc( (CPU_SIZE_T) 100u,
                          (CPU_SIZE_T)  4u,
                          (CPU_SIZE_T)&octets_reqd,
                          (LIB_ERR *)&err);

if (err != LIB_ERR_NONE) {
    printf("COULD NOT GET MEMORY BLOCK FROM HEAP.");
}
```

4-4-3 Mem_HeapGetSizeRem()

Gets the remaining heap memory pool size available to allocate.

FILES

lib_mem.h/lib_mem.c

PROTOTYPE

```
CPU_SIZE_T Mem_HeapGetSizeRem (CPU_SIZE_T align,  
                                LIB_ERR    *perr);
```

ARGUMENTS

align Desired word boundary alignment (in octets) to return remaining memory size from.

perr Pointer to variable that will receive the return error code from this function:

```
LIB_MEM_ERR_NONE  
LIB_MEM_ERR_NULL_PTR  
LIB_MEM_ERR_INVALID_POOL  
LIB_MEM_ERR_INVALID_MEM_ALIGN
```

RETURNED VALUE

Remaining heap memory size (in octets), if no errors;

0, otherwise.

REQUIRED CONFIGURATION

Available only if `LIB_MEM_CFG_ALLOC_EN` is `DEF_ENABLED` in `app_cfg.h` (see section 4-4).

NOTES / WARNINGS

None.

EXAMPLE USAGE

```
void      *pmem_blk;
CPU_SIZE_T  octets_reqd;
CPU_SIZE_T  size_rem;
LIB_ERR     err;

pmem_blk = Mem_HeapAlloc((CPU_SIZE_T) 100u,
                        (CPU_SIZE_T) 4u,
                        (CPU_SIZE_T)&octets_reqd,
                        (LIB_ERR *)&err);
if (err == LIB_ERR_NONE) {
    size_rem = Mem_HeapGetSizeRem(4u, &err);
    if (err == LIB_ERR_NONE) {
        printf("%u more octets available.", size_rem);
    }
}
```

4-4-4 Mem_SegGetSizeRem()

Gets a memory pool's remaining segment size available to allocate.

FILES

lib_mem.h/lib_mem.c

PROTOTYPE

```
CPU_SIZE_T Mem_SegGetSizeRem (MEM_POOL *pmem_pool,  
                               CPU_SIZE_T align,  
                               LIB_ERR *perr);
```

ARGUMENTS

pmem_pool Pointer to a memory pool structure.

align Desired word boundary alignment (in octets) to return remaining memory size from.

perr Pointer to variable that will receive the return error code from this function:

```
LIB_MEM_ERR_NONE  
LIB_MEM_ERR_NULL_PTR  
LIB_MEM_ERR_INVALID_POOL  
LIB_MEM_ERR_INVALID_MEM_ALIGN
```

RETURNED VALUE

Remaining memory segment size (in octets), if no errors;

0, otherwise.

REQUIRED CONFIGURATION

Available only if `LIB_MEM_CFG_ALLOC_EN` is `DEF_ENABLED` in `app_cfg.h` (see section 4-4).

NOTES / WARNINGS

Remaining size of memory segments returned from either memory segment's configured dedicated memory, if any, or heap memory pool, otherwise.

EXAMPLE USAGE

```
MEM_POOL    AppMemPoolFromHeap;
MEM_POOL    AppMemPoolFromUserMemSeg;
CPU_SIZE_T  octets_reqd;
CPU_SIZE_T  size_rem;
LIB_ERR     err;

Mem_PoolCreate( (MEM_POOL      *) &AppMemPoolFromHeap,
                (void          *) 0,                /* Create pool from heap ... */
                (CPU_SIZE_T    ) 0u,
                (MEM_POOL_BLK_QTY) 10u,             /* ... with 10 blocks ... */
                (CPU_SIZE_T    ) 100u,             /* ... of 100 octets each ... */
                (CPU_SIZE_T    ) 4u,               /* ... and align each block to a 4-byte boundary. */
                (CPU_SIZE_T    *) &octets_reqd,
                (LIB_ERR        *) &err);

if (err == LIB_ERR_NONE) {
    size_rem = Mem_SegGetSizeRem(&AppMemPoolFromHeap, 4u, &err);
    if (err == LIB_ERR_NONE) {
        printf("%u more octets available.", size_rem);
    }
}

Mem_PoolCreate( (MEM_POOL      *) &AppMemPoolFromUserMemSeg,
                (void          *) 0x21000000,       /* Create pool from memory at 0x21000000 ... */
                (CPU_SIZE_T    ) 10000u,           /* ... from a 10000-octet segment ... */
                (MEM_POOL_BLK_QTY) 10u,             /* ... with 10 blocks ... */
                (CPU_SIZE_T    ) 100u,             /* ... of 100 octets each ... */
                (CPU_SIZE_T    ) 4u,               /* ... and align each block to a 4-byte boundary. */
                (CPU_SIZE_T    *) &octets_reqd,
                (LIB_ERR        *) &err);

if (err == LIB_ERR_NONE) {
    size_rem = Mem_SegGetSizeRem(&AppMemPoolFromUserMemSeg, 4u, &err);
    if (err == LIB_ERR_NONE) {
        printf("%u more octets available.", size_rem);
    }
}
}
```

4-4-5 Mem_PoolClr()

Clears a memory pool by setting all memory pool controls to their uninitialized values.

FILES

lib_mem.h/lib_mem.c

PROTOTYPE

```
void Mem_PoolClr (MEM_POOL *pmem_pool,  
                  LIB_ERR  *perr);
```

ARGUMENTS

pmem_pool Pointer to a memory pool structure to clear.

perr Pointer to variable that will receive the return error code from this function:

LIB_MEM_ERR_NONE
LIB_MEM_ERR_NULL_PTR

RETURNED VALUE

None.

REQUIRED CONFIGURATION

Available only if LIB_MEM_CFG_ALLOC_EN is DEF_ENABLED in app_cfg.h (see section 4-4).

NOTES / WARNINGS

pmem_pool must be passed a valid pointer to the address of a declared **MEM_POOL** variable.

EXAMPLE USAGE

```
MEM_POOL  AppMemPool;
LIB_ERR   err;

Mem_PoolClr(&AppMemPool, &err); /* Clear memory pool. */

if (err != LIB_ERR_NONE) {
    printf("COULD NOT CLEAR MEMORY POOL.");
}
```

4-4-6 Mem_PoolCreate()

Creates and initializes a memory pool.

FILES

lib_mem.h/lib_mem.c

PROTOTYPE

```
void Mem_PoolCreate (MEM_POOL      *pmem_pool,  
                    void           *pmem_base_addr,  
                    CPU_SIZE_T     mem_size,  
                    MEM_POOL_BLK_QTY blk_nbr,  
                    CPU_SIZE_T     blk_size,  
                    CPU_SIZE_T     blk_align,  
                    CPU_SIZE_T     *pocets_reqd,  
                    LIB_ERR         *perr);
```

ARGUMENTS

<code>pmem_pool</code>	Pointer to a memory pool structure to create.
<code>pmem_base_addr</code>	Memory pool base address: NULL address Memory pool allocated from general-purpose heap; Non- NULL address Memory pool allocated from dedicated memory specified by non- NULL base address.
<code>mem_size</code>	Size of memory pool segment (in octets).
<code>blk_nbr</code>	Number of memory pool blocks to create.
<code>blk_size</code>	Size of memory pool blocks to create (in octets).
<code>blk_align</code>	Alignment of memory pool blocks to create (in octets).

`poctets_reqd` Pointer to a variable to ...

Return the number of octets required to successfully allocate the memory pool, if any errors;

Return 0, otherwise.

`perr` Pointer to variable that will receive the return error code from this function:

```
LIB_MEM_ERR_NONE
LIB_MEM_ERR_NULL_PTR
LIB_MEM_ERR_HEAP_NOT_FOUND
LIB_MEM_ERR_HEAP_EMPTY
LIB_MEM_ERR_HEAP_OVF
LIB_MEM_ERR_SEG_EMPTY
LIB_MEM_ERR_SEG_OVF
LIB_MEM_ERR_INVALID_SEG_SIZE
LIB_MEM_ERR_INVALID_SEG_OVERLAP
LIB_MEM_ERR_INVALID_BLK_NBR
LIB_MEM_ERR_INVALID_BLK_SIZE
LIB_MEM_ERR_INVALID_BLK_ALIGN
```

RETURNED VALUE

None.

REQUIRED CONFIGURATION

Available only if `LIB_MEM_CFG_ALLOC_EN` is `DEF_ENABLED` in `app_cfg.h` (see section 4-4).

NOTES / WARNINGS

`pmem_pool` must be passed a valid pointer to the address of a declared `MEM_POOL` variable.

EXAMPLE USAGE

```
MEM_POOL    AppMemPoolFromHeap;
MEM_POOL    AppMemPoolFromUserMemSeg;
CPU_SIZE_T  octets_reqd;
LIB_ERR     err;

Mem_PoolCreate((MEM_POOL    *)&AppMemPoolFromHeap,
               (void        *) 0,                /* Create pool from heap ... */
               (CPU_SIZE_T  ) 0u,
               (MEM_POOL_BLK_QTY) 10u,           /* ... with 10 blocks ... */
               (CPU_SIZE_T  ) 100u,             /* ... of 100 octets each ... */
               (CPU_SIZE_T  ) 4u,               /* ... and align each block to a 4-byte boundary. */
               (CPU_SIZE_T  *)&octets_reqd,
               (LIB_ERR     *)&err);

if (err != LIB_ERR_NONE) {
    printf("COULD NOT CREATE MEMORY POOL.");
    if (err == LIB_MEM_ERR_HEAP_EMPTY) {
        printf("Heap empty ... %u more octets needed.", octets_reqd);
    }
    return;
}

Mem_PoolCreate((MEM_POOL    *)&AppMemPoolFromUserMemSeg,
               (void        *)0x21000000,        /* Create pool from memory at 0x21000000 ... */
               (CPU_SIZE_T  )10000u,            /* ... from a 10000-octet segment ... */
               (MEM_POOL_BLK_QTY) 10u,           /* ... with 10 blocks ... */
               (CPU_SIZE_T  ) 100u,             /* ... of 100 octets each ... */
               (CPU_SIZE_T  ) 4u,               /* ... and align each block to a 4-byte boundary. */
               (CPU_SIZE_T  *)&octets_reqd,
               (LIB_ERR     *)&err);

if (err != LIB_ERR_NONE) {
    printf("COULD NOT CREATE MEMORY POOL.");
    if (err == LIB_MEM_ERR_HEAP_EMPTY) {
        printf("Heap empty ... %u more octets needed.", octets_reqd);
    } else if (err == LIB_MEM_ERR_SEG_EMPTY) {
        printf("Segment empty ... %u more octets needed.", octets_reqd);
    }
    return;
}
```

4-4-7 Mem_PoolBlkGetNbrAvail()

Gets a memory pool's remaining number of blocks available to allocate.

FILES

lib_mem.h/lib_mem.c

PROTOTYPE

