Micriµm

Empowering Embedded Systems

µC/LIB

V1.27

User's Manual

www.Micrium.com

Registration

Please register the software via email. This way we can make sure you will receive updates or notifications of updates as soon as they become available. For registration please provide the following information:

- Your full name and the name of your supervisor
- Your company name
- Your job title
- Your email address and telephone number
- Company name and address
- Your company's main phone number
- Your company's web site address
- Name and version of the product

Please send this information to: **licensing@micrium.com**

Contact address

Micriμm
949 Crestview Circle
Weston, FL 33327-1848
U.S.A.
Phone : +1 954 217 2036
FAX : +1 954 217 2037
WEB : **www.micrium.com**
Email : **support@micrium.com**

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
V1.20	2006/06/01	ITJ	Updated
V1.21	2006/08/09	ITJ	Updated
V1.22	2006/09/20	ITJ	Updated
V1.23	2007/02/27	ITJ	Updated
V1.24	2007/05/04	ITJ	Updated
V1.25	2008/06/20	ITJ	Updated
V1.26	2008/12/01	ITJ	Updated
V1.27	2009/12/05	ITJ	Updated

Table Of Contents

I Introduction 7
I.1 Portable 7
I.2 Scalable 7
I.3 Coding Standards 7
I.4 MISRA C 7
I.5 Safety Critical Certification 8
I.6 µC/LIB Limitations 8

1 Getting Started with µC/LIB 9
1.00 Installing µC/LIB 9

2 µC/LIB Constant and Macro Library 12
2.00.01 Boolean Constants 12
2.00.02 Bit Constants 12
2.00.03 Octet Constants 12
2.00.04 Number Base Constants 12
2.00.05 Integer Constants 13
2.00.06 Time Constants 13
2.10 Macros 14
2.10.01.01 DEF_BIT() 14
2.10.01.02 DEF_BIT_MASK() 15
2.10.01.03 DEF_BIT_FIELD() 16
2.10.01.04 DEF_BIT_SET() 17
2.10.01.05 DEF_BIT_CLR() 18
2.10.01.06 DEF_BIT_IS_SET() 19
2.10.01.07 DEF_BIT_IS_CLR() 20
2.10.01.08 DEF_BIT_IS_SET_ANY() 21
2.10.01.09 DEF_BIT_IS_CLR_ANY() 22
2.10.02.01 DEF_MIN() 23
2.10.02.02 DEF_MAX() 24
2.10.02.03 DEF_ABS() 25

3 µC/LIB Memory Library 26
3.00 µC/LIB Memory Library Configuration 26
3.10.01 MEM_VAL_GET_xxx() 27
3.10.02 MEM_VAL_SET_xxx() 29
3.10.03 MEM_VAL_COPY_GET_xxx() 31
3.10.04 MEM_VAL_COPY_SET_xxx() 33
3.10.05 MEM_VAL_COPY_xxx() 35
3.20.01 Mem_Clr() 36
3.20.02 Mem_Set() 37
3.20.03 Mem_Copy() 38
3.20.04 Mem_Cmp() 39

3.30 µC/LIB Memory Allocation Functionality 40
3.30.01 Mem_HeapAlloc() 41
3.30.02.01 Mem_PoolCreate() 43
3.30.02.02 Mem_PoolBlkGet() 46
3.30.02.03 Mem_PoolBlkFree() 48

3.40 µC/LIB Memory Library Optimization 50
3.40.01 Mem_Copy() Optimization 50

4 µC/LIB String Library 51
4.00 µC/LIB String Library Configuration 51
4.10.01 Str_Len() 52
4.10.02.01 Str_Copy() 53
4.10.02.02 Str_Copy_N() 54
4.10.03.01 Str_Cat() 55
4.10.03.02 Str_Cat_N() 56
4.10.04.01 Str_Cmp() 57
4.10.04.02 Str_Cmp_N() 58
4.10.04.03 Str_CmpIgnoreCase() 59
4.10.04.04 Str_CmpIgnoreCase_N() 60
4.10.05.01 Str_Char() 62
4.10.05.02 Str_Char_N() 63
4.10.05.03 Str_Char_Last() 64
4.10.06 Str_Str() 65
4.10.07.01 Str_FmtNbr_Int32U() 66
4.10.07.02 Str_FmtNbr_Int32S() 69
4.10.07.03 Str_FmtNbr_32() 72
4.10.08.01 Str_ParseNbr_Int32U() 74
4.10.08.02 Str_ParseNbr_Int32S() 77

5 µC/LIB ASCII Library 80
5.00 Character Value Constants 80
5.10 Functions & Macros 81
5.10.01 ASCII_IS_ALPHA() / ASCII_IsAlpha() 81
5.10.02 ASCII_IS_ALPHA_NUM() / ASCII_IsAlphaNum() 82
5.10.03 ASCII_IS_LOWER() / ASCII_IsLower() 83
5.10.04 ASCII_IS_UPPER() / ASCII_IsUpper() 84
5.10.05 ASCII_IS_DIG() / ASCII_IsDig() 85
5.10.06 ASCII_IS_DIG_OCT() / ASCII_IsDigOct() 86

5.10.07 ASCII_IS_DIG_HEX() / ASCII_IsDigHex() 87
5.10.08 ASCII_IS_BLANK() / ASCII_IsBlank() 88
5.10.09 ASCII_IS_SPACE() / ASCII_IsSpace() 89
5.10.10 ASCII_IS_PRINT() / ASCII_IsPrint() 90
5.10.11 ASCII_IS_GRAPH() / ASCII_IsGraph() 91
5.10.12 ASCII_IS_PUNCT() / ASCII_IsPunct() 92
5.10.13 ASCII_IS_CTRL() / ASCII_IsCtrl() 93
5.10.14 ASCII_TO_LOWER() / ASCII_ToLower() 94
5.10.15 ASCII_TO_UPPER() / ASCII_ToUpper() 95

A µC/LIB Licensing Policy 96

Introduction

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

I.1 Portable

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

I.2 Scalable

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

I.3 Coding Standards

Coding standards have been established early in the design of µC/LIB and include the following:

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

I.4 MISRA C

The source code for µC/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**.

I.5 Safety Critical Certification

µC/LIB has been designed and implemented with safety critical certification in mind. µC/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), and IEC 61058 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, µC/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 µC/LIB is a very robust and highly-reliable software module.

I.6 µC/LIB Limitations

By design, we have limited some of the feature of µC/LIB. Table I-1 describes those limitations.

Does not support variable argument library functions

Table I-1, µC/LIB limitations for current software version

Chapter 1

Getting Started with µC/LIB

This chapter provides information on the distribution and installation of µC/LIB.

1.00 Installing µC/LIB

The distribution of µC/LIB is typically included in a ZIP file called: **`uC-LIB-Vxyy.zip`**. µC/LIB could also have been included in the distribution of another Micrium ZIP file (µC/OS-II, µC/TCP-IP, µC/FS, etc.). The ZIP file contains all the source code and documentation for µC/LIB as well as all other required software modules. All modules are placed in their respective directories as shown in Figure 1-1.

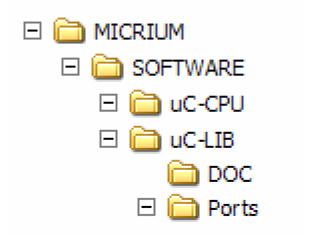


Figure 1-1, µC/LIB Module Directories

`\uC-CPU` This directory contains CPU-specific code which depends on the processor and compiler used. The directory contains additional sub-directories specific for each processor/compiler combination organized as follows :

```
\MICRIUM\SOFTWARE\uC-CPU\<CPU Type>\<Compiler>
```

The µC/CPU directory contains one master CPU file :

```
\MICRIUM\SOFTWARE\uC-CPU\cpu_def.h
```

`cpu_def.h`
This file declares `#define` constants for CPU word sizes, endianness, critical section methods, and other processor configuration.

Each sub-directory contains source files specific for each processor/compiler combination :

```
\MICRIUM\SOFTWARE\uC-CPU\<CPU Type>\<Compiler>\cpu.h
\MICRIUM\SOFTWARE\uC-CPU\<CPU Type>\<Compiler>\cpu_a.asm
```

`cpu.h`
This file contains configuration specific to the processor, such as data type definitions, processor address and data word sizes, endianness, and critical section implementation. The data type definitions are declared so as to be independent of processor and compiler word sizes.

