

Micrium

Empowering Embedded Systems

μ C/CPU

V1.23

User's Manual

www.Micrium.com

Disclaimer

Specifications written in this manual are believed to be accurate, but are not guaranteed to be entirely free of error. Specifications in this manual may be changed for functional or performance improvements without notice. Please make sure your manual is the latest edition. While the information herein is assumed to be accurate, Micrium assumes no responsibility for any errors or omissions and makes no warranties. Micrium specifically disclaims any implied warranty of fitness for a particular purpose.

Copyright notice

You may not extract portions of this manual or modify the PDF file in any way without the prior written permission of Micrium. The software described in this document is furnished under a license and may only be used or copied in accordance with the terms of such a license.

© 2004-2009 Micrium, Weston, Florida 33327-1848, U.S.A.

Trademarks

Names mentioned in this manual may be trademarks of their respective companies. Brand and product names are trademarks or registered trademarks of their respective holders.

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

Micrium

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
V1.23	2009/06/22	SR	Created Manual
V1.23	2009/07/05	ITJ	Updated Manual
V1.23	2009/07/13	ITJ	Released Manual V1.23

Table Of Contents

I	Introduction.....	6
I.1	Portable	6
I.2	Scalable.....	6
I.3	Coding Standards	6
I.4	MISRA C	6
I.5	Safety Critical Certification	7
I.6	μ C/CPU Limitations	7
1	Getting Started with μ C/CPU	8
1.00	Installing μ C/CPU	8
2	μ C/CPU Processor/Compiler Port File(s)	11
2.01	Standard Data Types	11
2.02.01	CPU Word Sizes	11
2.02.02	CPU Word-Memory Order	11
2.03	CPU Stacks	12
2.04	CPU Critical Sections	12
2.04.01	CPU_SR_ALLOC()	13
2.04.02.01	CPU_CRITICAL_ENTER()	14
2.04.02.02	CPU_CRITICAL_EXIT()	15
2.04.03.01	CPU_INT_DIS()	16
2.04.03.02	CPU_INT_EN()	17

3	μC/CPU Library	18
3.00	μC/CPU Configuration	18
3.01	CPU_Init()	19
3.02.01	CPU_NameClr()	20
3.02.02	CPU_NameSet()	21
3.02.03	CPU_NameGet()	22
3.03	CPU Timestamps	23
3.03.01.01	CPU_TS_Get()	24
3.03.01.02	CPU_TS_GetLo()	25
3.03.02	CPU_TS_Update()	26
3.03.03.01	CPU_TS_TmrInit()	27
3.03.03.02	CPU_TS_TmrRd()	28
3.03.04	CPU_TS_to_uSec()	29
3.04	CPU Interrupts Disable Time Measurements	31
3.04.01	CPU_IntDisMeasMaxGet()	31
3.04.02	CPU_IntDisMeasMaxCurGet()	32
3.04.03	CPU_IntDisMeasMaxCurReset()	33
3.05	CPU_CntLeadZeros()	34
A	μC/CPU Licensing Policy	35

Introduction

Designed with **Micrium**'s renowned quality, scalability and reliability, the purpose of **μC/CPU** is to provide a clean, organized ANSI C implementation of each processor's/compiler's hardware-dependent.

I.1 Portable

μC/CPU was designed for the vast variety of embedded applications. The processor-dependent source code for **μC/CPU** is designed to be ported to any processor (CPU) and compiler while **μC/CPU**'s core library source code is designed to be independent of and used with any processor/compiler.

I.2 Scalable

The memory footprint of **μC/CPU** 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/CPU** 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/CPU** 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

μ C/CPU has been designed and implemented with safety critical certification in mind. μ C/CPU 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/CPU 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/CPU is a very robust and highly-reliable software module.

I.6 μ C/CPU Limitations

By design, we have limited some of the feature of μ C/CPU. Table I-1 describes those limitations.

Support for 64-bit data NOT available for all CPUs

Table I-1, μ C/CPU limitations for current software version

Getting Started with μ C/CPU

This chapter provides information on the distribution and installation of μ C/CPU.