```
MEM_POOL_BLK_QTY Mem_PoolBlkGetNbrAvail (MEM_POOL *pmem_pool,  
                                           LIB_ERR *perr)
```

ARGUMENTS

pmem_pool Pointer to a memory pool structure.

perr Pointer to variable that will receive the return error code from this function:

```
LIB_MEM_ERR_NONE  
LIB_MEM_ERR_NULL_PTR  
LIB_MEM_ERR_INVALID_POOL
```

RETURNED VALUE

Remaining memory pool blocks, if no errors;

0, otherwise.

REQUIRED CONFIGURATION

Available only if `LIB_MEM_CFG_ALLOC_EN` is `DEF_ENABLED` in `app_cfg.h` (see section 4-4).

NOTES / WARNINGS

`Mem_PoolBlkGetNbrAvail()` **ONLY** supports non-heap memory pools. `Mem_HeapGetSizeRem()/Mem_SegGetSizeRem()` should be used for heap memory pool/segment (see section 4-4-3 and section 4-4-4).

EXAMPLE USAGE

```
MEM_POOL      AppMemPool;
CPU_SIZE_T    octets_reqd;
void          *pmem_blk;
MEM_POOL_BLK_QTY nbr_blk_rem;
LIB_ERR       err;

Mem_PoolCreate( (MEM_POOL      *)&AppMemPool,
                (void          *) 0,           /* Create pool from heap ... */
                (CPU_SIZE_T    ) 0u,           /* ... with 10 blocks ... */
                (MEM_POOL_BLK_QTY) 10u,         /* ... of 100 octets each ... */
                (CPU_SIZE_T    ) 100u,         /* ... and align each block to a 4-byte boundary. */
                (CPU_SIZE_T    *) &octets_reqd,
                (LIB_ERR       *) &err);

if (err != LIB_ERR_NONE) {
    printf("COULD NOT CREATE MEMORY POOL.");
    return;
}

/* Get an 80-byte memory block from the pool. */
pmem_blk = Mem_PoolBlkGet(&AppMemPool, 80u, &err);
if (err != LIB_ERR_NONE) {
    printf("COULD NOT GET MEMORY BLOCK FROM MEMORY POOL.");
    return;
}

/* Get number of remaining memory pool blocks. */
nbr_blk_rem = Mem_PoolBlkGetNbrAvail(&AppMemPool, &err);
if (err == LIB_ERR_NONE) {
    printf("%u more blocks available.", nbr_blk_rem);
}
```

4-4-8 Mem_PoolBlkGet()

Gets a memory block from memory pool.

FILES

lib_mem.h/lib_mem.c

PROTOTYPE

```
void *Mem_PoolBlkGet (MEM_POOL    *pmem_pool,  
                     CPU_SIZE_T   size,  
                     LIB_ERR      *perr);
```

ARGUMENTS

pmem_pool Pointer to memory pool to get memory block from.

size Size of requested memory (in octets).

perr Pointer to variable that will receive the return error code from this function:

```
LIB_MEM_ERR_NONE  
LIB_MEM_ERR_NULL_PTR  
LIB_MEM_ERR_POOL_EMPTY  
LIB_MEM_ERR_INVALID_POOL  
LIB_MEM_ERR_INVALID_BLK_IDX  
LIB_MEM_ERR_INVALID_BLK_SIZE
```

RETURNED VALUE

Pointer to memory block, if no errors;

Pointer to `NULL`, otherwise.

REQUIRED CONFIGURATION

Available only if `LIB_MEM_CFG_ALLOC_EN` is `DEF_ENABLED` in `app_cfg.h` (see section 4-4).

NOTES / WARNINGS

None.

EXAMPLE USAGE

```
MEM_POOL      AppMemPool;
CPU_SIZE_T    octets_reqd;
void          *pmem_blk;
LIB_ERR       err;

Mem_PoolCreate( (MEM_POOL      *)&AppMemPool,
                (void          *) 0,           /* Create pool from heap ... */
                (CPU_SIZE_T    ) 0u,
                (MEM_POOL_BLK_QTY) 10u,       /* ... with 10 blocks ... */
                (CPU_SIZE_T    ) 100u,       /* ... of 100 octets each ... */
                (CPU_SIZE_T    ) 4u,         /* ... and align each block to a 4-byte boundary. */
                (CPU_SIZE_T    *)&octets_reqd,
                (LIB_ERR       *)&err);

if (err != LIB_ERR_NONE) {
    printf("COULD NOT CREATE MEMORY POOL.");
    if (err == LIB_MEM_ERR_HEAP_EMPTY) {
        printf("Heap empty ... %u more octets needed.", octets_reqd);
    }
    return;
}

/* Get an 80-byte memory block from the pool. */
pmem_blk = Mem_PoolBlkGet(&AppMemPool, 80u, &err);
if (err != LIB_ERR_NONE) {
    printf("COULD NOT GET MEMORY BLOCK FROM MEMORY POOL.");
    return;
}
```

4-4-9 Mem_PoolBlkGetUsedAtIx()

Gets a used memory block from memory pool, by index.

FILES

lib_mem.h/lib_mem.c

PROTOTYPE

```
void *Mem_PoolBlkGetUsedByIx (MEM_POOL    *pmem_pool,  
                             MEM_POOL_IX  used_ix,  
                             LIB_ERR      *perr);
```

ARGUMENTS

pmem_pool Pointer to memory pool to get memory block from.

used_ix Index of the used memory block to get.

perr Pointer to variable that will receive the return error code from this function:

```
LIB_MEM_ERR_NONE  
LIB_MEM_ERR_NULL_PTR  
LIB_MEM_ERR_POOL_FULL  
LIB_MEM_ERR_INVALID_POOL  
LIB_MEM_ERR_INVALID_BLK_IX
```

RETURNED VALUE

Pointer to memory block, if no errors;

Pointer to NULL, otherwise.

REQUIRED CONFIGURATION

Available only if LIB_MEM_CFG_ALLOC_EN is DEF_ENABLED in app_cfg.h (see section 4-4).

NOTES / WARNINGS

None.

4-4-10 Mem_PoolBlkFree()

Frees a memory block back to memory pool.

FILES

lib_mem.h/lib_mem.c

PROTOTYPE

```
void Mem_PoolBlkFree (MEM_POOL *pmem_pool,  
                     void *pmem_blk,  
                     LIB_ERR *perr);
```

ARGUMENTS

pmem_pool Pointer to memory pool to free memory block to.

pmem_blk Pointer to memory block address to free.

perr Pointer to variable that will receive the return error code from this function:

```
LIB_MEM_ERR_NONE  
LIB_MEM_ERR_NULL_PTR  
LIB_MEM_ERR_POOL_FULL  
LIB_MEM_ERR_INVALID_POOL  
LIB_MEM_ERR_INVALID_BLK_ADDR  
LIB_MEM_ERR_INVALID_BLK_ADDR_IN_POOL
```

RETURNED VALUE

None.

REQUIRED CONFIGURATION

Available only if `LIB_MEM_CFG_ALLOC_EN` is `DEF_ENABLED` in `app_cfg.h` (see section 4-4).

NOTES / WARNINGS

None.

EXAMPLE USAGE

```
MEM_POOL      AppMemPool;
CPU_SIZE_T    octets_reqd;
void          *pmem_blk;
LIB_ERR       err;

Mem_PoolCreate( (MEM_POOL      *)&AppMemPool,
                (void          *) 0,           /* Create pool from heap ... */
                (CPU_SIZE_T    ) 0u,
                (MEM_POOL_BLK_QTY) 10u,        /* ... with 10 blocks ... */
                (CPU_SIZE_T    ) 100u,        /* ... of 100 octets each ... */
                (CPU_SIZE_T    ) 4u,          /* ... and align each block to a 4-byte boundary. */
                (CPU_SIZE_T    *)&octets_reqd,
                (LIB_ERR        *)&err);

if (err != LIB_ERR_NONE) {
    printf("COULD NOT CREATE MEMORY POOL.");
    if (err == LIB_MEM_ERR_HEAP_EMPTY) {
        printf("Heap empty ... %u more octets needed.", octets_reqd);
    }
    return;
}

/* Get an 80-byte memory block from the pool. */
pmem_blk = Mem_PoolBlkGet(&AppMemPool, 80u, &err);
if (err != LIB_ERR_NONE) {
    printf("COULD NOT GET MEMORY BLOCK FROM MEMORY POOL.");
    return;
}

/* Free 80-byte memory block back to pool. */
Mem_PoolBlkFree(&AppMemPool, pmem_blk, &err);
if (err != LIB_ERR_NONE) {
    printf("COULD NOT FREE MEMORY BLOCK TO MEMORY POOL.");
    return;
}
```

4-4-11 Mem_PoolBlkIdxGet()

Gets index of a memory block in a memory pool.

FILES

lib_mem.h/lib_mem.c

PROTOTYPE

```
void Mem_PoolBlkIdxGet (MEM_POOL *pmem_pool,  
                        void *pmem_blk,  
                        LIB_ERR *perr);
```

ARGUMENTS

pmem_pool Pointer to memory pool.

pmem_blk Pointer to memory block to get index for.

perr Pointer to variable that will receive the return error code from this function:

```
LIB_MEM_ERR_NONE  
LIB_MEM_ERR_NULL_PTR  
LIB_MEM_ERR_POOL_EMPTY  
LIB_MEM_ERR_INVALID_POOL  
LIB_MEM_ERR_INVALID_BLK_ADDR  
LIB_MEM_ERR_INVALID_BLK_ADDR_IN_POOL
```

RETURNED VALUE

Index of the memory block.

REQUIRED CONFIGURATION

Available only if `LIB_MEM_CFG_ALLOC_EN` is `DEF_ENABLED` in `app_cfg.h` (see section 4-4).

NOTES / WARNINGS

None.

4-5 MEMORY LIBRARY OPTIMIZATION

All μC/LIB memory functions have been C-optimized for improved run-time performance, independent of processor or compiler optimizations. This is accomplished by performing memory operations on CPU-aligned word boundaries whenever possible.

In addition, some μC/LIB memory functions have been assembly-optimized for certain processors/compilers. If these optimizations are defined in assembly files found in appropriate port directories for each specific processor/compiler combination. See Figure 4-1 for an example port directory:

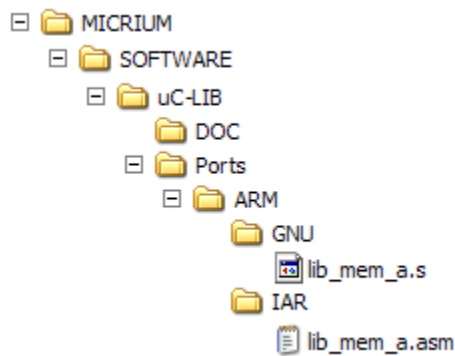


Figure 4-1 **μC/LIB Example Port Directory**

µC/LIB String Library

µC/LIB contains library functions that replace standard library string functions such as `strlen()`, `strcpy()`, `strcmp()`, etc. These functions are defined in `lib_str.c`.

5-1 STRING LIBRARY CONFIGURATION

The following µC/LIB string library configurations may be optionally configured in `app_cfg.h` :

<code>LIB_STR_CFG_FP_EN</code>	Enable floating-point string conversion functions (see section 5-2-19). This feature may be configured to either <code>DEF_DISABLED</code> or <code>DEF_ENABLED</code> .
<code>LIB_STR_CFG_FP_MAX_NBR_DIG_SIG</code>	Configure the maximum number of significant digits to calculate and/or display for floating point string functions.

5-2 STRING LIBRARY FUNCTIONS

5-2-1 Str_Len()

Determines the length of a string.

FILES

lib_str.h/lib_str.c

PROTOTYPE