`cpu_a.asm`
This file contains assembly code to enable/disable interrupts, implement critical section methods, and any other code specific to the processor.

`\uC-LIB` This directory contains the µC/LIB library source files common to many Micriµm products and is shown in Figure 1-2.

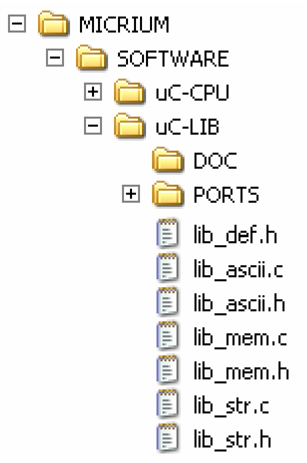


Figure 1-2, µC/LIB Library Files

`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 2 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. These functions are replaced with **`Mem_Clr()`**, **`Mem_Set()`**, **`Mem_Copy()`**, and **`Mem_Cmp()`**, respectively. See Chapter 3 for more details.

`lib_str.c` and **`lib_str.h`**
These files contain source code to replace standard library functions **`strlen()`**, **`strcpy()`**, **`strcmp()`**, etc. These functions are replaced with **`Str_Len()`**, **`Str_Copy()`**, and **`Str_Cmp()`**, respectively. See Chapter 4 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. These functions are replaced with **ASCII_ToLower()**, **ASCII_ToUpper()**, **ASCII_IsAlpha()**, and **ASCII_IsDig()**, respectively. See Chapter 5 for more details.

\Application

This directory represents the application's directory or directory tree. Application files which intend to make use of µC/LIB constants, macros, or functions should #include the desired µC/LIB header files.

app_cfg.h
This application-specific configuration file declares #define constants used to configure Micriµm products and/or non-Micriµm-related application files. This file is required by µC/LIB to #define its configuration constants.

Chapter 2

µC/LIB Constant and Macro Library

µC/LIB contains many standard constants and macros. Common constants include Boolean, bit-mask, and integer values; common macros include 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**.

2.00.01 Boolean Constants

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

2.00.02 Bit Constants

µC/LIB contains bit constants such as **DEF_BIT_00**, **DEF_BIT_07**, and **DEF_BIT_15**, which define values corresponding to specific bit positions. Currently, µC/LIB supports bit constants up to 32-bits (**DEF_BIT_31**). These constants should be used to configure, assign, and test appropriately-sized bit-field or integer values or variables.

2.00.03 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.

2.00.04 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.

2.00.05 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.

2.00.06 Time Constants

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

2.10 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.

2.10.01.01 DEF_BIT()

This macro is called to create a bit mask based on a single bit-number position.

Prototype

```
DEF_BIT(bit);
```

Arguments

`bit`	This is the bit number of the bit mask to set.

Returned Value

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

Notes / Warnings

1) **`bit`** values that overflow the target CPU &/or compiler environment (e.g. negative or greater-than-CPU-data-size values) **MAY** generate compiler warnings &/or errors.

Example

```
CPU_INT16U  mask;

mask = DEF_BIT(12);
```

2.10.01.02 DEF_BIT_MASK()

This macro is called to shift a bit mask.

Prototype

```
DEF_BIT_MASK(bit_mask, bit_shift);
```

Arguments

`bit_mask`	This is the bit mask to shift.
`bit_shift`	This is the number of bit positions to left-shift the bit mask.

Returned Value

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

Notes / Warnings

1) **`bit_shift`** values that overflow the target CPU &/or compiler environment (e.g. negative or greater-than-CPU-data-size values) **MAY** generate compiler warnings &/or errors.

Example

```
CPU_INT16U  mask;
CPU_INT16U  mask_hi;


mask    = 0x0064;
mask_hi = DEF_BIT_MASK(mask, 8);
```

2.10.01.03 DEF_BIT_FIELD()

This macro is called to create a contiguous, multi-bit bit field.

Prototype

```
DEF_BIT_FIELD(bit_field, bit_shift);
```

Arguments

`bit_field`	This is the number of contiguous bits to set in the bit field.
`bit_shift`	This is the 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.

Notes / Warnings

1) **`bit_field`**/**`bit_shift`** values that overflow the target CPU &/or compiler environment (e.g. negative or greater-than-CPU-data-size values) **MAY** generate compiler warnings &/or errors.

Example

```
CPU_INT08U  upper_nibble;

upper_nibble = DEF_BIT_FIELD(4, 4);
```

2.10.01.04 DEF_BIT_SET()

This macro is called to set the appropriate bits in a value according to a specified bit mask.

Prototype

```
DEF_BIT_SET(val, mask);
```

Arguments

`val`	This is the value to modify by setting the specified bits.
`mask`	This is the mask of bits to set in the value.

Returned Value

Modified value with specified bits set.

Notes / Warnings

None.

Example

```
CPU_INT16U  flags;
CPU_INT16U  flags_alarm;


flags       = 0x0000;
flags_alarm = DEF_BIT_00 | DEF_BIT_03;
DEF_BIT_SET(flags, flags_alarm);
```

2.10.01.05 DEF_BIT_CLR()

This macro is called to clear the appropriate bits in a value according to a specified bit mask.

Prototype

```
DEF_BIT_CLR(val, mask);
```

Arguments

`val`	This is the value to modify by clearing the specified bits.
`mask`	This is the mask of bits to clear in the value.

Returned Value

Modified value with specified bits clear.

Notes / Warnings

None.

Example

```
CPU_INT16U  flags;
CPU_INT16U  flags_alarm;


flags       = 0x0FFF;
flags_alarm = DEF_BIT_00 | DEF_BIT_03;
DEF_BIT_CLR(flags, flags_alarm);
```

2.10.01.06 DEF_BIT_IS_SET()

This macro is called to determine if all the specified bits in a value are set according to a specified bit mask.

Prototype

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

Arguments

`val`	This is the value to test if the specified bits are set.
`mask`	This is the 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`.

Notes / Warnings

None.

Example

```
CPU_INT16U  flags;
CPU_INT16U  flags_mask;
CPU_INT16U  flags_set;


flags      = 0x0369;
flags_mask = DEF_BIT_08 | DEF_BIT_09;
flags_set  = DEF_BIT_IS_SET(flags, flags_mask);
```

2.10.01.07 DEF_BIT_IS_CLR()

This macro is called to determine if all the specified bits in a value are clear according to a specified bit mask.

Prototype

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

Arguments

`val`	This is the value to test if the specified bits are clear.
`mask`	This is the 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`.

Notes / Warnings

None.

Example

```
CPU_INT16U  alarms;
CPU_INT16U  alarms_mask;
CPU_INT16U  alarms_clr;


alarms      = 0x07F0;
alarms_mask = DEF_BIT_04 | DEF_BIT_03;
alarms_clr  = DEF_BIT_IS_CLR(alarms, alarms_mask);
```

2.10.01.08 DEF_BIT_IS_SET_ANY()

This macro is called to determine if any of the specified bits in a value are set according to a specified bit mask.

Prototype

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

Arguments

`val`	This is the value to test if any of the specified bits are set.
`mask`	This is the 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`.

Notes / Warnings

None.

Example

```
CPU_INT16U  flags;
CPU_INT16U  flags_mask;
CPU_INT16U  flags_set;


flags      = 0x0369;
flags_mask = DEF_BIT_08 | DEF_BIT_09;
flags_set  = DEF_BIT_IS_SET_ANY(flags, flags_mask);
```

2.10.01.09 DEF_BIT_IS_CLR_ANY()

This macro is called to determine if any of the specified bits in a value are clear according to a specified bit mask.

Prototype

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

Arguments

`val`	This is the value to test if any of the specified bits are clear.
`mask`	This is the 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`.

Notes / Warnings

None.

Example

```
CPU_INT16U  alarms;
CPU_INT16U  alarms_mask;
CPU_INT16U  alarms_clr;


alarms      = 0x07F0;
alarms_mask = DEF_BIT_04 | DEF_BIT_03;
alarms_clr  = DEF_BIT_IS_CLR_ANY(alarms, alarms_mask);
```

2.10.02.01 DEF_MIN()

This macro is called to determine the minimum of two values.

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**.

Notes / Warnings

None.