1.00 Installing μ C/CPU

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

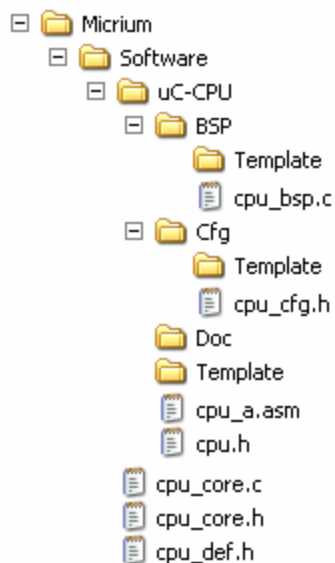


Figure 1-1, μ C/CPU Module Directories and Files

`\uC-CPU` This directory contains CPU-specific code which depends on the processor and compiler used by your application, as well as CPU-independent source files.

The main `µC/CPU` directory contains three master CPU files :

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

`cpu_def.h`

This file declares `#define` constants used to configure processor/compiler-specific CPU word sizes, endianness word order, critical section methods, and other processor configuration.

`cpu_core.c` and `cpu_core.h`

These files contain source code that implements `µC/CPU` features such as host name allocation, timestamps, time measurements, and counting lead zeros. See Chapter 3 for more details.

`\Cfg\Template\cpu_cfg.h`

This template file includes configuration for `µC/CPU` features such as host name allocation, timestamps, time measurements, and assembly optimization. Your application **MUST** include a `cpu_cfg.h` configuration file with application-specific configuration settings.

`\BSP\Template\cpu_bsp.c`

This file includes function templates for the Board-Specific (BSP) code required if certain `µC/CPU` features such as timestamp time measurements and assembly optimization are enabled. Your application **MUST** include code for all BSP functions enabled in `cpu_cfg.h`.

`µC/CPU` directory also contains additional sub-directories specific for each processor/compiler combination organized as follows :

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

`cpu.h`

This file contains `µC/CPU` configuration specific to the processor (`CPU Type`) and compiler (`Compiler`), such as data type definitions, processor address and data word sizes, endianness word order, and critical section macros. See Chapter 2 for more details.

`cpu_a.asm` or `cpu_a.s`

These (optional) files contains assembly code to enable/disable interrupts, implement critical section methods, and any other processor-specific code **NOT** already defined or implemented in the processor's `cpu.h` (or `cpu.c`).

`cpu.c`

This (optional) file contains C and/or assembly code to implement processor-specific code **NOT** already defined or implemented in the processor's `cpu.h` (or `cpu_a.asm`).

`\Template\cpu.h` and `cpu_a.asm`

These template `µC/CPU` configuration files include example configurations for a generic processor/compiler.

An example of ARM-specific CPU processor files is shown in Figure 1-2.

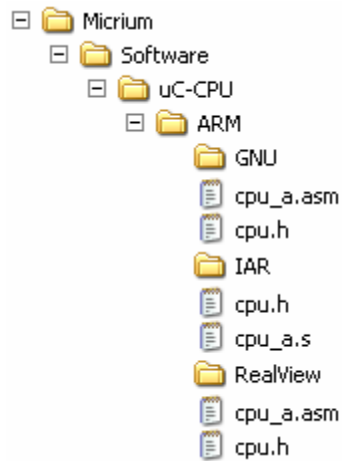


Figure 1-2, μ C/CPU ARM CPU Directories and Files Example

\Application

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

cpu_cfg.h

This application-specific configuration file is required by μ C/CPU to **#define** its configuration constants.

Chapter 2

μC/CPU Processor/Compiler Port File(s)

μC/CPU contains configuration specific to each processor and compiler, such as standard data type definitions, processor address and data word sizes, endianness word order, critical section macros, and possibly other functions and macros. These are defined in each specific processor/compiler subdirectory's `cpu.h`.

2.01 Standard Data Types

μC/CPU ports define standard data types such as `CPU_CHAR`, `CPU_BOOLEAN`, `CPU_INT08U`, `CPU_INT16S`, `CPU_FP32`, etc. These data types are used in Micrium applications, and may be used in your applications, to facilitate portability independent of and between processors/compilers. Most μC/CPU processor/compiler port files minimally support 32-bit data types, but **MAY** optionally support 64-bit (or greater) data types.

In addition, several regularly-used function pointer data types are defined.

2.02.01 CPU Word Sizes

μC/CPU ports include word size configuration such as `CPU_CFG_ADDR_SIZE` and `CPU_CFG_DATA_SIZE`, configured via `CPU_WORD_SIZE_08`, `CPU_WORD_SIZE_16`, and `CPU_WORD_SIZE_32`.