```
CPU_SIZE_T Str_Len (const CPU_CHAR *pstr);
```

ARGUMENTS

pstr Pointer to the string.

RETURNED VALUE

Length of string, in number of characters, before, but not including, the terminating **NULL** character.

REQUIRED CONFIGURATION

None.

NOTES / WARNINGS

String buffer not modified.

String length calculation terminates if string pointer points to or overlaps the **NULL** address.

EXAMPLE USAGE

```
CPU_SIZE_T len;  
  
len = Str_Len("SomeString");
```

5-2-2 Str_Len_N()

Determines the length of a string, up to a maximum number of characters.

FILES

lib_str.h/lib_str.c

PROTOTYPE

```
CPU_SIZE_T Str_Len_N (const CPU_CHAR *pstr,  
                      CPU_SIZE_T len_max);
```

ARGUMENTS

pstr Pointer to the string.

len_max Maximum number of string characters to search.

RETURNED VALUE

Length of string, in number of characters, before, but not including, the terminating **NULL** character; if terminating **NULL** character found;

Maximum number of characters to search, if terminating **NULL** character not found.

REQUIRED CONFIGURATION

None.

NOTES / WARNINGS

String buffer not modified.

The maximum number of characters to search does not include the terminating **NULL** character. Therefore, if **Str_Len()** returns the maximum number of search characters, then the string is **not** **NULL**-terminated within the maximum number of search characters.

String length calculation terminates if string pointer points to or overlaps the **NULL** address.

EXAMPLE USAGE

```
CPU_SIZE_T  len;

len = Str_Len_N("SomeString", MAX_SIZE);
if (len >= MAX_SIZE) {
    printf("STRING IS TOO LONG!");
}
```

5-2-3 Str_Copy()

Copies string character values from one string memory buffer to another memory buffer.

FILES

lib_str.h/lib_str.c

PROTOTYPE

```
CPU_CHAR *Str_Copy (    CPU_CHAR *pstr_dest,  
                        const CPU_CHAR *pstr_src);
```

ARGUMENTS

pstr_dest Pointer to the string memory buffer to copy string characters into.

pstr_src Pointer to the string memory buffer to copy string characters from.

RETURNED VALUE

Pointer to copied destination string, if no errors;

Pointer to **NULL**, otherwise.

REQUIRED CONFIGURATION

None.

NOTES / WARNINGS

Destination buffer size (**pstr_dest**) is not validated; buffer overruns must be prevented by caller. Destination buffer size **must** be large enough to accomodate the entire source string size including its terminating **NULL** character.

String copy terminates if either string pointer points to or overlaps the **NULL** address.

EXAMPLE USAGE

```
CPU_CHAR  AppBuf[20];
CPU_CHAR  *pstr;

pstr = Str_Copy(&AppBuf[0], "Hello World!");
if (pstr == (CPU_CHAR *)0) {
    printf("STRING COPY FAILED!");
}
```

5-2-4 Str_Copy_N()

Copies string character values from one string memory buffer to another memory buffer, up to a maximum number of characters.

FILES

lib_str.h/lib_str.c

PROTOTYPE

```
CPU_CHAR *Str_Copy_N (      CPU_CHAR *pstr_dest,  
                           const CPU_CHAR *pstr_src,  
                           CPU_SIZE_T len_max);
```

ARGUMENTS

pstr_dest Pointer to the string memory buffer to copy string characters into.

pstr_src Pointer to the string memory buffer to copy string characters from.

len_max Maximum number of string characters to copy.

RETURNED VALUE

Pointer to copied destination string, if no errors;

Pointer to **NULL**, otherwise.

REQUIRED CONFIGURATION

None.

NOTES / WARNINGS

The maximum number of characters copied **may and should** include the terminating **NULL** character. Note that IEEE Std 1003.1, 2004 Edition, Section ‘**strncpy()**’: APPLICATION USAGE’ states that “if there is no null byte in the first [**len_max**] bytes of the array pointed to by [**pstr_src**], the result is not null-terminated”.

Destination buffer size (**pstr_dest**) is not validated; buffer overruns must be prevented by caller. Destination buffer size **should** be large enough to accomodate the entire source string size including its terminating **NULL** character.

String copy terminates if either string pointer points to or overlaps the **NULL** address.

EXAMPLE USAGE

```
CPU_CHAR  AppBuf[20];
CPU_CHAR  *pstr;

pstr = Str_Copy_N(&AppBuf[0], "Hello World!", (sizeof(AppBuf)));
if (pstr == (CPU_CHAR *)0) {
    printf("STRING COPY FAILED!");
}
```

5-2-5 str_Cat()

Concatenates a string to the end of another string.

FILES

lib_str.h/lib_str.c

PROTOTYPE

```
CPU_CHAR *Str_Cat (      CPU_CHAR *pstr_dest,  
                        const CPU_CHAR *pstr_cat);
```

ARGUMENTS

pstr_dest Pointer to the string memory buffer to append string characters into.

pstr_cat Pointer to the string to concatenate onto the destination string.

RETURNED VALUE

Pointer to concatenated destination string, if no errors;

Pointer to **NULL**, otherwise.

REQUIRED CONFIGURATION

None.

NOTES / WARNINGS

Destination buffer size (**pstr_dest**) is not validated; buffer overruns must be prevented by caller. IEEE Std 1003.1, 2004 Edition, Section ‘**strcat()** : DESCRIPTION’ states that “the initial byte of [**pstr_cat**] overwrites the null byte at the end of [**pstr_dest**]” and a “terminating null byte” is appended “to the end of the string pointed to by [**pstr_dest**]”.

Therefore, the destination buffer size ***must*** be large enough to accomodate the original destination string size plus the entire concatenated string size, but including only a single terminating **NULL** character.

String concatenation terminates if either string pointer points to or overlaps the **NULL** address.

EXAMPLE USAGE

```
CPU_CHAR    AppBuf[30];
CPU_CHAR    *pstr;

pstr = Str_Copy(&AppBuf[0], "Hello  World!");
if (pstr != (CPU_CHAR *)0) {
    pstr = Str_Cat(&AppBuf[0], "Goodbye World!");
}

if (pstr == (CPU_CHAR *)0) {
    printf("STRING COPY/CONCATENATION FAILED!");
}
```

5-2-6 str_Cat_N()

Concatenates a string to the end of another string, up to a maximum number of characters.

FILES

lib_str.h/lib_str.c

PROTOTYPE

```
CPU_CHAR *Str_Cat_N (    CPU_CHAR *pstr_dest,  
                        const CPU_CHAR *pstr_cat,  
                        CPU_SIZE_T len_max);
```

ARGUMENTS

pstr_dest Pointer to the string memory buffer to append string characters into.

pstr_cat Pointer to the string to concatenate onto the destination string.

len_max Maximum number of string characters to concatenate.

RETURNED VALUE

Pointer to concatenated destination string, if no errors;

Pointer to **NULL**, otherwise.

REQUIRED CONFIGURATION

None.

NOTES / WARNINGS

The maximum number of characters concatenated does not include the terminating **NULL** character. Note that IEEE Std 1003.1, 2004 Edition, Section ‘**strncat()** : DESCRIPTION’ states that “the **strncat()** function shall append ... the array pointed to by [**pstr_cat**] to the end of the string pointed to by [**pstr_dest**]” but “not more than [**len_max**] bytes.”

Destination buffer size (**pstr_dest**) is not validated; buffer overruns must be prevented by caller. IEEE Std 1003.1, 2004 Edition, Section ‘**strncat()** : DESCRIPTION’ states that “the initial byte of [**pstr_cat**] overwrites the null byte at the end of [**pstr_dest**]” and “a terminating null byte is always appended to the result”. Therefore, the destination buffer size **should** be large enough to accomodate the original destination string size plus the entire concatenated string size, but including only a single terminating **NULL** character.

String concatenation terminates if either string pointer points to or overlaps the **NULL** address.

EXAMPLE USAGE

```
CPU_CHAR    AppBuf[30];
CPU_CHAR    *pstr;
CPU_SIZE_T  len;

pstr = Str_Copy_N(&AppBuf[0], "Hello World!", sizeof(AppBuf));
if (pstr != (CPU_CHAR *)0) {
    len = Str_Len_N(&AppBuf[0], sizeof(AppBuf));
    if ((len + sizeof((CPU_CHAR)'\0'))      /* If 'Hello' string including its NULL character ... */
        < sizeof(AppBuf)) {                /* ... fits entirely in AppBuf[], ... */
        pstr = Str_Cat_N(&AppBuf[0],
                        "Goodbye World!", /* ... concatenate 'Goodbye' string ... */
                        /* ... while limiting to remaining AppBuf[] size. */
                        (sizeof(AppBuf) - (len + sizeof((CPU_CHAR)'\0'))));
    } else {
        printf("COPY STRING IS TOO LONG!");
    }
}

if (pstr == (CPU_CHAR *)0) {
    printf("STRING COPY/CONCATENATION FAILED!");
}
```

5-2-7 str_Cmp()

Determines if two strings are identical.

FILES

lib_str.h/lib_str.c

PROTOTYPE

```
CPU_INT16S str_Cmp (const CPU_CHAR *p1_str,  
                    const CPU_CHAR *p2_str);
```

ARGUMENTS

p1_str Pointer to the first string.

p2_str Pointer to the second string.

RETURNED VALUE

Zero value, if strings are identical; i.e., both strings are identical for the specified length of characters.

Positive value, if **p1_str** is greater than **p2_str**; i.e., **p1_str** points to a character of higher value than **p2_str** for the first non-matching character found.

Negative value, if **p1_str** is less than **p2_str**; i.e., **p1_str** points to a character of lesser value than **p2_str** for the first non-matching character found.

REQUIRED CONFIGURATION

None.

NOTES / WARNINGS

String buffers not modified.

String comparison terminates if either string pointer points to or overlaps the **NULL** address.

Since 16-bit signed arithmetic is performed to calculate a non-identical comparison return value, `CPU_CHAR` native data type size **must** be 8-bit.

EXAMPLE USAGE

```
CPU_INT16S  cmp;  
  
cmp = Str_Cmp("Hello World!", "Hello World.");
```

5-2-8 str_Cmp_N()

Determines if two strings are identical for up to a specified length of characters.

FILES

lib_str.h/lib_str.c

PROTOTYPE