Example

```
CPU_INT16S  x;
CPU_INT16S  y;
CPU_INT16S  z;


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

2.10.02.02 DEF_MAX()

This macro is called to determine the maximum of two values.

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**.

Notes / Warnings

None.

Example

```
CPU_INT16S  x;
CPU_INT16S  y;
CPU_INT16S  z;


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

2.10.02.03 DEF_ABS()

This macro is called to determine the absolute value of a value.

Prototype

```
DEF_ABS(a);
```

Arguments

a	Value to calculate absolute value.

Returned Value

The absolute value of **a**.

Notes / Warnings

None.

Example

```
CPU_INT16S  y;
CPU_INT16S  z;


y = -101;
z =  DEF_ABS(x, y);
```

Chapter 3

µC/LIB Memory Library

µC/LIB contains library functions that replace standard library memory functions such as **`memclr()`**, **`memset()`**, **`memcpy()`**, **`memcmp()`**, etc. These functions are defined in **`lib_mem.c`**.

3.00 µC/LIB Memory Library Configuration

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

`uC_CFG_OPTIMIZE_ASM_EN`	Implement certain functionality in assembly-optimized files (see Section 3.40).
`LIB_MEM_CFG_ARG_CHK_EXT_EN`	Includes code to check external arguments for functions called by the user. You can set this configuration constant to either **`DEF_DISABLED`** or **`DEF_ENABLED`**.
`LIB_MEM_CFG_ALLOC_EN`	Include memory allocation functionality (see Section 3.30).
`LIB_MEM_CFG_HEAP_SIZE`	Heap size, in octets (see Section 3.30).

3.10.01 MEM_VAL_GET_xxx()

These macro’s are called to decode data values from any CPU memory address.

Prototype

```
MEM_VAL_GET_xxx(addr);
```

Arguments

`addr`	This is the lowest CPU memory address of the data value to decode.

Returned Value

Decoded data value from CPU memory address.

Notes / Warnings

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

`MEM_VAL_GET_xxx_BIG()`	Decode big- endian data values -- data words' most significant octet @ lowest memory address
`MEM_VAL_GET_xxx_LITTLE()`	Decode little-endian data values -- data words' least significant octet @ lowest memory address
`MEM_VAL_GET_xxx()`	Decode data values using CPU's native or configured data-word order

2) CPU memory addresses/pointers **NOT** checked for `NULL`.

3) a) `MEM_VAL_GET_xxx()` macro's 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.

   b) However, any variable to receive the returned data value **MUST** start on an appropriate CPU word-aligned address.

4) `MEM_VAL_COPY_GET_xxx()` macro's are more efficient than `MEM_VAL_GET_xxx()` macro's & are also independent of CPU data-word-alignment & **SHOULD** be used whenever possible.

   See also Section 3.10.03 Note #4.

Example

```
CPU_INT08U  *pval;
CPU_INT16U   val;


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

3.10.02 MEM_VAL_SET_xxx()

These macro's are called to encode data values to any CPU memory address.

Prototype

```
MEM_VAL_SET_xxx(addr, val);
```

Arguments

`addr`	This is the lowest CPU memory address to encode the data value.
`val`	This is the data value to encode.

Returned Value

None.

Notes / Warnings

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

`MEM_VAL_SET_xxx_BIG()`	Encode big- endian data values -- data words' most significant octet @ lowest memory address
`MEM_VAL_SET_xxx_LITTLE()`	Encode little-endian data values -- data words' least significant octet @ lowest memory address
`MEM_VAL_SET_xxx()`	Encode data values using CPU's native or configured data-word order

2) CPU memory addresses/pointers **NOT** checked for `NULL`.

3) a) `MEM_VAL_SET_xxx()` macro's 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.

   b) However, `val` data value to encode **MUST** start on appropriate CPU word-aligned address.

4) `MEM_VAL_COPY_SET_xxx()` macro's are more efficient than `MEM_VAL_SET_xxx()` macro's & are also independent of CPU data-word-alignment & **SHOULD** be used whenever possible.

   See also Section 3.10.04 Note #4.

Example

```
CPU_INT08U  *pval;
CPU_INT16U   val;


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

3.10.03 MEM_VAL_COPY_GET_xxx()

These macro's are called to copy & decode data values from any CPU memory address to any other memory address.

Prototype

```
MEM_VAL_COPY_GET_xxx(addr_dest, addr_src);
```

Arguments

`addr_dest`	This is the lowest CPU memory address to copy/decode source address's data value.
`addr_src`	This is the lowest CPU memory address of the data value to copy/decode.

Returned Value

None.

Notes / Warnings

1) 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 @ lowest memory address
`MEM_VAL_COPY_GET_xxx_LITTLE()`	Decode little-endian data values -- data words' least significant octet @ lowest memory address
`MEM_VAL_COPY_GET_xxx()`	Decode data values using CPU's native or configured data-word order

2) CPU memory addresses/pointers **NOT** checked for `NULL`.

3) `MEM_VAL_COPY_GET_xxx()` macro's 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.

4) `MEM_VAL_COPY_GET_xxx()` macro's are more efficient than `MEM_VAL_GET_xxx()` macro's & are also independent of CPU data-word-alignment & **SHOULD** be used whenever possible.

Example

```
CPU_INT16U  *pmem;
CPU_INT16U  *pval;


pmem = &SomeAddr;                     /* Any CPU Address */
pval = &SomeVal;                      /* Any CPU Address */
MEM_VAL_COPY_GET_INT16U(pmem, pval);
```

3.10.04 MEM_VAL_COPY_SET_xxx()

These macro's are called to copy & encode data values from any CPU memory address to any other memory address.

Prototype

```
MEM_VAL_COPY_SET_xxx(addr_dest, addr_src);
```

Arguments

`addr_dest`	This is the lowest CPU memory address to copy/encode source address's data value.
`addr_src`	This is the lowest CPU memory address of the data value to copy/encode.

Returned Value

None.

Notes / Warnings

1) 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 @ lowest memory address
`MEM_VAL_COPY_SET_xxx_LITTLE()`	Encode little-endian data values -- data words' least significant octet @ lowest memory address
`MEM_VAL_COPY_SET_xxx()`	Encode data values using CPU's native or configured data-word order

2) CPU memory addresses/pointers **NOT** checked for `NULL`.

3) `MEM_VAL_COPY_SET_xxx()` macro's 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.

4) `MEM_VAL_COPY_SET_xxx()` macro's are more efficient than `MEM_VAL_SET_xxx()` macro's & are also independent of CPU data-word-alignment & **SHOULD** be used whenever possible.

Example

```
CPU_INT16U  *pmem;
CPU_INT16U  *pval;


pmem = &SomeAddr;                    /* Any CPU Address */
pval = &SomeVal;                     /* Any CPU Address */
MEM_VAL_COPY_SET_INT16U(pmem, pval);
```

3.10.05 MEM_VAL_COPY_xxx()

These macro's are called to copy data values from any CPU memory address to any other memory address.

Prototype

```
MEM_VAL_COPY_xxx(addr_dest, addr_src);
```

Arguments

`addr_dest`	This is the lowest CPU memory address to copy source address's data value.
`addr_src`	This is the lowest CPU memory address of the data value to copy.

Returned Value

None.

Notes / Warnings

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

2) CPU memory addresses/pointers **NOT** checked for `NULL`.

3) `MEM_VAL_COPY_xxx()` macro's 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.

Example

```
CPU_INT16U  *pmem;
CPU_INT16U  *pval;


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

3.20.01 Mem_Clr()

This function is called to clear a memory buffer. In other words, set all octets in the memory buffer to a value of '0'.

Prototype

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

Arguments

`pmem`	This is the pointer to the memory buffer to be clear.
`size`	This is the number of memory buffer octets to clear.

Returned Value

None.

Notes / Warnings

1) Zero-sized clears allowed.

Example

```
CPU_CHAR  AppBuf[10];


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

3.20.02 Mem_Set()

This function is called to fill a memory buffer with a specific value. In other words, set all octets in the memory buffer to the specific value.

Prototype

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

Arguments

`pmem`	This is the pointer to the memory buffer to be set with a specific value.
`data_val`	This is the value to set.
`size`	This is the number of memory buffer octets to set.

Returned Value

None.

Notes / Warnings

1) Zero-sized sets allowed.