In addition, the following CPU word sizes are also defined based on the configured sizes of `CPU_CFG_ADDR_SIZE` and `CPU_CFG_DATA_SIZE`: `CPU_ADDR`, `CPU_DATA`, `CPU_ALIGN`, and `CPU_SIZE_T`.

2.02.02 CPU Word-Memory Order

μC/CPU ports configure `CPU_CFG_ENDIAN_TYPE` to indicate the processor's word-memory order endianness. `CPU_ENDIAN_TYPE_LITTLE` indicates that a CPU stores/reads data words in memory with the most significant octets at lower memory addresses (and the least significant octets at higher memory addresses) while a `CPU_ENDIAN_TYPE_BIG` CPU stores/reads data words in memory with the most significant octets at higher memory addresses (and the least significant octets at lower memory addresses).

2.03 CPU Stacks

μ C/CPU ports configure `CPU_CFG_STK_GROWTH` to indicate the direction in memory a CPU updates its stack pointers after pushing data onto its stacks. `CPU_STK_GROWTH_HI_TO_LO` indicates that a CPU decrements its stack pointers to the next lower memory address after data is pushed onto a CPU stack while a `CPU_STK_GROWTH_LO_TO_HI` CPU increments its stack pointers to the next higher memory address after data is pushed.

In addition, each μ C/CPU processor port defines a `CPU_STK` data type to the CPU's stack word size.

2.04 CPU Critical Sections

μ C/CPU ports include CPU critical section configuration `CPU_CFG_CRITICAL_METHOD` that indicates how a CPU disables/re-enables interrupts when entering/exiting critical, protected sections :

`CPU_CRITICAL_METHOD_INT_DIS_EN` merely disables/enables interrupts on critical section enter/exit. This is **NOT** a preferred method since it does **NOT** support multiple levels of interrupts. However, with some processors/compilers, this is the only available method.

`CPU_CRITICAL_METHOD_STATUS_STK` pushes/pops interrupt status onto stack before disabling/re-enabling interrupts. This is one preferred method since it supports multiple levels of interrupts. However, this method assumes that the compiler provides C-level &/or assembly-level functionality for pushing/saving the interrupt status onto a local stack, disabling interrupts, and popping/restoring the interrupt status from the local stack.

`CPU_CRITICAL_METHOD_STATUS_LOCAL` saves/restores interrupt status to a local variable before disabling/re-enabling interrupts. This also is a preferred method since it supports multiple levels of interrupts. However, this method assumes that the compiler provides C-level &/or assembly-level functionality for saving the interrupt status to a local variable, disabling interrupts, and restoring the interrupt status from the local variable.

Each μ C/CPU processor port implements critical section macros with calls to interrupt disable/enable macros. Applications should **ONLY** use the critical section macros (Section 2.04.02) since interrupt disable/enable macros (Section 2.04.03) are only intended for use by core μ C/CPU functions.

Each μ C/CPU processor port may define its interrupt disable/enable macros with inline-assembly directly in `cpu.h`, or calls to C functions defined in `cpu.c`, or calls to assembly subroutines defined in `cpu_a.asm` (or `cpu_a.s`). The specific implementation **SHOULD** be based on the processor port's configured CPU critical section method (see Section 2.04.03).

In addition, each μ C/CPU processor port defines an appropriately-sized `CPU_SR` data type large enough to completely store the processor's/compiler's status word. `CPU_CRITICAL_METHOD_STATUS_LOCAL` method requires each function that calls critical section macros or interrupt disable/enable macros to declare local variable `cpu_sr` of type `CPU_SR`, which **SHOULD** be declared via the `CPU_SR_ALLOC()` macro (see Section 2.04.01).

2.04.01 CPU_SR_ALLOC()

Allocates CPU status register word as local variable `cpu_sr`, when necessary, for use with critical section macros.

Prototype

```
CPU_SR_ALLOC( );
```

Arguments

None.

Returned Value

None.

Notes / Warnings

- 1) `CPU_SR_ALLOC()` **MUST** be called immediately after the last local variable declaration in a function.

Example

```
CPU_BOOLEAN  ts_init;  
CPU_TS       ts_cur;  
CPU_SR_ALLOC();           /* Declared immediately after all local variables ... */  
  
                        /* ... but before any code statements.          */  
ts_init = DEF_YES;  
ts_cur  = CPU_TS_TmrRd();
```

2.04.02.01 CPU_CRITICAL_ENTER()

Enters critical sections, disabling interrupts.