```
CPU_INT16S  Str_Cmp_N (const CPU_CHAR  *p1_str,
                        const CPU_CHAR  *p2_str,
                        CPU_SIZE_T      len_max);
```

ARGUMENTS

p1_str Pointer to the first string.

p2_str Pointer to the second string.

len_max Maximum number of string characters to compare.

RETURNED VALUE

Zero value, if strings are identical; i.e., both strings are identical for the specified length of characters.

Positive value, if **p1_str** is greater than **p2_str**; i.e., **p1_str** points to a character of higher value than **p2_str** for the first non-matching character found.

Negative value, if **p1_str** is less than **p2_str**; i.e., **p1_str** points to a character of lesser value than **p2_str** for the first non-matching character found.

REQUIRED CONFIGURATION

None.

NOTES / WARNINGS

String buffers not modified.

String comparison terminates if either string pointer points to or overlaps the **NULL** address.

Since 16-bit signed arithmetic is performed to calculate a non-identical comparison return value, **CPU_CHAR** native data type size **must** be 8-bit.

EXAMPLE USAGE

```
CPU_INT16S  cmp;  
  
cmp = Str_Cmp_N("Hello World!", "Hello World.", 11u);
```

5-2-9 Str_CmpIgnoreCase()

Determines if two strings are identical, ignoring case.

FILES

lib_str.h/lib_str.c

PROTOTYPE

```
CPU_INT16S Str_CmpIgnoreCase (const CPU_CHAR *p1_str,  
                               const CPU_CHAR *p2_str);
```

ARGUMENTS

p1_str Pointer to the first string.

p2_str Pointer to the second string.

RETURNED VALUE

Zero value, if strings are identical (ignoring case); i.e., both strings are identical (ignoring case) for the specified length of characters.

Positive value, if **p1_str** is greater than **p2_str**, ignoring case; i.e., **p1_str** points to a character (when converted to lower case) of higher value than **p2_str** for the first non-matching character found.

Negative value, if **p1_str** is less than **p2_str**, ignoring case; i.e., **p1_str** points to a character (when converted to lower case) of lesser value than **p2_str** for the first non-matching character found.

REQUIRED CONFIGURATION

None.

NOTES / WARNINGS

`Str_CmpIgnoreCase()` behaves as if the two strings were converted to lower case and then compared with `Str_Cmp()`.

String buffers not modified.

String comparison terminates if either string pointer points to or overlaps the `NULL` address.

Since 16-bit signed arithmetic is performed to calculate a non-identical comparison return value, `CPU_CHAR` native data type size **must** be 8-bit.

EXAMPLE USAGE

```
CPU_INT16S  cmp;

cmp = Str_CmpIgnoreCase("Hello World!", "hElLo WoRlD.");
```

5-2-10 Str_CmpIgnoreCase_N()

Determines if two strings are identical for up to a specified length of characters, ignoring case.

FILES

lib_str.h/lib_str.c

PROTOTYPE

```
CPU_INT16S Str_CmpIgnoreCase_N (const CPU_CHAR *p1_str,  
                                const CPU_CHAR *p2_str,  
                                CPU_SIZE_T len_max);
```

ARGUMENTS

p1_str Pointer to the first string.

p2_str Pointer to the second string.

len_max Maximum number of string characters to compare.

RETURNED VALUE

Zero value, if strings are identical (ignoring case); i.e., both strings are identical (ignoring case) for the specified length of characters.

Positive value, if **p1_str** is greater than **p2_str**, ignoring case; i.e., **p1_str** points to a character (when converted to lower case) of higher value than **p2_str** for the first non-matching character found.

Negative value, if **p1_str** is less than **p2_str**, ignoring case; i.e., **p1_str** points to a character (when converted to lower case) of lesser value than **p2_str** for the first non-matching character found.

REQUIRED CONFIGURATION

None.

NOTES / WARNINGS

`Str_CmpIgnoreCase_N()` behaves as if the two strings were converted to lower case and then compared with `Str_Cmp_N()`.

String buffers not modified.

String comparison terminates if either string pointer points to or overlaps the `NULL` address.

Since 16-bit signed arithmetic is performed to calculate a non-identical comparison return value, `CPU_CHAR` native data type size **must** be 8-bit.

EXAMPLE USAGE

```
CPU_INT16S  cmp;  
  
cmp = Str_CmpIgnoreCase_N("Hello World!", "hELLo WoRLD.", 11u);
```

5-2-11 str_Char()

Finds the first occurrence of a specific character in a string.

FILES

lib_str.h/lib_str.c

PROTOTYPE

```
CPU_CHAR *Str_Char (const CPU_CHAR *pstr,  
                    CPU_CHAR srch_char);
```

ARGUMENTS

pstr Pointer to the string to search for the specified character.

srch_char Character to search for in the string.

RETURNED VALUE

Pointer to first occurrence of character in string, if no errors;

Pointer to **NULL**, otherwise.

REQUIRED CONFIGURATION

None.

NOTES / WARNINGS

String buffer not modified.

String search terminates if string pointer points to or overlaps the **NULL** address.

EXAMPLE USAGE

```
CPU_CHAR  *pstr;  
  
pstr = Str_Char("Hello World!", 'l');
```

5-2-12 Str_Char_N()

Finds the first occurrence of a specific character in a string, up to a maximum number of characters.

FILES

lib_str.h/lib_str.c

PROTOTYPE

```
CPU_CHAR *Str_Char_N (const CPU_CHAR *pstr,  
                        CPU_SIZE_T len_max,  
                        CPU_CHAR srch_char);
```

ARGUMENTS

pstr Pointer to the string to search for the specified character.

len_max Maximum number of string characters to search.

srch_char Character to search for in the string.

RETURNED VALUE

Pointer to first occurrence of character in string, if no errors;

Pointer to **NULL**, otherwise.

REQUIRED CONFIGURATION

None.

NOTES / WARNINGS

String buffer not modified.

String search terminates if string pointer points to or overlaps the **NULL** address.

Ideally, `Str_Char_N()`'s `len_max` argument would be the last argument in this function's argument list for consistency with all other custom string library functions. However, the `len_max` argument is sequentially ordered as the second argument to comply with most standard library's `strnchr()` argument list.

EXAMPLE USAGE

```
CPU_CHAR  *pstr;  
  
pstr = Str_Char_N("Hello World!", 5u, 'l');
```

5-2-13 Str_Char_Last()

Finds the last occurrence of a specific character in a string.

FILES

lib_str.h/lib_str.c

PROTOTYPE

```
CPU_CHAR *Str_Char_Last (const CPU_CHAR *pstr,  
                          CPU_CHAR srch_char);
```

ARGUMENTS

pstr Pointer to the string to search for the specified character.

srch_char Character to search for in the string.

RETURNED VALUE

Pointer to last occurrence of character in string, if no errors;

Pointer to **NULL**, otherwise.

REQUIRED CONFIGURATION

None.

NOTES / WARNINGS

String buffer not modified.

String search terminates if string pointer points to or overlaps the **NULL** address.

EXAMPLE USAGE

```
CPU_CHAR  *pstr;  
  
pstr = Str_Char_Last("Hello World!", 'l');
```

5-2-14 Str_Char_Last_N()

Finds the last occurrence of a specific character in a string, up to a maximum number of characters.

FILES

lib_str.h/lib_str.c

PROTOTYPE

```
CPU_CHAR *Str_Char_Last_N (const CPU_CHAR *pstr,  
                           CPU_SIZE_T len_max,  
                           CPU_CHAR srch_char);
```

ARGUMENTS

pstr Pointer to the string to search for the specified character.

len_max Maximum number of string characters to search.

srch_char Character to search for in the string.

RETURNED VALUE

Pointer to last occurrence of character in string, if no errors;

Pointer to **NULL**, otherwise.

REQUIRED CONFIGURATION

None.

NOTES / WARNINGS

String buffer not modified.

String search terminates if string pointer points to or overlaps the **NULL** address.

Ideally, `Str_Char_Last_N()`'s `len_max` argument would be the last argument in this function's argument list for consistency with all other custom string library functions. However, the `len_max` argument is sequentially ordered as the second argument to comply with most standard library's `strnchr()` argument list.

EXAMPLE USAGE

```
CPU_CHAR  *pstr;  
  
pstr = Str_Char_Last_N("Hello World!", 5u, 'l');
```

5-2-15 str_str()

Finds the first occurrence of a specific string within another string.

FILES

lib_str.h/lib_str.c

PROTOTYPE

```
CPU_CHAR *Str_Str (const CPU_CHAR *pstr,  
                  const CPU_CHAR *pstr_srch);
```

ARGUMENTS

pstr Pointer to the string to search for the specified string.

pstr_srch Pointer to the string to search for in the string.

RETURNED VALUE

Pointer to first occurrence of search string in string, if specified string found in search string and no errors.

Pointer to search string, if specified string is zero-length **NULL**-string.

Pointer to **NULL**, otherwise.

REQUIRED CONFIGURATION

None.

NOTES / WARNINGS

String buffers not modified.

String search terminates if string pointer points to or overlaps the **NULL** address.

EXAMPLE USAGE

```
CPU_CHAR  *pstr;  
  
pstr = Str_Str("Hello World!", "lo");
```

5-2-16 Str_Str_N()

Finds the first occurrence of a specific string within another string, up to a maximum number of characters.

FILES

lib_str.h/lib_str.c

PROTOTYPE

```
CPU_CHAR *Str_Str_N (const CPU_CHAR *pstr,  
                     const CPU_CHAR *pstr_srch,  
                     CPU_SIZE_T len_max);
```

ARGUMENTS

pstr Pointer to the string to search for the specified string.

pstr_srch Pointer to the string to search for in the string.

len_max Maximum number of string characters to search.

RETURNED VALUE

Pointer to first occurrence of search string in string, if specified string found in search string and no errors.

Pointer to search string, if specified string is zero-length **NULL**-string.

Pointer to **NULL**, otherwise.

REQUIRED CONFIGURATION

None.

NOTES / WARNINGS

String buffers not modified.

String search terminates if string pointer points to or overlaps the **NULL** address.

EXAMPLE USAGE

```
CPU_CHAR  *pstr;  
  
pstr = Str_Str_N("Hello World!", "lo", 10u);
```

5-2-17 Str_FmtNbr_Int32U()

Converts and formats a 32-bit unsigned integer into a string.

FILES

lib_str.h/lib_str.c

PROTOTYPE

```
CPU_CHAR *Str_FmtNbr_Int32U (CPU_INT32U   nbr,  
                             CPU_INT08U   nbr_dig,  
                             CPU_INT08U   nbr_base,  
                             CPU_CHAR     lead_char,  
                             CPU_BOOLEAN  lower_case,  
                             CPU_BOOLEAN  nul,  
                             CPU_CHAR     *pstr);
```

ARGUMENTS

nbr Number to format into a string.

nbr_dig Number of integer digits to format into the number string.

nbr_base Base of the number to format into the number string.

lead_char Option to prepend a leading character into the formatted number string:

'\0'	Do not prepend leading character to string.
Printable character	Prepend leading character to string.
Unprintable character	Format invalid string.

lower_case Option to format any alphabetic characters (if any) in lower case:

DEF_NO	Format alphabetic characters in upper case.
DEF_YES	Format alphabetic characters in lower case.

nul Option to **NULL**-terminate the formatted number string:

DEF_NO	Do not append terminating NULL -character to string.
DEF_YES	Append terminating NULL -character to string.

pstr Pointer to the string memory buffer to return the formatted number string.

RETURNED VALUE

Pointer to formatted number string, if no errors;

Pointer to **NULL**, otherwise.

REQUIRED CONFIGURATION

None.

NOTES / WARNINGS

The following constants may be used to specify the number of digits to format (**nbr_dig**):

DEF_INT_32U_NBR_DIG_MIN	Minimum number of 32-bit unsigned digits
DEF_INT_32U_NBR_DIG_MAX	Maximum number of 32-bit unsigned digits

The number's base (**nbr_base**) must be between 2 and 36, inclusive. The following constants may be used to specify the number base:

DEF_NBR_BASE_BIN	Base 2
DEF_NBR_BASE_OCT	Base 8
DEF_NBR_BASE_DEC	Base 10
DEF_NBR_BASE_HEX	Base 16

For any unsuccessful string format or errors, an invalid string of question marks ('?') will be formatted, where the number of question marks is determined by the number of digits to format (**nbr_dig**). Also, whenever an invalid string is formatted for any reason, a **NULL** pointer is returned.

If the number of digits to format (**nbr_dig**) is zero; then no formatting is performed except possible **NULL**-termination of the string. Example:

```
nbr      = 23456
nbr_dig  = 0
pstr     = ""
```

If the number of digits to format (**nbr_dig**) is less than the number of significant integer digits of the number to format (**nbr**); then an invalid string is formatted instead of truncating any significant integer digits. Example:

```
nbr      = 23456
nbr_dig  = 3
pstr     = "???"
```

Leading character option (**lead_char**) prepends leading characters prior to the first non-zero significant digit. Leading character must be a printable ASCII character; but must not be a number base digit, with the exception of '0'.

For unsigned integers, the number of leading characters is such that the total number of significant integer digits plus the number of leading characters is equal to the requested number of integer digits to format (**nbr_dig**). Example:

```
nbr      = 23456
nbr_dig  = 7
lead_char = ' '
pstr     = " 23456"
```

If the value of the number to format (**nbr**) is zero and the number of digits to format (**nbr_dig**) is non-zero, but no leading character (**lead_char**) available; then one digit of '0' value is formatted. This is not a leading character; but a single integer digit of '0' value. Example:

```
nbr      = 0
nbr_dig  = 7
lead_char = '\0'
pstr     = "0"
```


When **NULL**-character terminate option (**nul**) is disabled, it prevents overwriting previous character array formatting. **Warning:** Unless **pstr** character array is pre-/post-terminated, if **NULL**-character terminate option is disabled, it will cause character string run-on.

Format buffer size not validated; buffer overruns must be prevented by caller. To prevent character buffer overrun:

Character array size **must** be $\geq (\text{nbr_dig} + 1 \text{ NUL terminator})$ characters

EXAMPLE USAGE

```
CPU_CHAR  AppBuf[20];
CPU_CHAR  *pstr;

pstr = Str_FmtNbr_Int32U( (CPU_INT32U) 12345678u,
                        (CPU_INT08U) 10,
                        (CPU_INT08U) 10,
                        (CPU_CHAR  ) '0',
                        (CPU_BOOLEAN) DEF_NO,
                        (CPU_BOOLEAN) DEF_YES,
                        (CPU_CHAR  *)&AppBuf[0]);
```

5-2-18 Str_FmtNbr_Int32S()

Converts and formats a 32-bit signed integer into a string.

FILES

lib_str.h/lib_str.c

PROTOTYPE

```
CPU_CHAR *Str_FmtNbr_Int32S (CPU_INT32S   nbr,  
                             CPU_INT08U   nbr_dig,  
                             CPU_INT08U   nbr_base,  
                             CPU_CHAR     lead_char,  
                             CPU_BOOLEAN  lower_case,  
                             CPU_BOOLEAN  nul,  
                             CPU_CHAR     *pstr);
```

ARGUMENTS

nbr Number to format into a string.

nbr_dig Number of integer digits to format into the number string.

nbr_base Base of the number to format into the number string.

lead_char Option to prepend a leading character into the formatted number string:

'\0'	Do not prepend leading character to string.
Printable character	Prepend leading character to string.
Unprintable character	Format invalid string.

lower_case Option to format any alphabetic characters (if any) in lower case:

DEF_NO	Format alphabetic characters in upper case.
DEF_YES	Format alphabetic characters in lower case.

nul Option to **NULL**-terminate the formatted number string:

DEF_NO	Do not append terminating NULL -character to string.
DEF_YES	Append terminating NULL -character to string.

pstr Pointer to the string memory buffer to return the formatted number string.

RETURNED VALUE

Pointer to formatted number string, if no errors;

Pointer to **NULL**, otherwise.

REQUIRED CONFIGURATION

None.

NOTES / WARNINGS

The following constants may be used to specify the number of digits to format (**nbr_dig**):

DEF_INT_32S_NBR_DIG_MIN + 1	Minimum number of 32-bit signed digits
DEF_INT_32S_NBR_DIG_MAX + 1	Maximum number of 32-bit signed digits (plus 1 digit for possible negative sign)

The number's base (**nbr_base**) must be between 2 and 36, inclusive. The following constants may be used to specify the number base:

DEF_NBR_BASE_BIN	Base 2
DEF_NBR_BASE_OCT	Base 8
DEF_NBR_BASE_DEC	Base 10
DEF_NBR_BASE_HEX	Base 16

For any unsuccessful string format or errors, an invalid string of question marks ('?') will be formatted, where the number of question marks is determined by the number of digits to format (**nbr_dig**). Also, whenever an invalid string is formatted for any reason, a **NULL** pointer is returned.

If the number of digits to format (**nbr_dig**) is zero; then no formatting is performed except possible **NULL**-termination of the string. Example:

```
nbr      = -23456
nbr_dig  =  0
nbr_base = 10
pstr     = ""
```

If the number of digits to format (**nbr_dig**) is less than the number of significant integer digits of the number to format (**nbr**); then an invalid string is formatted instead of truncating any significant integer digits. Example:

```
nbr      = 23456
nbr_dig  =  3
nbr_base = 10
pstr     = "???"
```

If the number to format (**nbr**) is negative but the number of digits to format (**nbr_dig**) is equal to the number of significant integer digits of the number to format (**nbr**); then an invalid string is formatted instead of truncating the negative sign. Example:

```
nbr      = -23456
nbr_dig  =  5
nbr_base = 10
pstr     = "?????"
```

Leading character option (**lead_char**) prepends leading characters prior to the first non-zero significant digit. Leading character must be a printable ASCII character; but must not be a number base digit, with the exception of '0'.

For signed integers, the number of leading characters is such that the total number of significant integer digits plus the number of leading characters plus possible negative sign character is equal to the requested number of integer digits to format (**nbr_dig**). Examples:

```
nbr      = 23456
nbr_dig  = 7
nbr_base = 10
lead_char = ' '
pstr     = " 23456"

nbr      = -23456
nbr_dig  = 7
nbr_base = 10
lead_char = ' '
pstr     = " -23456"
```

If the value of the number to format (**nbr**) is zero and the number of digits to format (**nbr_dig**) is non-zero, but no leading character (**lead_char**) available; then one digit of '0' value is formatted. This is not a leading character; but a single integer digit of '0' value. Example:

```
nbr      = 0
nbr_dig  = 7
lead_char = '\0'
pstr     = "0"
```

If the number to format (**nbr**) is negative and the leading character (**lead_char**) is a '0' digit; then the negative sign character prefixes all leading characters prior to the formatted number. Examples:

```
nbr      = -23456
nbr_dig  = 8
nbr_base = 10
lead_char = '0'
pstr     = "-0023456"
```

```
nbr          = -43981
nbr_dig      = 8
nbr_base     = 16
lead_char    = '0'
lower_case   = DEF_NO
pstr         = "-000ABCD"
```

If the number to format (**nbr**) is negative and the leading character (**lead_char**) is **not** a '0' digit; then the negative sign character immediately prefixes the most significant digit of the formatted number. Examples:

```
nbr          = -23456
nbr_dig      = 8
nbr_base     = 10
lead_char    = '#'
pstr         = "##-23456"
```

```
nbr          = -43981
nbr_dig      = 8
nbr_base     = 16
lead_char    = '#'
lower_case   = DEF_YES
pstr         = "###-abcd"
```

When **NULL**-character terminate option (**nul**) is disabled, it prevents overwriting previous character array formatting. **Warning:** Unless **pstr** character array is pre-/post-terminated, if **NULL**-character terminate option is disabled, it will cause character string run-on.

Format buffer size not validated; buffer overruns must be prevented by caller. To prevent character buffer overrun:

```
Character array size must be >= (nbr_dig      +
                                1 negative sign +
                                1 NUL terminator) characters
```

EXAMPLE USAGE

```
CPU_CHAR  AppBuf[20];
CPU_CHAR  *pstr;

pstr = Str_FmtNbr_Int32S((CPU_INT32S )-12345678,
                        (CPU_INT08U ) 10,
                        (CPU_INT08U ) 10,
                        (CPU_CHAR   ) '0',
                        (CPU_BOOLEAN) DEF_NO,
                        (CPU_BOOLEAN) DEF_YES,
                        (CPU_CHAR   *)&AppBuf[0]);
```

5-2-19 Str_FmtNbr_32()

Converts and formats a 32-bit floating point number into a string.

FILES

lib_str.h/lib_str.c

PROTOTYPE

```
CPU_CHAR *Str_FmtNbr_32 (CPU_FP32    nbr,  
                          CPU_INT08U  nbr_dig,  
                          CPU_INT08U  nbr_dp,  
                          CPU_CHAR    lead_char,  
                          CPU_BOOLEAN nul,  
                          CPU_CHAR    *pstr);
```

ARGUMENTS

nbr	Number to format into a string.	
nbr_dig	Number of integer	digits to format into the number string.
nbr_dp	Number of decimal	digits to format into the number string.
lead_char	Option to prepend a leading character into the formatted number string:	
	'\0'	Do not prepend leading character to string.
	Printable character	Prepend leading character to string.
	Unprintable character	Format invalid string.
nul	Option to NULL -terminate the formatted number string:	
	DEF_NO	Do not append terminating NULL -character to string.
	DEF_YES	Append terminating NULL -character to string.
pstr	Pointer to the string memory buffer to return the formatted number string.	

RETURNED VALUE

Pointer to formatted number string, if no errors;

Pointer to `NULL`, otherwise.

REQUIRED CONFIGURATION

Available only if `LIB_STR_CFG_FP_EN` is `DEF_ENABLED` in `app_cfg.h` (see section 5-1).

NOTES / WARNINGS

For any unsuccessful string format or errors, an invalid string of question marks ('?') will be formatted, where the number of question marks is determined by the number of digits (`nbr_dig`) and number of decimal point digits (`nbr_dp`) to format. Also, whenever an invalid string is formatted for any reason, a `NULL` pointer is returned.

If the total number of digits to format (`nbr_dig + nbr_dp`) is zero; then no formatting is performed except possible `NULL`-termination of the string. Example:

```
nbr      = -23456.789
nbr_dig  =  0
nbr_dp   =  0
pstr     = ""
```

If the number of digits to format (`nbr_dig`) is less than the number of significant integer digits of the number to format (`nbr`); then an invalid string is formatted instead of truncating any significant integer digits. Example:

```
nbr      = 23456.789
nbr_dig  =  3
nbr_dp   =  2
pstr     = "??????"
```

If the number to format (**nbr**) is negative but the number of digits to format (**nbr_dig**) is equal to the number of significant integer digits of the number to format (**nbr**); then an invalid string is formatted instead of truncating the negative sign. Example:

```
nbr      = -23456.789
nbr_dig  =  5
nbr_dp   =  2
pstr     = "???????"
```

If the number to format (**nbr**) is negative but the number of significant integer digits is zero, and the number of digits to format (**nbr_dig**) is zero but the number of decimal point digits to format (**nbr_dp**) is non-zero; then the negative sign immediately prefixes the decimal point—with no decimal digits formatted, not even a single decimal digit of '0'. Example:

```
nbr      = -0.7895
nbr_dig  =  0
nbr_dp   =  2
pstr     = "-.78"
```

If the number to format (**nbr**) is positive but the number of significant integer digits is zero, and the number of digits to format (**nbr_dig**) is zero but the number of decimal point digits to format (**nbr_dp**) is non-zero; then a single decimal digit of '0' prefixes the decimal point. This '0' digit is used whenever a negative sign is not formatted so that the formatted string's decimal point is not floating, but fixed in the string as the 2nd character. Example:

```
nbr      =  0.7895
nbr_dig  =  0
nbr_dp   =  2
pstr     = "0.78"
```

If the total number of digits to format (`nbr_dig` + `nbr_dp`) is greater than the configured maximum accuracy (`LIB_STR_CFG_FP_MAX_NBR_DIG_SIG`), all digits or decimal places following all significantly-accurate digits of the number to format (`nbr`) will be replaced and formatted with zeros ('0'). Example:

```
nbr           = 123456789.012345
nbr_dig       = 9
nbr_dp        = 6
LIB_STR_CFG_FP_MAX_NBR_DIG_SIG = 7
pstr          = "123456700.000000"
```

Also, if the total number of digits to format (`nbr_dig` + `nbr_dp`) is greater than the maximum accuracy of the CPU's and/or compiler's 32-bit floating-point numbers, digits following all significantly-accurate digits of the number to format (`nbr`) will be inaccurate. Therefore, one or more least-significant digits of the number to format (`nbr`) may be rounded and not necessarily truncated due to the inaccuracy of the CPU's and/or compiler's floating-point implementation.

Leading character option (`lead_char`) prepends leading characters prior to the first non-zero significant digit. Leading character must be a printable ASCII character; but must not be a base-10 digit, with the exception of '0'.

For floating point numbers, the number of leading characters is such that the total number of significant integer digits plus the number of leading characters plus possible negative sign character is equal to the requested number of integer digits to format (`nbr_dig`). Examples:

```
nbr           = 23456.789
nbr_dig       = 7
nbr_dp        = 2
lead_char     = ' '
pstr          = " 23456.78"
```

```
nbr           = -23456.789
nbr_dig       = 7
nbr_dp        = 2
lead_char     = ' '
pstr          = " -23456.78"
```

If the integer value of the number to format (**nbr**) is zero and the number of digits to format (**nbr_dig**) is greater than one **OR** the number is not negative; but no leading character (**lead_char**) available; then one digit of '0' value is formatted preceding the decimal point. This is not a leading character; but a single integer digit of '0' value. Examples:

```
nbr      = 0.789
nbr_dig   = 7
nbr_dp    = 2
lead_char = '\0'
pstr      = "0.78"
```

```
nbr      = 0.789
nbr_dig   = 0
nbr_dp    = 2
lead_char = '\0'
pstr      = "0.78"
```

If the number to format (**nbr**) is negative and the leading character (**lead_char**) is a '0' digit; then the negative sign character prefixes all leading characters prior to the formatted number. Example:

```
nbr      = -23456.789
nbr_dig   = 8
nbr_dp    = 2
lead_char = '0'
pstr      = "-0023456.78"
```

If the number to format (**nbr**) is negative and the leading character (**lead_char**) is **not** a '0' digit; then the negative sign character immediately prefixes the most significant digit of the formatted number. Example:

```
nbr      = -23456.789
nbr_dig   = 8
nbr_dp    = 2
lead_char = '#'
pstr      = "##-23456.78"
```

When **NULL**-character terminate option (**nul**) is disabled, it prevents overwriting previous character array formatting. **Warning:** Unless **pstr** character array is pre-/post-terminated, if **NULL**-character terminate option is disabled, it will cause character string run-on.

Format buffer size not validated; buffer overruns must be prevented by caller. To prevent character buffer overrun:

Character array size **must** be \geq (**nbr_dig** +
 nbr_dp +
 1 negative sign +
 1 decimal point +
 1 NUL terminator) characters

EXAMPLE USAGE

```
CPU_CHAR  AppBuf[20];
CPU_CHAR  *pstr;

pstr = Str_FmtNbr_32((CPU_FP32  )-1234.5678,
                    (CPU_INT08U ) 5,
                    (CPU_INT08U ) 2,
                    (CPU_CHAR   ) ' ',
                    (CPU_BOOLEAN) DEF_NO,
                    (CPU_BOOLEAN) DEF_YES,
                    (CPU_CHAR   *)&AppBuf[0]);
```

5-2-20 Str_ParseNbr_Int32U()

Parses a 32-bit unsigned integer from a string.

FILES

lib_str.h/lib_str.c

PROTOTYPE

```
CPU_INT32U Str_ParseNbr_Int32U (const CPU_CHAR *pstr,  
                                CPU_CHAR **pstr_next,  
                                CPU_INT08U  nbr_base);
```

ARGUMENTS

pstr Pointer to string.

pstr_end Pointer to a variable to ...

Return a pointer to first character following the integer string, if no errors;
Return a pointer to **pstr**, if any errors.

nbr_base Base of number to parse:

0 (zero); the actual base will be determined from the integer string:

If the integer string begins with "0x" or "0X", the base is 16.

If the integer string begins with "0" but not "0x"/"0X", the base is 8.

Otherwise, the base is 10.

Integer between 2 and 36, inclusive.

RETURNED VALUE

Parsed integer, if integer was successfully parsed and did not.
`DEF_INT_32U_MAX_VAL`, if parsed integer overflowed to the most positive value.
0, otherwise.

REQUIRED CONFIGURATION

None.

NOTES / WARNINGS

The input string consists of:

An initial, possibly empty, sequence of white-space characters.

An optional sign character ('+'); a negative sign character ('-') will be interpreted as an invalid character.

A sequence of characters representing an integer in some radix:

If the base is 16, one of the optional character sequences "0x" or "0X";

A sequence of letters and digits. The letters from 'a'/'A' to 'z'/'Z' are assigned the values 10 through 35, respectively; but only letters and digits whose assigned values are less than that of the base are valid.

A string of invalid or unrecognized characters, perhaps including a terminating `NULL` character.

Return integer value and next string pointer (`pstr_end`) should be used to diagnose parse success or failure. Examples:

Valid parse string integer:

```
pstr      = "    ABCDE xyz"
nbr_base  = 16
nbr       = 703710
pstr_next = " xyz"
```

Invalid parse string integer:

```
pstr      = "      ABCDE"
nbr_base  = 10
nbr       = 0
pstr_next = pstr = "      ABCDE"
```

Valid hexadecimal parse string integer:

```
pstr      = "      0xGABCDE"
nbr_base  = 16
nbr       = 0
pstr_next = "xGABCDE"
```

Valid decimal parse string integer ('0x' prefix ignored following invalid hexadecimal characters):

```
pstr      = "      0xGABCDE"
nbr_base  = 0
nbr       = 0
pstr_next = "xGABCDE"
```

Valid decimal parse string integer ('0' prefix ignored following invalid octal characters):

```
pstr      = "      0GABCDE"
nbr_base  = 0
nbr       = 0
pstr_next = "GABCDE"
```

Parse string integer overflow:

```
pstr      = "      12345678901234567890*123456"
nbr_base  = 10
nbr       = DEF_INT_32S_MAX_VAL
pstr_next = "*123456"
```


Invalid negative unsigned parse string:

```
pstr      = "  -12345678901234567890*123456"
nbr_base  = 10
nbr        = 0
pstr_next = pstr = "  -12345678901234567890*123456"
```

EXAMPLE USAGE

```
CPU_INT32U  nbr;
CPU_CHAR    *pstr_end;

nbr = Str_ParseNbr_Int32U((CPU_CHAR *)"01234534*-23434>345344",
                          (CPU_CHAR **)&pstr_end,
                          (CPU_INT08U ) 10u);
```

5-2-21 Str_ParseNbr_Int32S()

Parses a 32-bit signed integer from a string.

FILES

lib_str.h/lib_str.c

PROTOTYPE

```
CPU_INT32S Str_ParseNbr_Int32S (CPU_CHAR *pstr,  
                                CPU_CHAR **pstr_end,  
                                CPU_INT08U nbr_base);
```

ARGUMENTS

pstr Pointer to string.

pstr_end Pointer to a variable to ...

Return a pointer to first character following the integer string, if no errors;
Return a pointer to **pstr**, if any errors.

nbr_base Base of number to parse:

0 (zero); the actual base will be determined from the integer string:

If the integer string begins with "0x" or "0X", the base is 16.

If the integer string begins with "0" but not "0x"/"0X", the base is 8.

Otherwise, the base is 10.

Integer between 2 and 36, inclusive.

RETURNED VALUE

Parsed integer,	if integer was successfully parsed and neither overflowed or underflowed.
DEF_INT_32S_MAX_VAL,	if parsed integer overflowed to the most positive value.
DEF_INT_32S_MIN_VAL,	if parsed integer underflowed to the most negative value.
0,	otherwise.

REQUIRED CONFIGURATION

None.

NOTES / WARNINGS

The input string consists of:

An initial, possibly empty, sequence of white-space characters.

An optional sign character ('-' or '+').

A sequence of characters representing an integer in some radix:

If the base is 16, one of the optional character sequences "0x" or "0X";

A sequence of letters and digits. The letters from 'a'/'A' to 'z'/'Z' are assigned the values 10 through 35, respectively; but only letters and digits whose assigned values are less than that of the base are valid.

A string of invalid or unrecognized characters, perhaps including a terminating **NULL** character.

Return integer value and next string pointer (**pstr_end**) should be used to diagnose parse success or failure. Examples:

Valid parse string integer:

```
pstr      = "      ABCDE xyz"
nbr_base  = 16
nbr       = 703710
pstr_next = " xyz"
```

Invalid parse string integer:

```
pstr      = "      ABCDE"
nbr_base  = 10
nbr       = 0
pstr_next = pstr = "      ABCDE"
```

Valid hexadecimal parse string integer:

```
pstr      = "      0xGABCDE"
nbr_base  = 16
nbr       = 0
pstr_next = "xGABCDE"
```

Valid decimal parse string integer ('0x' prefix ignored following invalid hexadecimal characters):

```
pstr      = "      0xGABCDE"
nbr_base  = 0
nbr       = 0
pstr_next = "xGABCDE"
```

Valid decimal parse string integer ('0' prefix ignored following invalid octal characters):