Example

```
CPU_CHAR  AppBuf[10];


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

3.20.03 Mem_Copy()

This function is called to copy values from one memory buffer to another memory buffer.

Prototype

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

Arguments

`pdest`	This is the pointer to the memory buffer to copy octets into.
`psrc`	This is the pointer to the memory buffer to copy octets from.
`size`	This is the number of memory buffer octets to copy.

Returned Value

None.

Notes / Warnings

1) Zero-sized copies allowed.

2) Memory buffers **NOT** checked for overlapping.

3) This function can be configured to build an assembly-optimized version (see Sections 3.00 and 3.40.01).

Example

```
CPU_CHAR  AppBuf[10];


Mem_Copy((void     *)&AppBuf[0],
         (void     *)“ABCD”,
         (CPU_SIZE_T) Str_Len(“ABCD”));
```

3.20.04 Mem_Cmp()

This function is called to compare values from two memory buffers.

Prototype

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

Arguments

`p1_mem`	This is the pointer to the first memory buffer to compare.
`p2_mem`	This is the pointer to the second memory buffer to compare.
`size`	This is the 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.

Notes / Warnings

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

Example

```
CPU_CHAR     AppBuf[10];
CPU_BOOLEAN  cmp;


Mem_Copy((void     *)&AppBuf[0],
         (void     *)“ABCD”,
         (CPU_SIZE_T) Str_Len(“ABCD”));
cmp = Mem_Cmp((void     *)&AppBuf[2],
              (void     *)“CD”,
              (CPU_SIZE_T) Str_Len(“CD”));
```

3.30 µC/LIB Memory Allocation Functionality

The µC/LIB memory allocation functionality provides for the creation of memory pools from which blocks can be dynamically gotten and freed during application execution. Memory pool blocks can be allocated from either a general purpose-heap or from dedicated memory specified by the application. In addition, single memory blocks may be allocated directly from the heap.

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

`LIB_MEM_CFG_ALLOC_EN`	Must be configured to **`DEF_ENABLED`** to include memory allocation functionality and heap.
`LIB_MEM_CFG_HEAP_SIZE`	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.

3.30.01 Mem_HeapAlloc()

This function gets a single memory block from the heap.

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 …

a) Return the number of octets required to successfully allocate the memory block, if any errors;
b) Return `0`, otherwise.

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

`LIB_MEM_ERR_NONE`	Memory block successfully allocated.
`LIB_MEM_ERR_HEAP_EMPTY`	**NO** available memory in heap.
`LIB_MEM_ERR_INVALID_MEM_SIZE`	Invalid requested memory size.

Returned Value

Pointer to memory block,	if **NO** errors.
Pointer to **`NULL`**,	otherwise.

Notes / Warnings

None.

Example

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


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

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

3.30.02.01 Mem_PoolCreate()

This function is called to create a memory pool.

Prototype

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

Arguments

`pmem_pool` Pointer to a memory pool structure to create (see Note #1).

`pmem_base_addr` Memory pool base address:

a)	Null address	Memory pool allocated from general-purpose heap.
b)	Non-null address	Memory pool allocated from dedicated memory specified by its base address..

`mem_size` Size of memory pool segment (in octets).

`blk_nbr` Number of memory pool blocks to create.

`blk_size` Size of memory pool blocks to create (in octets).

`blk_align` Alignment of memory pool blocks to create (in octets).

`poctets_reqd` Pointer to a variable to …

a) Return the number of octets required to successfully allocate the memory pool, if any errors;
b) Return `0`, otherwise.

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

`LIB_MEM_ERR_NONE`	Memory pool successfully created.
`LIB_MEM_ERR_NULL_PTR`	Argument `pmem_pool` passed a `NULL` pointer.
`LIB_MEM_ERR_HEAP_NOT_FOUND`	Heap segment **NOT** found.
`LIB_MEM_ERR_HEAP_EMPTY`	**NO** available memory in heap segment.
`LIB_MEM_ERR_SEG_EMPTY`	**NO** available memory in memory segment.
`LIB_MEM_ERR_INVALID_SEG_SIZE`	Invalid memory segment size.

LIB_MEM_ERR_INVALID_SEG_OVERLAP	Memory segment overlaps other memory segment(s) in memory pool table.
LIB_MEM_ERR_INVALID_BLK_NBR	Invalid memory pool number of blocks.
LIB_MEM_ERR_INVALID_BLK_SIZE	Invalid memory pool block size.

Returned Value

None.

Notes / Warnings

1) **pmem_pool** **MUST** be passed a pointer to the address of a declared **MEM_POOL** variable.

Example

```
MEM_POOL  AppMemPoolFromHeap;
MEM_POOL  AppMemPoolFromUserMemSeg;


void AppFnct (void)
{
    CPU_SIZE_T  octets_reqd;
    LIB_ERR     err;
    :
    :
    Mem_PoolCreate((MEM_POOL   *)&AppMemPoolFromHeap,
                   (void       *)  0,       /* Create pool from heap   ...                    */
                   (CPU_SIZE_T  )  0,
                   (CPU_SIZE_T  ) 10,       /* ... With 10 blocks      ...                    */
                   (CPU_SIZE_T  )100,       /* ... Of 100 octets each  ...                    */
                   (CPU_SIZE_T  )  4,       /* ... 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   ... %d more octets needed.", octets_reqd);
        }
    }
    :
    :
    Mem_PoolCreate((MEM_POOL   *)&AppMemPoolFromUserMemSeg,
                   (void       *)0x21000000, /* Create pool from memory at 0x21000000 ...      */
                   (CPU_SIZE_T  )10000,      /* ... From a 10000-octet segment        ...      */
                   (CPU_SIZE_T  )   10,      /* ... With 10 blocks                    ...      */
                   (CPU_SIZE_T  )  100,      /* ... Of 100 octets each                ...      */
                   (CPU_SIZE_T  )    4,      /* ... 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   ... %d more octets needed.", octets_reqd);
        } else if (err == LIB_MEM_ERR_SEG_EMPTY) {
            printf("Segment full ... %d more octets needed.", octets_reqd);
        }
    }
    :
    :
}
```

3.30.02.02 Mem_PoolBlkGet()

This function gets a memory block from memory pool.

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`	Memory block successfully returned.
`LIB_MEM_ERR_POOL_EMPTY`	**NO** memory blocks available in memory pool.
`LIB_MEM_ERR_NULL_PTR`	Argument `pmem_pool` passed a `NULL` pointer.
`LIB_MEM_ERR_INVALID_POOL`	Invalid memory pool type.
`LIB_MEM_ERR_INVALID_BLK_SIZE`	Invalid memory pool block size requested.
`LIB_MEM_ERR_INVALID_BLK_IX`	Invalid memory pool block index.

Returned Value

Pointer to memory block,	if **NO** errors.
Pointer to `NULL`,	otherwise.

Notes / Warnings

None.

Example

```
MEM_POOL  AppMemPool;


void AppFnct (void)
{
    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  )  0,
                   (CPU_SIZE_T  ) 10,       /* ... With 10 blocks      ...                      */
                   (CPU_SIZE_T  )100,       /* ... Of 100 octets each  ...                      */
                   (CPU_SIZE_T  )  4,       /* ... 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   ... %d more octets needed.", octets_reqd);
        return;
    }
    :
    :
                                            /* Get a 100-byte memory block from the pool.     */
    pmem_blk = Mem_PoolBlkGet((MEM_POOL *)&AppMemPool,
                              (CPU_SIZE_T) 100,
                              (LIB_ERR  *)&err);

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

3.30.02.03 Mem_PoolBlkFree()

This function frees a memory block to memory pool.

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`	Memory block successfully freed.
`LIB_MEM_ERR_POOL_FULL`	ALL memory blocks already available in pool.
`LIB_MEM_ERR_NULL_PTR`	Argument `pmem_pool`/`pmem_blk` passed a `NULL` pointer.
`LIB_MEM_ERR_INVALID_POOL`	Invalid memory pool type.
`LIB_MEM_ERR_INVALID_BLK_ADDR`	Invalid memory block address.
`LIB_MEM_ERR_INVALID_BLK_ADDR_IN_POOL`	Memory pool address already in memory pool.

Returned Value

None.

Notes / Warnings

None.