Prototype

```
CPU_CRITICAL_ENTER();
```

Arguments

None.

Returned Value

None.

Notes / Warnings

- 1) `CPU_CRITICAL_ENTER()`/`CPU_CRITICAL_EXIT()` **SHOULD** be used to protect critical sections of code from interrupted or concurrent access when no other protection mechanisms are available or appropriate. For example, system code that must be re-entrant but without use of a software lock should protect the code using CPU critical sections.
- 2) Since interrupts are disabled upon calling `CPU_CRITICAL_ENTER()` and are not re-enabled until after calling `CPU_CRITICAL_EXIT()`, interrupt and operating system context switching are postponed while all critical sections have not completely exited.
- 3) Critical sections can be nested any number of times as long as `CPU_CFG_CRITICAL_METHOD` is **NOT** configured as `CPU_CRITICAL_METHOD_INT_DIS_EN`, which would re-enable interrupts upon the first call to `CPU_CRITICAL_EXIT()`, not the last call.
- 4) `CPU_CRITICAL_ENTER()` **SHOULD/MUST ALWAYS** call `CPU_CRITICAL_EXIT()` once critical section protection is no longer needed.

Example

```
CPU_SR_ALLOC();

CPU_CRITICAL_ENTER();
:
:           /* Code protected by critical sections ... */
:           /* ... from interrupts or concurrent access. */
:
CPU_CRITICAL_EXIT();
```

2.04.02.02 CPU_CRITICAL_EXIT()

Exits critical sections, restoring previous interrupt status and/or enabling interrupts.

Prototype

```
CPU_CRITICAL_EXIT();
```

Arguments

None.

Returned Value

None.

Notes / Warnings

- 1) **CPU_CRITICAL_ENTER() / CPU_CRITICAL_EXIT()** **SHOULD** be used to protect critical sections of code from interrupted or concurrent access when no other protection mechanisms are available or appropriate. For example, system code that must be re-entrant but without use of a software lock should protect the code using CPU critical sections.
- 2) Since interrupts are disabled upon calling **CPU_CRITICAL_ENTER()** and are not re-enabled until after calling **CPU_CRITICAL_EXIT()**, interrupt and operating system context switching are postponed while all critical sections have not completely exited.
- 3) Critical sections can be nested any number of times as long as **CPU_CFG_CRITICAL_METHOD** is **NOT** configured as **CPU_CRITICAL_METHOD_INT_DIS_EN**, which would re-enable interrupts upon the first call to **CPU_CRITICAL_EXIT()**, not the last call.
- 4) **CPU_CRITICAL_EXIT()** **MUST ALWAYS** call **CPU_CRITICAL_ENTER()** at the start of critical section protection.

Example

```
CPU_SR_ALLOC();

CPU_CRITICAL_ENTER();
:
:           /* Code protected by critical sections ... */
:           /* ... from interrupts or concurrent access. */
:
CPU_CRITICAL_EXIT();
```

2.04.03.01 CPU_INT_DIS()

Saves current interrupt status, if processor/compiler capable, & then disables interrupts.

Prototype

```
CPU_INT_DIS();
```

Arguments

None.

Returned Value

None.

Notes / Warnings

- 1) **CPU_INT_DIS()** **SHOULD** be defined based on the processor port's configured CPU critical section method, **CPU_CFG_CRITICAL_METHOD**; and may be defined with inline-assembly directly in **cpu.h**, or with calls to C functions defined in **cpu.c**, or calls to assembly subroutines defined in **cpu_a.asm** (or **cpu_a.s**). See also Section 2.04.

Example Templates

The following example templates assume corresponding functions are defined in either **cpu.c** or **cpu_a.asm**:

```
#if (CPU_CFG_CRITICAL_METHOD == CPU_CRITICAL_METHOD_INT_DIS_EN)
#define CPU_INT_DIS() { CPU_IntDis(); }           /* Disable interrupts. */
#endif

#if (CPU_CFG_CRITICAL_METHOD == CPU_CRITICAL_METHOD_STATUS_STK)
#define CPU_INT_DIS() { CPU_SR_Push(); }          /* Push CPU status & disable interrupts. */
#endif

#if (CPU_CFG_CRITICAL_METHOD == CPU_CRITICAL_METHOD_STATUS_LOCAL)
#define CPU_INT_DIS() { cpu_sr = CPU_SR_Save(); } /* Save CPU status & disable interrupts. */
#endif
```

2.04.03.02 CPU_INT_EN()

Restores previous interrupt status and/or enables interrupts.