```
pstr      = "      0GABCDE"
nbr_base  = 0
nbr       = 0
pstr_next = "GABCDE"
```

Parse string integer overflow:

```
pstr      = "  12345678901234567890*123456"
nbr_base  = 10
nbr        = DEF_INT_32S_MAX_VAL
pstr_next = "*123456"
```

Parse string integer underflow:

```
pstr      = " -12345678901234567890*123456"
nbr_base  = 10
nbr        = DEF_INT_32S_MIN_VAL
pstr_next = "*123456"
```

EXAMPLE USAGE

```
CPU_INT32S  nbr;
CPU_CHAR    *pstr_end;

nbr = Str_ParseNbr_Int32S((CPU_CHAR *)"-1234534*-23434>345344",
                          (CPU_CHAR **)&pstr_end,
                          (CPU_INT08U ) 10u);
```

µC/LIB ASCII Library

µC/LIB contains library functions that replace standard library character classification and case conversion functions and macros such as `tolower()`, `toupper()`, `isalpha()`, `isdigit()`, etc. Character classification functions and macros determine whether a character belongs to a certain class of character (e.g., uppercase alphabetic characters). Character case conversion functions and macros convert a character from uppercase to lowercase or lowercase to uppercase. These functions are defined in `lib_ascii.c`.

6-1 CHARACTER VALUE CONSTANTS

µC/LIB contains many character value constants such as

```
ASCII_CHAR_LATIN_DIGIT_ZERO ... ASCII_CHAR_LATIN_DIGIT_NINE
ASCII_CHAR_LATIN_UPPER_A    ... ASCII_CHAR_LATIN_UPPER_Z
ASCII_CHAR_LATIN_LOWER_A    ... ASCII_CHAR_LATIN_LOWER_Z
```

One constant exists for each ASCII character, though additional aliases are provided for some characters. These constants should be used to configure, assign, and test appropriately-sized ASCII character values or variables.

6-2 ASCII LIBRARY MACROS AND FUNCTIONS

6-2-1 ASCII_IS_ALPHA() / ASCII_IsAlpha()

Determines whether a character is an alphabetic character.

FILES

lib_ascii.h/lib_ascii.c

PROTOTYPES

```
ASCII_IS_ALPHA(c);  
  
CPU_BOOLEAN ASCII_IsAlpha (CPU_CHAR c);
```

ARGUMENTS

c Character to examine.

RETURNED VALUE

DEF_YES, if character is an alphabetic character;

DEF_NO, if character is not an alphabetic character.

REQUIRED CONFIGURATION

None.

NOTES / WARNINGS

ISO/IEC 9899:TC2, Section 7.4.1.2.(2) states that “**isalpha()** returns true only for the characters for which **isupper()** or **islower()** is true”.

EXAMPLE USAGE

```
CPU_CHAR    c;  
CPU_BOOLEAN  alpha;  
  
c    = ASCII_CHAR_LATIN_UPPER_G;  
alpha = ASCII_IS_ALPHA(c);
```


6-2-2 ASCII_IS_ALPHA_NUM() / ASCII_IsAlphaNum)

Determines whether a character is an alphanumeric character.

FILES

lib_ascii.h/lib_ascii.c

PROTOTYPES

```
ASCII_IS_ALPHA_NUM(c);  
  
CPU_BOOLEAN ASCII_IsAlphaNum (CPU_CHAR c);
```

ARGUMENTS

c Character to examine.

RETURNED VALUE

DEF_YES, if character is an alphanumeric character;

DEF_NO, if character is not an alphanumeric character.

REQUIRED CONFIGURATION

None.

NOTES / WARNINGS

ISO/IEC 9899:TC2, Section 7.4.1.1.(2) states that “**isalnum()** returns true only for the characters for which **isalpha()** or **isdigit()** is true”.

EXAMPLE USAGE

```
CPU_CHAR    c;  
CPU_BOOLEAN  alpha_num;  
  
c           = ASCII_CHAR_LATIN_UPPER_G;  
alpha_num = ASCII_IS_ALPHA_NUM(c);
```

6-2-3 ASCII_IS_LOWER() / ASCII_IsLower()

Determines whether a character is a lowercase alphabetic character.

FILES

lib_ascii.h/lib_ascii.c

PROTOTYPES

```
ASCII_IS_LOWER(c);  
  
CPU_BOOLEAN ASCII_IsLower (CPU_CHAR c);
```

ARGUMENTS

c Character to examine.

RETURNED VALUE

DEF_YES, if character is a lowercase alphabetic character;

DEF_NO, if character is not a lowercase alphabetic character.

REQUIRED CONFIGURATION

None.

NOTES / WARNINGS

ISO/IEC 9899:TC2, Section 7.4.1.7.(2) states that “**islower()** returns true only for the lowercase letters”.

EXAMPLE USAGE

```
CPU_CHAR    c;  
CPU_BOOLEAN lower;  
  
c    = ASCII_CHAR_LATIN_LOWER_G;  
lower = ASCII_IS_LOWER(c);
```

6-2-4 ASCII_IS_UPPER() / ASCII_IsUpper()

Determines whether a character is an uppercase alphabetic character.

FILES

lib_ascii.h/lib_ascii.c

PROTOTYPES

```
ASCII_IS_UPPER(c);  
  
CPU_BOOLEAN ASCII_IsUpper (CPU_CHAR c);
```

ARGUMENTS

c Character to examine.

RETURNED VALUE

DEF_YES, if character is an uppercase alphabetic character;

DEF_NO, if character is not an uppercase alphabetic character.

REQUIRED CONFIGURATION

None.

NOTES / WARNINGS

ISO/IEC 9899:TC2, Section 7.4.1.11.(2) states that “**isupper()** returns true only for the uppercase letters”.

EXAMPLE USAGE

```
CPU_CHAR    c;  
CPU_BOOLEAN upper;  
  
c    = ASCII_CHAR_LATIN_UPPER_G;  
upper = ASCII_IS_UPPER(c);
```

6-2-5 ASCII_IS_DIG() / ASCII_IsDig()

Determines whether a character is a decimal-digit character.

FILES

lib_ascii.h/lib_ascii.c

PROTOTYPES

```
ASCII_IS_DIG(c);  
  
CPU_BOOLEAN ASCII_IsDig (CPU_CHAR c);
```

ARGUMENTS

c Character to examine.

RETURNED VALUE

DEF_YES, if character is a decimal-digit character;

DEF_NO, if character is not a decimal-digit character.

REQUIRED CONFIGURATION

None.

NOTES / WARNINGS

ISO/IEC 9899:TC2, Section 7.4.1.5.(2) states that “**isdigit()** ... tests for any decimal-digit character”.

EXAMPLE USAGE

```
CPU_CHAR    c;  
CPU_BOOLEAN dig;  
  
c  = ASCII_CHAR_DIGIT_SEVEN;  
dig = ASCII_IS_DIG(c);
```


6-2-6 ASCII_IS_DIG_OCT() / ASCII_IsDigOct()

Determines whether a character is an octal-digit character.

FILES

lib_ascii.h/lib_ascii.c

PROTOTYPES

```
ASCII_IS_DIG_OCT(c);  
  
CPU_BOOLEAN ASCII_IsDigOct (CPU_CHAR c);
```

ARGUMENTS

c Character to examine.

RETURNED VALUE

DEF_YES, if character is an octal-digit character;

DEF_NO, if character is not an octal-digit character.

REQUIRED CONFIGURATION

None.

NOTES / WARNINGS

None.

EXAMPLE USAGE

```
CPU_CHAR    c;  
CPU_BOOLEAN dig_oct;  
  
c           = ASCII_CHAR_DIGIT_SEVEN;  
dig_oct = ASCII_IS_DIG_OCT(c);
```

6-2-7 ASCII_IS_DIG_HEX() / ASCII_IsDigHex()

Determines whether a character is a hexadecimal-digit character.

FILES

lib_ascii.h/lib_ascii.c

PROTOTYPES

```
ASCII_IS_DIG_HEX(c);  
  
CPU_BOOLEAN ASCII_IsDigHex (CPU_CHAR c);
```

ARGUMENTS

c Character to examine.

RETURNED VALUE

DEF_YES, if character is a hexadecimal-digit character;

DEF_NO, if character is not a hexadecimal-digit character.

REQUIRED CONFIGURATION

None.

NOTES / WARNINGS

ISO/IEC 9899:TC2, Section 7.4.1.12.(2) states that “**isxdigit()** ... tests for any hexadecimal-digit character”.

EXAMPLE USAGE

```
CPU_CHAR    c;
CPU_BOOLEAN  dig_hex;

c            = ASCII_CHAR_LATIN_UPPER_C;
dig_hex = ASCII_IS_DIG_HEX(c);
```

6-2-8 ASCII_IS_BLANK() / ASCII_IsBlank()

Determines whether a character is a standard blank character.

FILES

lib_ascii.h/lib_ascii.c

PROTOTYPES

```
ASCII_IS_BLANK(c);  
  
CPU_BOOLEAN ASCII_IsBlank (CPU_CHAR c);
```

ARGUMENTS

c Character to examine.

RETURNED VALUE

DEF_YES, if character is a standard blank character;

DEF_NO, if character is not a standard blank character.

REQUIRED CONFIGURATION

None.

NOTES / WARNINGS

ISO/IEC 9899:TC2, Section 7.4.1.3.(2) states that “**isblank()** returns true only for the standard blank characters”. ISO/IEC 9899:TC2, Section 7.4.1.3.(2) defines “the standard blank characters” as the “space (' '), and horizontal tab ('\t ')”.

EXAMPLE USAGE

```
CPU_CHAR    c;  
CPU_BOOLEAN  blank;  
  
c    = ASCII_CHAR_LINE_FEED;  
blank = ASCII_IS_BLANK(c);
```

6-2-9 ASCII_IS_SPACE() / ASCII_IsSpace()

Determines whether a character is a white-space character.

FILES

lib_ascii.h/lib_ascii.c

PROTOTYPES

```
ASCII_IS_SPACE(c);  
  
CPU_BOOLEAN ASCII_IsSpace (CPU_CHAR c);
```

ARGUMENTS

c Character to examine.

RETURNED VALUE

DEF_YES, if character is a white-space character;

DEF_NO, if character is not a white-space character.

REQUIRED CONFIGURATION

None.

NOTES / WARNINGS

ISO/IEC 9899:TC2, Section 7.4.1.10.(2) states that “**isspace()** returns true only for the standard white-space characters”. ISO/IEC 9899:TC2, Section 7.4.1.10.(2) defines “the standard white-space characters” as the “space (' '), form feed ('\f'), new-line ('\n'), carriage return ('\r'), horizontal tab ('\t'), and vertical tab ('\v’)”.

EXAMPLE USAGE

```
CPU_CHAR    c;  
CPU_BOOLEAN space;  
  
c    = ASCII_CHAR_CARRIAGE_RETURN;  
space = ASCII_IS_SPACE(c);
```

6-2-10 ASCII_IS_PRINT() / ASCII_IsPrint()

Determines whether a character is a printing character.

FILES

lib_ascii.h/lib_ascii.c

PROTOTYPES