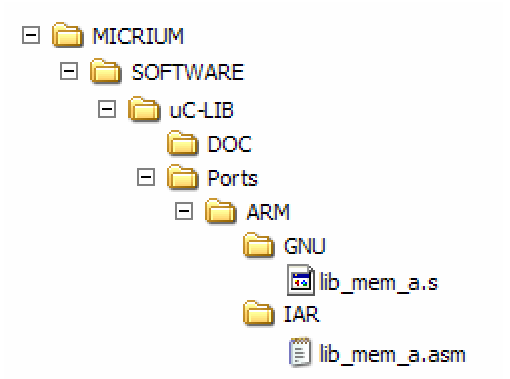
Example

```
MEM_POOL  AppMemPool;


void AppFnct (void)
{
    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  )  0,
                   (CPU_SIZE_T  ) 10,       /* ... With 10 blocks      ...                      */
                   (CPU_SIZE_T  )100,       /* ... Of 100 octets each  ...                      */
                   (CPU_SIZE_T  )  4,       /* ... 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   ... %d more octets needed.”, octets_reqd);
        return;
    }
    :
    :
                                            /* Get a 100-byte memory block from the pool.     */
    pmem_blk = Mem_PoolBlkGet((MEM_POOL *)&AppMemPool,
                              (CPU_SIZE_T) 100,
                              (LIB_ERR  *)&err);

    if (err != LIB_ERR_NONE) {
        printf(“COULD NOT GET MEMORY BLOCK FROM MEMORY POOL.”);
    }
    :
    :
                                            /* Free  100-byte memory block back to  pool.     */
    Mem_PoolBlkFree((MEM_POOL *)&AppMemPool,
                    (void     *) pmem_blk,
                    (LIB_ERR  *)&err);

    if (err != LIB_ERR_NONE) {
        printf(“COULD NOT FREE MEMORY BLOCK TO MEMORY POOL.”);
    }
}
```

3.40 µC/LIB 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 3-1 for an example port directory :



```
MICRIUM
  SOFTWARE
    uC-LIB
      DOC
      Ports
        ARM
          GNU
            lib_mem_a.s
          IAR
            lib_mem_a.asm
```

Figure 3-1, µC/LIB Example Port Directory

3.40.01 Mem_Copy() Optimization

Future Release

Chapter 4

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`**.

4.00 µC/LIB String Library Configuration

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

`LIB_STR_CFG_FP_EN`	Enable floating-point string conversion functions (see Section 4.10.09).

4.10.01 Str_Len()

This function is called to determine the length of a string.

Prototype

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

Arguments

`pstr`	This is the pointer to the string.

Returned Value

Length of string in number of characters in string before but **NOT** including the terminating **`NULL`** character.

Notes / Warnings

1) String buffer **NOT** modified.

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

Example

```
CPU_INT16U  len;

len = (CPU_INT16U)Str_Len(“Hello World.”);
```

4.10.02.01 Str_Copy()

This function is called to copy string character values from one string memory buffer to another memory buffer.

Prototype

```
CPU_CHAR   *Str_Copy (CPU_CHAR  *pdest,
                      CPU_CHAR  *psrc);
```

Arguments

`pdest`	This is the pointer to the string memory buffer to copy string characters into.
`psrc`	This is the 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.

Notes / Warnings

1) Destination buffer size **NOT** validated; buffer overruns **MUST** be prevented by caller.

   a) Destination buffer size **MUST** be large enough to accomodate the entire source string size including the terminating **`NULL`** character.

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

Example

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


pstr = Str_Copy(&AppBuf[0], “Hello World!”);
```

4.10.02.02 Str_Copy_N()

This function is called to copy string character values from one string memory buffer to another memory buffer, up to a maximum number of characters.

Prototype

```
CPU_CHAR   *Str_Copy_N (CPU_CHAR    *pdest,
                        CPU_CHAR    *psrc,
                        CPU_SIZE_T   len_max);
```

Arguments

`pdest`	This is the pointer to the string memory buffer to copy string characters into.
`psrc`	This is the pointer to the string memory buffer to copy string characters from.
`len_max`	This is the maximum number of string characters to copy.

Returned Value

Pointer to copied destination string,	if **NO** errors.
Pointer to **`NULL`**,	otherwise.

Notes / Warnings

1) Destination buffer size **NOT** validated; buffer overruns **MUST** be prevented by caller.

   a) Destination buffer size **MUST** be large enough to accomodate the entire source string size including the terminating **`NULL`** character.

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

3) The maximum number of characters copied does **NOT** include the terminating **`NULL`** character.

Example

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


pstr = Str_Copy_N(&AppBuf[0], “Hello World!”, 6);
```

4.10.03.01 Str_Cat()

This function is called to concatenate a string to the end of another string.

Prototype

```
CPU_CHAR  *Str_Cat (CPU_CHAR  *pdest,
                    CPU_CHAR  *pstr_cat);
```

Arguments

`pdest`	This is the pointer to the string memory buffer to append string characters into.
`pstr_cat`	This is the pointer to the string to concatenate onto the destination string.

Returned Value

Pointer to concatenated destination string,	if **NO** errors.
Pointer to **`NULL`**,	otherwise.

Notes / Warnings

1) Destination buffer size **NOT** validated; buffer overruns **MUST** be prevented by caller.

   a) Destination buffer size **MUST** be large enough to accomodate the entire source string size including the terminating **`NULL`** character.

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

Example

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


pstr = Str_Copy(&AppBuf[0], “Hello   World!”);
pstr = Str_Cat (&AppBuf[0], “Goodbye World!”);
```

4.10.03.02 Str_Cat_N()

This function is called to concatenate a string to the end of another string, up to a maximum number of characters.

Prototype

```
CPU_CHAR  *Str_Cat_N (CPU_CHAR    *pdest,
                      CPU_CHAR    *pstr_cat,
                      CPU_SIZE_T   len_max);
```

Arguments

`pdest`	This is the pointer to the string memory buffer to append string characters into.
`pstr_cat`	This is the pointer to the string to concatenate onto the destination string.
`len_max`	This is the maximum number of string characters to concatenate.

Returned Value

Pointer to concatenated destination string,	if **NO** errors.
Pointer to **`NULL`**,	otherwise.

Notes / Warnings

1) Destination buffer size **NOT** validated; buffer overruns **MUST** be prevented by caller.

    a) Destination buffer size **MUST** be large enough to accomodate the entire source string size including the terminating **`NULL`** character.

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

3) The maximum number of characters concatenated does **NOT** include the terminating **`NULL`** character.

Example

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


pstr = Str_Copy (&AppBuf[0], “Hello   World!”);
pstr = Str_Cat_N(&AppBuf[0], “Goodbye World!”, 8);
```

4.10.04.01 Str_Cmp()

This function is called to determine if two strings are identical.

Prototype

```
CPU_INT16S  Str_Cmp (CPU_CHAR  *p1_str,
                     CPU_CHAR  *p2_str);
```

Arguments

`p1_str`	This is the pointer to the first string.
`p2_str`	This is the pointer to the second string.

Returned Value

Zero value,	if strings are identical; i.e. both strings are identical in length and **ALL** 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.

Notes / Warnings

1) String buffers **NOT** modified.

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

3) 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

```
CPU_INT16S  cmp;


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

4.10.04.02 Str_Cmp_N()

This function is called to determine if two strings are identical for a specified length of characters.

Prototype

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

Arguments

`p1_str`	This is the pointer to the first string.
`p2_str`	This is the pointer to the second string.
`len_max`	This is the 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.

Notes / Warnings

1) String buffers **NOT** modified.

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

3) 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

```
CPU_INT16S  cmp;


cmp = Str_Cmp_N(“Hello World!”, “Hello World.”, 11);
```

4.10.04.03 Str_CmpIgnoreCase()

This function is called to determine if two strings are identical, ignoring case. It behaves as if the two strings were converted to lower case and then compared with `Str_Cmp()`.

Prototype

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

Arguments

`p1_str`	This is the pointer to the first string.
`p2_str`	This is the pointer to the second string.

Returned Value

Zero value,	if strings are identical (ignoring case); i.e. both strings are identical in length and **ALL** characters (ignoring case).
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.

Notes / Warnings

1) String buffers **NOT** modified.

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

3) 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

```
CPU_INT16S  cmp;


cmp = Str_CmpIgnoreCase(“Hello World!”, “hElLo WoRlD!”);
```

4.10.04.04 Str_CmpIgnoreCase_N()

This function is called to determine if two strings are identical for a specified length of characters, ignoring case. It behaves as if the two strings were converted to lower case and then compared with `Str_Cmp_N()`.

Prototype

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

Arguments

`p1_str`	This is the pointer to the first string.
`p2_str`	This is the pointer to the second string.
`len_max`	This is the 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.

Notes / Warnings

1) String buffers **NOT** modified.

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

3) 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