Prototype

`CPU_INT_EN();`

Arguments

None.

Returned Value

None.

Notes / Warnings

- 1) `CPU_INT_DIS()` **SHOULD** be defined based on the processor port's configured CPU critical section method, `CPU_CFG_CRITICAL_METHOD`; and may be defined with inline-assembly directly in `cpu.h`, or with calls to C functions defined in `cpu.c`, or calls to assembly subroutines defined in `cpu_a.asm` (or `cpu_a.s`). See also Section 2.04.

Example

The following example templates assume corresponding functions are defined in either `cpu.c` or `cpu_a.asm`:

```
#if (CPU_CFG_CRITICAL_METHOD == CPU_CRITICAL_METHOD_INT_DIS_EN)
#define CPU_INT_EN() { CPU_IntEn(); } /* Enable interrupts. */
#endif

#if (CPU_CFG_CRITICAL_METHOD == CPU_CRITICAL_METHOD_STATUS_STK)
#define CPU_INT_EN() { CPU_SR_Pop(); } /* Pop CPU status. */
#endif

#if (CPU_CFG_CRITICAL_METHOD == CPU_CRITICAL_METHOD_STATUS_LOCAL)
#define CPU_INT_EN() { CPU_SR_Restore(cpu_sr); } /* Restore CPU status. */
#endif
```

μC/CPU Library

μC/CPU contains library features such as host name allocation, timestamps, time measurements, counting lead zeros, etc. These functions are defined in `cpu_core.c`.

3.00 μC/CPU Configuration

The following μC/CPU configurations may be optionally configured in `cpu_cfg.h` :

CPU_CFG_NAME_EN	Includes code to set & get a configured CPU host name (see Section 3.02). This feature may be configured to either DEF_DISABLED or DEF_ENABLED .
CPU_CFG_NAME_SIZE	Configures the maximum CPU name size (in number of ASCII characters, including the terminating NULL character).
CPU_CFG_TS_EN	Includes CPU timestamp functionality (see Section 3.03). This feature may be configured to either DEF_DISABLED or DEF_ENABLED .
CPU_CFG_INT_DIS_MEAS_EN	Includes code to measure & return maximum interrupts disabled time (see Section 3.04). This feature is enabled if the macro is #define 'd in <code>cpu_cfg.h</code> .
CPU_CFG_INT_DIS_MEAS_OVRHD_NBR	Configures the number of times to measure & calculate the interrupts disabled time measurement overhead.
CPU_CFG_LEAD_ZEROS_ASM_PRESENT	Implements counting lead zeros functionality in assembly (see Section 3.05). This feature is enabled if the macro is #define 'd in <code>cpu_cfg.h</code> (or <code>cpu.h</code>).

3.01 CPU_Init()

Initializes the core CPU module.

Prototype

```
void CPU_Init (void);
```

Arguments

None.

Returned Value

None.

Notes / Warnings

- 1) **MUST** be called prior to calling any other core CPU functions :
 - a) CPU host name
 - b) CPU timestamps
 - c) CPU interrupts disabled time measurements

3.02.01 CPU_NameClr()

Clears the CPU host name.

Prototype

```
void CPU_NameClr (void);
```

Arguments

None.

Returned Value

None.

Notes / Warnings

- 1) This function enabled **ONLY** if **CPU_CFG_NAME_EN** is **DEF_ENABLED** in **cpu_cfg.h** (see Section 3.00).

Example

```
CPU_NameClr(); /* Clear CPU host name. */
```

3.02.02 CPU_NameSet()

Sets the CPU host name.

Prototype

```
void CPU_NameSet (CPU_CHAR *p_name,  
                  CPU_ERR *p_err);
```

Arguments

`p_name` Pointer to CPU host name to set (see Note #2).

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

<code>CPU_ERR_NONE</code>	CPU host name successfully set.
<code>CPU_ERR_NULL_PTR</code>	Argument <code>p_name</code> passed a <code>NULL</code> pointer.
<code>CPU_ERR_NAME_SIZE</code>	Invalid CPU host name size.

Returned Value

None.

Notes / Warnings

- 1) This function enabled **ONLY** if `CPU_CFG_NAME_EN` is `DEF_ENABLED` in `cpu_cfg.h` (see Section 3.00).
- 2) `p_name` ASCII string size, including the terminating `NULL` character, **MUST** be less than or equal to `CPU_CFG_NAME_SIZE`.