```
ASCII_IS_PRINT(c);  
  
CPU_BOOLEAN ASCII_IsPrint (CPU_CHAR c);
```

ARGUMENTS

c Character to examine.

RETURNED VALUE

DEF_YES, if character is a printing character;

DEF_NO, if character is not a printing character.

REQUIRED CONFIGURATION

None.

NOTES / WARNINGS

ISO/IEC 9899:TC2, Section 7.4.1.8.(2) states that “**isprint()** ... tests for any printing character including space (' ')”. ISO/IEC 9899:TC2, Section 7.4.(3), Note 169, states that in “the seven-bit US ASCII character set, the printing characters are those whose values lie from 0x20 (space) through 0x7E (tilde)”.

EXAMPLE USAGE

```
CPU_CHAR    c;
CPU_BOOLEAN  print;

c    = ASCII_CHAR_LATIN_UPPER_G;
print = ASCII_IS_PRINT(c);
```

6-2-11 ASCII_IS_GRAPH() / ASCII_IsGraph()

Determines whether a character is a graphic character.

FILES

lib_ascii.h/lib_ascii.c

PROTOTYPES

```
ASCII_IS_GRAPH(c);  
  
CPU_BOOLEAN ASCII_IsGraph (CPU_CHAR c);
```

ARGUMENTS

c Character to examine.

RETURNED VALUE

DEF_YES, if character is a graphic character;

DEF_NO, if character is not a graphic character.

REQUIRED CONFIGURATION

None.

NOTES / WARNINGS

ISO/IEC 9899:TC2, Section 7.4.1.6.(2) states that “**isgraph()** ... tests for any printing character except space (' ')”. ISO/IEC 9899:TC2, Section 7.4.(3), Note 169, states that in “the seven-bit US ASCII character set, the printing characters are those whose values lie from 0x20 (space) through 0x7E (tilde)”.

EXAMPLE USAGE

```
CPU_CHAR    c;  
CPU_BOOLEAN  graph;  
  
c    = ASCII_CHAR_LATIN_UPPER_G;  
graph = ASCII_IS_GRAPH(c);
```

6-2-12 ASCII_IS_PUNCT() / ASCII_IsPunct()

Determines whether a character is a punctuation character.

FILES

lib_ascii.h/lib_ascii.c

PROTOTYPES

```
ASCII_IS_PUNCT(c);  
  
CPU_BOOLEAN ASCII_IsPunct (CPU_CHAR c);
```

ARGUMENTS

c Character to examine.

RETURNED VALUE

DEF_YES, if character is a punctuation character;

DEF_NO, if character is not a punctuation character.

REQUIRED CONFIGURATION

None.

NOTES / WARNINGS

ISO/IEC 9899:TC2, Section 7.4.1.9.(2) states that “**ispunct()** returns true for every printing character for which neither **isspace()** nor **isalnum()** is true”.

EXAMPLE USAGE

```
CPU_CHAR    c;  
CPU_BOOLEAN punct;  
  
c    = ASCII_CHAR_COLON;  
punct = ASCII_IS_PUNCT(c);
```

6-2-13 ASCII_IS_CTRL() / ASCII_IsCtrl()

Determines whether a character is a control character.

FILES

lib_ascii.h/lib_ascii.c

PROTOTYPES

```
ASCII_IS_CTRL(c);  
  
CPU_BOOLEAN ASCII_IsCtrl (CPU_CHAR c);
```

ARGUMENTS

c Character to examine.

RETURNED VALUE

DEF_YES, if character is a control character;

DEF_NO, if character is not a control character.

REQUIRED CONFIGURATION

None.

NOTES / WARNINGS

ISO/IEC 9899:TC2, Section 7.4.1.4.(2) states that “**iscntrl()** ... tests for any control character”. ISO/IEC 9899:TC2, Section 7.4.(3), Note 169, states that in “the seven-bit US ASCII character set, ... the control characters are those whose values lie from 0 (NUL) through 0x1F (US), and the character 0x7F (DEL)”.

EXAMPLE USAGE

```
CPU_CHAR    c;
CPU_BOOLEAN  ctrl;

c    = ASCII_CHAR_DELETE;
ctrl = ASCII_IS_CTRL(c);
```

6-2-14 ASCII_TO_LOWER() / ASCII_ToLower()

Converts an uppercase alphabetic character to its corresponding lowercase alphabetic character.

FILES

lib_ascii.h/lib_ascii.c

PROTOTYPES

```
ASCII_TO_LOWER(c);  
  
CPU_CHAR ASCII_ToLower (CPU_CHAR c);
```

ARGUMENTS

c Character to examine.

RETURNED VALUE

Lowercase equivalent of **c**, if character **c** is an uppercase character;

Character **c**, otherwise.

REQUIRED CONFIGURATION

None.

NOTES / WARNINGS

ISO/IEC 9899:TC2, Section 7.4.2.1.(2) states that “**tolower()** ... converts an uppercase letter to a corresponding lowercase letter”. ISO/IEC 9899:TC2, Section 7.4.2.1.(3) states that “if the argument is a character for which **isupper()** is true and there are one or more corresponding characters ... for which **islower()** is true, ... **tolower()** ... returns one of the corresponding characters; ... otherwise, the argument is returned unchanged”.

EXAMPLE USAGE

```
CPU_CHAR  c;  
CPU_CHAR  c_lower;  
  
c          = ASCII_CHAR_LATIN_UPPER_G;  
c_lower = ASCII_TO_LOWER(c);
```

6-2-15 ASCII_TO_UPPER() / ASCII_ToUpper()

Converts a lowercase alphabetic character to its corresponding uppercase alphabetic character.

FILES

lib_ascii.h/lib_ascii.c

PROTOTYPES

```
ASCII_TO_UPPER(c);  
  
CPU_CHAR ASCII_ToUpper (CPU_CHAR c);
```

ARGUMENTS

c Character to examine.

RETURNED VALUE

Uppercase equivalent of **c**, if character **c** is an lowercase character;

Character **c**, otherwise.

REQUIRED CONFIGURATION

None.

NOTES / WARNINGS

ISO/IEC 9899:TC2, Section 7.4.2.2.(2) states that “**toupper()** ... converts a lowercase letter to a corresponding uppercase letter”. ISO/IEC 9899:TC2, Section 7.4.2.2.(3) states that “if the argument is a character for which **islower()** is true and there are one or more corresponding characters ... for which **isupper()** is true, ... **toupper()** ... returns one of the corresponding characters; ... otherwise, the argument is returned unchanged”.

EXAMPLE USAGE

```
CPU_CHAR  c;  
CPU_CHAR  c_upper;  
  
c          = ASCII_CHAR_LATIN_LOWER_G;  
c_upper = ASCII_TO_UPPER(c);
```

6-2-16 ASCII_Cmp()

Determines if two characters are identical, ignoring case.

FILES

lib_ascii.h/lib_ascii.c

PROTOTYPE

```
CPU_BOOLEAN ASCII_Cmp (CPU_CHAR c1,  
                        CPU_CHAR c2);
```

ARGUMENTS

c1 First character.

c2 Second character.

RETURNED VALUE

DEF_YES, if characters are identical;

DEF_NO, if character are not identical.

REQUIRED CONFIGURATION

None.

NOTES / WARNINGS

None.

EXAMPLE USAGE

```
CPU_CHAR  c;  
CPU_CHAR  c_upper;  
  
c          = ASCII_CHAR_LATIN_LOWER_G;  
c_upper = ASCII_TO_UPPER(c);  
cmp        = ASCII_Cmp(c_upper, c_upper);
```

Chapter

7

μ C/LIB Mathematics Library

μ C/LIB contains library functions that replace standard mathematics functions such as `rand()`, `srand()`, etc. These functions are defined in `lib_math.c`.

7-1 MATHEMATICS LIBRARY FUNCTIONS

7-1-1 Math_Init()

Initializes the mathematics library.

FILES

lib_math.h/lib_math.c

PROTOTYPE

```
void Math_Init (void);
```

ARGUMENTS

None.

RETURNED VALUE

None.

REQUIRED CONFIGURATION

None.

NOTES / WARNINGS

Math_Init() must be called prior to calling any other mathematics library functions.

7-1-2 Math_RandSetSeed()

Sets the current pseudo-random number sequence.

FILES

lib_math.h/lib_math.c

PROTOTYPE

```
void Math_RandSetSeed (RAND_NBR seed);
```

ARGUMENTS

seed Initial (or current) value to set for the pseudo-random number sequence.

RETURNED VALUE

None.

REQUIRED CONFIGURATION

None.

NOTES / WARNINGS

IEEE Std 1003.1, 2004 Edition, Section ‘**rand()** : DESCRIPTION’ states that “**srand()** ... uses the argument as a seed for a new sequence of pseudo-random numbers to be returned by subsequent calls to **rand()**”.

EXAMPLE USAGE

```
RAND_NBR seed;  
  
seed = 9876;  
Math_RandSetSeed(seed);
```


7-1-3 Math_Rand()

Gets the next pseudo-random number.

FILES

lib_math.h/lib_math.c

PROTOTYPE

```
RAND_NBR Math_Rand (void);
```

ARGUMENTS

None.

RETURNED VALUE

Next pseudo-random number in the sequence.

REQUIRED CONFIGURATION

None.

NOTES / WARNINGS

Pseudo-random number generated implemented as a Linear Congruential Generator (LCG). The pseudo-random number generated is in the range $[0, 2^{31})$.

`Math_Rand()` is re-entrant since it calculates the next random number in critical sections.

EXAMPLE USAGE

```
RAND_NBR rand_nbr;  
  
rand_nbr = Math_Rand();
```

7-1-4 Math_RandSeed()

Gets the next pseudo-random number following **seed**.

FILES

lib_math.h/lib_math.c

PROTOTYPE

```
RAND_NBR Math_RandSeed (RAND_NBR seed);
```

ARGUMENTS

seed Initial (or current) value to set for the pseudo-random number sequence.

RETURNED VALUE

Next pseudo-random number in the sequence following **seed**.

REQUIRED CONFIGURATION

None.

NOTES / WARNINGS

Pseudo-random number generated implemented as a Linear Congruential Generator (LCG).
The pseudo-random number generated is in the range $[0, 2^{31})$.

Math_RandSeed() is re-entrant since it calculates the next random number using only local variables.

EXAMPLE USAGE

```
RAND_NBR  seed;
RAND_NBR  rand_nbr;

seed      = 9876;
rand_nbr = Math_RandSeed(seed);
```

Appendix

A

μC/LIB Licensing Policy

You need to obtain an “Object Code Distribution License” to embed μC/LIB in a product that is sold with the intent to make a profit. Each individual product (*i.e.*, your product) requires its own license, but the license allows you to distribute an unlimited number of units for the life of your product. Please indicate the processor type(s) (*i.e.*, ARM7, ARM9, MCF5272, MicroBlaze, Nios II, PPC, *etc.*) that you intend to use.

For licensing details, contact us at:

Micrium
1290 Weston Road, Suite 306
Weston, FL 33326
USA

Phone: +1 954 217 2036
Fax: +1 954 217 2037
E-mail: Licensing@Micrium.com
Web: www.Micrium.com