```
CPU_INT16S  cmp;

cmp = Str_CmpIgnoreCase_N(“Hello World!”, “hElLo WoRlD.”, 11);
```

4.10.05.01 Str_Char()

This function is called to find the first occurrence of a specific character in a string.

Prototype

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

Arguments

`pstr`	This is the pointer to the string to search for the specified character.
`srch_char`	This is the character to search for in the string.

Returned Value

Pointer to first occurrence of character in string,	if **NO** errors.
Pointer to **NULL**,	otherwise.

Notes / Warnings

1) String buffer **NOT** modified.

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

Example

```
CPU_CHAR  *pstr;

pstr = Str_Char(“Hello World!”, 'l');
```

4.10.05.02 Str_Char_N()

This function is called to find the first occurrence of a specific character in a string, up to a maximum number of characters.

Prototype

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

Arguments

`pstr`	This is the pointer to the string to search for the specified character.
`len_max`	This is the maximum number of string characters to search.
`srch_char`	This is the character to search for in the string.

Returned Value

Pointer to first occurrence of character in string,	if **NO** errors.
Pointer to **`NULL`**,	otherwise.

Notes / Warnings

1) String buffer **NOT** modified.

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

Example

```
CPU_CHAR  *pstr;

pstr = Str_Char_N(“Hello World!”, 'l', 5);
```

4.10.05.03 Str_Char_Last()

This function is called to find the last occurrence of a specific character in a string.

Prototype

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

Arguments

`pstr`	This is the pointer to the string to search for the specified character.
`srch_char`	This is the character to search for in the string.

Returned Value

Pointer to first occurrence of character in string,	if **NO** errors.
Pointer to **`NULL`**,	otherwise.

Notes / Warnings

1) String buffer **NOT** modified.

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

Example

```
CPU_CHAR  *pstr;

pstr = Str_Char_Last(“Hello World!”, 'l');
```

4.10.06 Str_Str()

This function is called to find the first occurrence of a specific string within another string.

Prototype

```
CPU_CHAR  *Str_Str (CPU_CHAR  *pstr,
                    CPU_CHAR  *psrch_str);
```

Arguments

`pstr`	This is the pointer to the string to search for the specified string.
`psrch_str`	This is the pointer to the string to search for in the string.

Returned Value

Pointer to first occurrence of search string in string,	if **NO** errors.
Pointer to **`NULL`**,	otherwise.

Notes / Warnings

1) String buffers **NOT** modified.

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

Example

```
CPU_CHAR  *pstr;

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

4.10.07.01 Str_FmtNbr_Int32U()

This function is called to convert & format a 32-bit unsigned integer into a string.

Prototype

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

Arguments

`nbr`	This is the number to format into a string.
`nbr_dig`	This is the number of integer digits to format into the number string (see Notes #1 & #6).
`nbr_base`	This is the base of the number to format into the number string (see Note #2).
`lead_char`	Option to prepend a leading character into the formatted number string (see Note #3).
`lower_case`	Option to format any alphabetic characters (if any) in lower case.
`nul`	Option to **`NULL`**-terminate the formatted number string (see Note #4).
`pstr`	This is the pointer to the string memory buffer to return the formatted number string (see Note #5).

Returned Value

Pointer to formatted number string,	if **NO** errors.
Pointer to **`NULL`**,	otherwise.

Notes / Warnings

1) The following constants may be used to specify the number of digits :

`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

2) The number's base **MUST** be between 2 & 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

3) a) Leading character option prepends leading characters prior to the first non-zero digit. The number of leading characters is such that the total number of integer digits is equal to the requested number of integer digits to format (`nbr_dig`).

b) Leading character **MUST** be a printable ASCII character.

c) 1) If the value of the number to format is zero
2) ... & the number of digits to format is non-zero,
3) ... but **NO** leading character available;
4) ... then one digit of '`0`' value is formatted.

This is **NOT** a leading character; but a single integer digit of '`0`' value.

4) a) `NULL`-character terminate option **DISABLED** prevents overwriting previous character array formatting.

b) **WARNING**: Unless `pstr` character array is pre-/post-terminated, `NULL`-character terminate option **DISABLED** will cause character string run-on.

5) a) Format buffer size **NOT** validated; buffer overruns **MUST** be prevented by caller.

b) To prevent character buffer overrun :

Character array size **MUST** be >= (`nbr_dig` +
1 `NUL` terminator) characters

6) 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 the most-significant digits of the formatted number will be truncated.

Example :

```
nbr     = 23456
nbr_dig = 3
pstr    = “456”
```

Example

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


pstr = Str_FmtNbr_Int32U((CPU_INT32U ) 12345678,
                         (CPU_INT08U )  5,
                         (CPU_INT08U ) 10,
                         (CPU_BOOLEAN) DEF_YES,
                         (CPU_BOOLEAN) DEF_NO,
                         (CPU_BOOLEAN) DEF_YES,
                         (CPU_CHAR   *)&AppBuf[0]);
```

4.10.07.02 Str_FmtNbr_Int32S()

This function is called to convert & format a 32-bit signed integer into a string.

Prototype

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

Arguments

`nbr`	This is the number to format into a string.
`nbr_dig`	This is the number of integer digits to format into the number string (see Notes #1 & #6).
`nbr_base`	This is the base of the number to format into the number string (see Note #2).
`lead_char`	Option to prepend a leading character into the formatted number string (see Note #3).
`lower_case`	Option to format any alphabetic characters (if any) in lower case.
`nul`	Option to **`NULL`**-terminate the formatted number string (see Note #4).
`pstr`	This is the pointer to the string memory buffer to return the formatted number string (see Note #5).

Returned Value

Pointer to formatted number string,	if **NO** errors.
Pointer to **`NULL`**,	otherwise.

Notes / Warnings

1) The following constants may be used to specify the number of digits :

`DEF_INT_32S_NBR_DIG_MIN`	Minimum number of 32-bit signed digits
`DEF_INT_32S_NBR_DIG_MAX`	Maximum number of 32-bit signed digits

2) The number's base **MUST** be between 2 & 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

3) a) Leading character option prepends leading characters prior to the first non-zero digit. The number of leading characters is such that the total number of integer digits is equal to the requested number of integer digits to format (`nbr_dig`).

b) Leading character **MUST** be a printable ASCII character.

c) 1) If the value of the number to format is zero
2) ... & the number of digits to format is non-zero,
3) ... but **NO** leading character available;
4) ... then one digit of '`0`' value is formatted.

This is **NOT** a leading character; but a single integer digit of '`0`' value.

4) a) `NULL`-character terminate option **DISABLED** prevents overwriting previous character array formatting.

b) **WARNING**: Unless `pstr` character array is pre-/post-terminated, `NULL`-character terminate option **DISABLED** will cause character string run-on.

5) a) Format buffer size **NOT** validated; buffer overruns **MUST** be prevented by caller.

b) To prevent character buffer overrun :

Character array size **MUST** be >= (`nbr_dig` +
1 `NUL` terminator) characters

6) a) 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 the most-significant digits of the formatted number will be truncated.

Example :

```
nbr     = 23456
nbr_dig = 3
pstr    = “456”
```

b) If number to format (`nbr`) is negative but the most-significant digits of the formatted number are truncated (see Note #2a); the negative sign still prefixes the truncated formatted number.

Example :

```
nbr     = -23456
nbr_dig =  3
pstr    = “-456”
```

Example

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


pstr = Str_FmtNbr_Int32S((CPU_INT32U )-12345678,
                         (CPU_INT08U )  5,
                         (CPU_INT08U ) 10,
                         (CPU_BOOLEAN) DEF_YES,
                         (CPU_BOOLEAN) DEF_NO,
                         (CPU_BOOLEAN) DEF_YES,
                         (CPU_CHAR  *)&AppBuf[0]);
```

4.10.07.03 Str_FmtNbr_32()

This function is called to convert & format a 32-bit floating point number into a string.

Prototype

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

Arguments

`nbr`	This is the number to format into a string.
`nbr_dig`	This is the number of integer digits to format into the number string (see Note #5).
`nbr_dp`	This is the number of decimal digits to format into the number string.
`lead_char`	Option to prepend a leading character into the formatted number string (see Note #2).
`nul`	Option to **`NULL`**-terminate the formatted number string (see Note #3).
`pstr`	This is the pointer to the string memory buffer to return the formatted number string (see Note #4).

Returned Value

Pointer to formatted number string,	if **NO** errors.
Pointer to **`NULL`**,	otherwise.

Notes / Warnings

1) This function enabled **ONLY** if **`LIB_STR_CFG_FP_EN`** enabled in **`app_cfg.h`** (see Section 4.00).

2) a) Leading character option prepends leading characters prior to the first non-zero digit. The number of leading characters is such that the total number of integer digits is equal to the requested number of integer digits to format (**`nbr_dig`**).

b) Leading character **MUST** be a printable ASCII character.

c) 1) If the integer value of the number to format is zero
2) ... & the number of digits to format is non-zero,
3) ... but **NO** leading character available;
4) ... then one digit of '`0`' value is formatted.

This is **NOT** a leading character; but a single integer digit of '`0`' value.

3) a) NULL-character terminate option **DISABLED** prevents overwriting previous character array formatting.

b) **WARNING**: Unless pstr character array is pre-/post-terminated, NULL-character terminate option **DISABLED** will cause character string run-on.

4) a) Format buffer size **NOT** validated; buffer overruns **MUST** be prevented by caller.

b) To prevent character buffer overrun :

```
Character array size MUST be  >=  (nbr_dig        +
                                   nbr_dp         +
                                   1 negative sign  +
                                   1 decimal point  +
                                   1 NUL terminator)  characters
```

5) a) 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 the most-significant digits of the formatted number will be truncated.

Example :

```
nbr     = 23456.789
nbr_dig = 3
nbr_dp  = 2
pstr    = “456.78”
```

b) If number to format (nbr) is negative but the most-significant digits of the formatted number are truncated (see Note #2a); the negative sign still prefixes the truncated formatted number.

Example :

```
nbr     = -23456.789
nbr_dig =  3
nbr_dp  =  2
pstr    = “-456.78”
```

Example

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


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

4.10.08.01 Str_ParseNbr_Int32U()

This function is called to parse a 32-bit unsigned integer from a string.

Prototype

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

Arguments

`pstr`	Pointer to string (see Notes #1 & #3).
`pstr_end`	Pointer to a variable to …

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

`nbr_base`	Base of number to parse (see Note #2).

Returned Value

Parsed integer,	if integer parsed with **NO** overflow.
`DEF_INT_32U_MAX_VAL`,	if integer parsed but overflowed.
`0`,	otherwise.

Notes / Warnings

1) String buffer **NOT** modified.

2) Base passed may be:

   a) Zero. The actual base will be determined from the integer string (see Note #3c):

      1) If the integer string begins with `“0x”` or `“0x”`, the base is 16.
      2) If the integer string begins with `“0”` but **NOT** `“0x”` or `“0X”`, the base is 8.
      3) Otherwise, the base is 10.

   b) Integer between 2 & 36, inclusive

3) The input string consists of:

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

   b) An optional sign character (`‘+’`).
      1) A negative sign character (`‘-’`) will be interpreted as an invalid character.

   c) A sequence of characters representing an integer in some radix:
      1) If the base is 16, one of the optional character sequences `“0x”` or `“0X”`.

2) A sequence of letters & digits. The letters from `'a'` (or `'A'`) to `'z'` (or `'Z'`) are assigned the values 10 through 35. Only letters & digits whose assigned values are less than that of the base are valid.

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

4) Return integer value & next string pointer should be used to diagnose parse success or failure :

a) Valid parse string integer :

```
pstr      = "      ABCDE xyz"
nbr_base  = 16

nbr       = 703710
pstr_next = " xyz"
```

b) Invalid parse string integer :

```
pstr      = "      ABCDE"
nbr_base  = 10

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

c) Valid hexadecimal parse string integer :

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

nbr       =  0
pstr_next = "xGABCDE"
```

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

```
pstr      = "      0xGABCDE"
nbr_base  =  0

nbr       =  0
pstr_next = "xGABCDE"
```

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

```
pstr      = "      0GABCDE"
nbr_base  =  0

nbr       =  0
pstr_next = "GABCDE"
```

f) Parse string integer overflow :

```
pstr      = "    12345678901234567890*123456"
nbr_base  = 10

nbr       = DEF_INT_32U_MAX_VAL
pstr_next = "*123456"
```

g) Invalid negative unsigned parse string :

```
pstr      = "  -12345678901234567890*123456"
nbr_base  = 10

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

Example

```
CPU_INT32U   nbr;
CPU_CHAR    *pstr_end;


nbr = Str_ParseNbr_Int32U((CPU_CHAR  *)"0x1234534*023434>345344",
                          (CPU_CHAR **)&pstr_end,
                          (CPU_INT08U ) 10);
```

4.10.08.02 Str_ParseNbr_Int32S()

This function is called to parse a 32-bit signed integer from a string.

Prototype

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

Arguments

`pstr`	Pointer to string (see Notes #1 & #3).
`pstr_end`	Pointer to a variable to …

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

`nbr_base`	Base of number to parse (see Note #2).

Returned Value

Parsed integer,	if integer could be parsed & overflow did **NOT** occur.
`DEF_INT_32S_MAX_VAL`,	if integer could be parsed & overflow did occur & integer is positive.
`DEF_INT_32S_MIN_VAL`,	if integer could be parsed & overflow did occur & integer is negative.
`0`,	otherwise.

Notes / Warnings

1) String buffer **NOT** modified.

2) Base passed may be:

a) Zero. The actual base will be determined from the integer string (see Note #3c):

1) If the integer string begins with `“0x”` or `“0x”`, the base is 16.
2) If the integer string begins with `“0”` but NOT `“0x”` or `“0X”`, the base is 8.
3) Otherwise, the base is 10.

b) Integer between 2 & 36, inclusive

3) The input string consists of:

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

b) An optional sign character (`‘-’` or `‘+’`).

c) A sequence of characters representing an integer in some radix:
1) If the base is 16, one of the optional character sequences `“0x”` or `“0X”`.

2) A sequence of letters & digits. The letters from `'a'` (or `'A'`) to `'z'` (or `'Z'`) are assigned the values 10 through 35. Only letters & digits whose assigned values are less than that of the base are valid.

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

4) Return integer value & next string pointer should be used to diagnose parse success or failure :

a) Valid parse string integer :

```
pstr      = "     -ABCDE xyz"
nbr_base  = 16

nbr       = -703710
pstr_next = " xyz"
```

b) Invalid parse string integer :

```
pstr      = "      ABCDE"
nbr_base  = 10

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

c) Valid hexadecimal parse string integer :

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

nbr       =  0
pstr_next = "xGABCDE"
```

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

```
pstr      = "      0xGABCDE"
nbr_base  =  0

nbr       =  0
pstr_next = "xGABCDE"
```

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

```
pstr      = "      0GABCDE"
nbr_base  =  0

nbr       =  0
pstr_next = "GABCDE"
```

f) Parse string integer overflow :

```
pstr      = "    12345678901234567890*123456"
nbr_base  = 10

nbr       = DEF_INT_32S_MAX_VAL
pstr_next = "*123456"
```

g) Parse string integer underflow :

```
pstr       = “  -12345678901234567890*123456”
nbr_base   = 10

nbr        = DEF_INT_32S_MIN_VAL
pstr_next  = “*123456”
```

Example

```
CPU_INT32U   nbr;
CPU_CHAR    *pstr_end;


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

Chapter 5

µC/LIB ASCII Library

µC/LIB contains library functions that replace standard library character functions such as **`tolower()`**, **`toupper()`**, **`isalpha()`**, **`isdigit()`**, etc. These functions are defined in **`lib_ascii.c`**.

5.00 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 character values or variables.

5.10 Functions & Macros

µC/LIB contains many common character classification and case conversion functions & macros. Character classification functions & macros determine whether a character belongs to a certain class of character (e.g., uppercase alphabetic characters). Character case conversion functions & macros convert a character from uppercase to lowercase or lowercase to uppercase.

5.10.01 ASCII_IS_ALPHA() / ASCII_IsAlpha()

These determine whether a character is an alphabetic character.

Macro Prototype

```
ASCII_IS_ALPHA(c);
```

Function Prototype

```
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.

Notes / Warnings

1) 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

```
CPU_CHAR     c;
CPU_BOOLEAN  alpha;


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

5.10.02 ASCII_IS_ALPHA_NUM() / ASCII_IsAlphaNum()

These determine whether a character is an alphanumeric character.

Macro Prototype

```
ASCII_IS_ALPHA_NUM(c);
```

Function Prototype

```
CPU_BOOLEAN  ASCII_IsAlpaNum (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.

Notes / Warnings

1) 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

```
CPU_CHAR     c;
CPU_BOOLEAN  alpha_num;


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

5.10.03 ASCII_IS_LOWER() / ASCII_IsLower()

These determine whether a character is a lowercase alphabetic character.

Macro Prototype

```
ASCII_IS_LOWER(c);
```

Function Prototype

```
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.

Notes / Warnings

1) ISO/IEC 9899:TC2, Section 7.4.1.7.(2) states that "`islower()` returns true only for the lowercase letters".

Example

```
CPU_CHAR     c;
CPU_BOOLEAN  lower;


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

5.10.04 ASCII_IS_UPPER() / ASCII_IsUpper()

These determine whether a character is an uppercase alphabetic character.

Macro Prototype

```
ASCII_IS_UPPER(c);
```

Function Prototype

```
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.

Notes / Warnings

1) ISO/IEC 9899:TC2, Section 7.4.1.11.(2) states that "`isupper()` returns true only for the uppercase letters".

Example

```
CPU_CHAR     c;
CPU_BOOLEAN  upper;


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

5.10.05 ASCII_IS_DIG() / ASCII_IsDig()

These determine whether a character is a decimal-digit character.

Macro Prototype

```
ASCII_IS_DIG(c);
```

Function Prototype

```
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.

Notes / Warnings

1) ISO/IEC 9899:TC2, Section 7.4.1.5.(2) states that "`isdigit()` … tests for any decimal-digit character".

Example

```
CPU_CHAR     c;
CPU_BOOLEAN  dig;


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

5.10.06 ASCII_IS_DIG_OCT() / ASCII_IsDigOct()

These determine whether a character is an octal-digit character.

Macro Prototype

```
ASCII_IS_DIG_OCT(c);
```

Function Prototype

```
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.

Notes / Warnings

None.

Example

```
CPU_CHAR     c;
CPU_BOOLEAN  dig_oct;


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

5.10.07 ASCII_IS_DIG_HEX() / ASCII_IsDigHex()

These determine whether a character is a hexadecimal-digit character.

Macro Prototype

```
ASCII_IS_DIG_HEX(c);
```

Function Prototype

```
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.

Notes / Warnings

1) ISO/IEC 9899:TC2, Section 7.4.1.12.(2) states that "`isxdigit()` … tests for any hexadecimal-digit character".

Example

```
CPU_CHAR     c;
CPU_BOOLEAN  dig_hex;


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

5.10.08 ASCII_IS_BLANK() / ASCII_IsBlank()

These determine whether a character is a standard blank character.

Macro Prototype

```
ASCII_IS_BLANK(c);
```

Function Prototype

```
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.

Notes / Warnings

1) a) ISO/IEC 9899:TC2, Section 7.4.1.3.(2) states that "`isblank()` returns true only for the standard blank characters".

b) ISO/IEC 9899:TC2, Section 7.4.1.3.(2) defines "the standard blank characters" as the "space (`' '`), and horizontal tab (`'\t'`)".

Example

```
CPU_CHAR     c;
CPU_BOOLEAN  blank;


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

5.10.09 ASCII_IS_SPACE() / ASCII_IsSpace()

These determine whether a character is a white-space character.

Macro Prototype

```
ASCII_IS_SPACE(c);
```

Function Prototype

```
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.

Notes / Warnings

1) a) ISO/IEC 9899:TC2, Section 7.4.1.10.(2) states that "**`isspace()`** returns true only for the standard white-space characters".

   b) 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

```
CPU_CHAR     c;
CPU_BOOLEAN  space;


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

5.10.10 ASCII_IS_PRINT() / ASCII_IsPrint()

These determine whether a character is a printing character.

Macro Prototype

```
ASCII_IS_PRINT(c);
```

Function Prototype

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

Arguments

`c`	Character to examine.

Returned Value

`DEF_YES`, if the character is a printing character.

`DEF_NO`, if the character is **NOT** a printing character.

Notes / Warnings

1) a) ISO/IEC 9899:TC2, Section 7.4.1.8.(2) states that "`isprint()` ... tests for any printing character including space (`'a'`)".

   b) 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

```
CPU_CHAR     c;
CPU_BOOLEAN  print;


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

5.10.11 ASCII_IS_GRAPH() / ASCII_IsGraph()

These determine whether a character is a graphic character

Macro Prototype

```
ASCII_IS_GRAPH(c);
```

Function Prototype

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

Arguments

c	Character to examine.

Returned Value

DEF_YES,	if the character is a graphic character.
DEF_NO,	if the character is **NOT** a graphic character.

Notes / Warnings

1) a) ISO/IEC 9899:TC2, Section 7.4.1.6.(2) states that "`isgraph()` ... tests for any printing character except space (' ')".

   b) 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

```
CPU_CHAR     c;
CPU_BOOLEAN  graph;


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

5.10.12 ASCII_IS_PUNCT() / ASCII_IsPunct()

These determine whether a character is a punctuation character

Macro Prototype

```
ASCII_IS_PUNCT(c);
```

Function Prototype

```
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.

Notes / Warnings

1) ISOISO/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

```
CPU_CHAR     c;
CPU_BOOLEAN  punct;


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

5.10.13 ASCII_IS_CTRL() / ASCII_IsCtrl()

These determine whether a character is a control character

Macro Prototype

```
ASCII_IS_CTRL(c);
```

Function Prototype

```
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.

Notes / Warnings

1) a) ISO/IEC 9899:TC2, Section 7.4.1.4.(2) states that "**iscntrl()** ... tests for any control character".

   b) 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

```
CPU_CHAR     c;
CPU_BOOLEAN  ctrl;


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

5.10.14 ASCII_TO_LOWER() / ASCII_ToLower()

These convert an uppercase alphabetic character to its corresponding lowercase alphabetic character.

Macro Prototype

```
ASCII_TO_LOWER(c);
```

Function Prototype

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

Arguments

c	Character to convert.

Returned Value

Uppercase equivalent of **c**,	if character **c** is an uppercase character.
Character **c**,	otherwise.

Notes / Warnings

1) a) ISO/IEC 9899:TC2, Section 7.4.2.1.(2) states that "**tolower()** ... converts an uppercase letter to a corresponding lowercase letter".

   b) ISO/IEC 9899:TC2, Section 7.4.2.1.(3) states that:

      1) "if the argument is a character for which **isupper()** is true and there are one or more corresponding characters ... for which **islower()** is true," ...

      2) "**tolower()** ... returns one of the corresponding characters;" ...

      3) "otherwise, the argument is returned unchanged."

Example

```
CPU_CHAR  c;
CPU_CHAR  c_lower;


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

5.10.15 ASCII_TO_UPPER() / ASCII_ToUpper()

These convert a lowercase alphabetic character to its corresponding uppercase alphabetic character.

Macro Prototype

```
ASCII_TO_UPPER(c);
```

Function Prototype

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

Arguments

c	Character to convert.

Returned Value

Uppercase equivalent of **c**,	if character **c** is a lowercase character.
Character **c**,	otherwise.

Notes / Warnings

1) a) ISO/IEC 9899:TC2, Section 7.4.2.2.(2) states that "**toupper()** ... converts a lowercase letter to a corresponding uppercase letter".

   b) ISO/IEC 9899:TC2, Section 7.4.2.2.(3) states that:

      1) "if the argument is a character for which **islower()** is true and there are one or more corresponding characters ... for which **isupper()** is true," ...

      2) "**toupper()** ... returns one of the corresponding characters;" ...

      3) "otherwise, the argument is returned unchanged."

Example

```
CPU_CHAR  c;
CPU_CHAR  c_upper;


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

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 'different' 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:

Micriµm
949 Crestview Circle
Weston, FL 33327-1848
U.S.A.

Phone : +1 954 217 2036
FAX : +1 954 217 2037

WEB : **www.micrium.com**
Email : **licensing@micrium.com**