Example

```
CPU_CHAR *p_name;  
CPU_ERR err;  
  
p_name = "ARM Target";  
  
CPU_NameSet(p_name, &err); /* Set CPU host name. */  
  
if (err != CPU_ERR_NONE) {  
    printf("COULD NOT SET CPU HOST NAME.");  
}
```

3.02.03 CPU_NameGet()

Gets the CPU host name.

Prototype

```
void CPU_NameGet (CPU_CHAR *p_name,  
                  CPU_ERR *p_err);
```

Arguments

p_name Pointer to an ASCII character array that will receive the return CPU host name ASCII string from this function (see Note #2).

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

CPU_ERR_NONE
CPU_ERR_NULL_PTR

CPU host name successfully returned.
Argument **p_name** passed a **NULL** pointer.

Returned Value

None.

Notes / Warnings

- 1) This function enabled **ONLY** if **CPU_CFG_NAME_EN** is **DEF_ENABLED** in **cpu_cfg.h** (see Section 3.00).
- 2) The size of the ASCII character array that will receive the return CPU host name ASCII string :
 - a) **MUST** be greater than or equal to the current CPU host name's ASCII string size including the terminating **NULL** character;
 - b) **SHOULD** be greater than or equal to **CPU_CFG_NAME_SIZE**.

Example

```
CPU_CHAR *p_name;  
CPU_ERR err;  
  
CPU_NameGet(p_name, &err); /* Get CPU host name. */  
  
if (err == CPU_ERR_NONE) {  
    printf("CPU Host Name = %s", p_name);  
} else {  
    printf("COULD NOT GET CPU HOST NAME.");  
}
```

3.03 CPU Timestamps

CPU timestamps emulate a real-time 64-bit timer by accumulating timer counts via `CPU_TS_Update()` which must be called periodically by an application-/developer-defined function (see Section 3.03.02). An application can then get CPU timestamps and use either as raw timer counts or converted to microseconds (see Section 3.03.04).

Note that if either the CPU timestamp feature **OR** the interrupts disable time measurement feature is enabled (see Section 3.00), then the application/developer **MUST** provide CPU timestamp timer functions (see Sections 3.03.03).

3.03.01.01 CPU_TS_Get()

Gets current CPU timestamp.

Prototype

```
void CPU_TS_Get (CPU_TS *p_ts_lo,  
                 CPU_TS *p_ts_hi);
```

Arguments

p_ts_lo	Pointer to timestamp variable that will receive the current CPU timestamp's lower half (in timestamp timer counts), if available.
p_ts_hi	Pointer to timestamp variable that will receive the current CPU timestamp's higher half (in timestamp timer counts), if available.

Returned Value

None.

Notes / Warnings

- 1) This function enabled **ONLY** if **CPU_CFG_TS_EN** is **DEF_ENABLED** in **cpu_cfg.h** (see Section 3.00).
- 2) The amount of time measured by CPU timestamps is calculated by either of the following equations :
 - a) **Time measured = Number timer counts * Timer period**

where

Number timer counts	Number of timer counts measured
Timer period	Timer's period in some units of (fractional) seconds
Time measured	Amount of time measured, in same units of (fractional) seconds as the Timer period

$$\text{b) Time measured} = \frac{\text{Number timer counts}}{\text{Timer frequency}}$$

where

Number timer counts	Number of timer counts measured
Timer frequency	Timer's frequency in some units of counts per second
Time measured	Amount of time measured, in seconds

Example

```
CPU_TS ts_lo;  
CPU_TS ts_hi;  
  
CPU_TS_Get(&ts_lo, &ts_hi); /* Get current CPU timestamp. */
```

3.03.01.02 CPU_TS_GetLo()

Gets current CPU timestamp, lower-half only.

Prototype

```
CPU_TS CPU_TS_GetLo (void);
```

Arguments

None.

Returned Value

Current CPU timestamp's lower half (in timestamp timer counts).

Notes / Warnings

- 1) This function enabled **ONLY** if **CPU_CFG_TS_EN** is **DEF_ENABLED** in **cpu_cfg.h** (see Section 3.00).
- 2) The amount of time measured by CPU timestamps is calculated by either of the following equations :

a)
$$\text{Time measured} = \text{Number timer counts} * \text{Timer period}$$

where

Number timer counts	Number of timer counts measured
Timer period	Timer's period in some units of (fractional) seconds
Time measured	Amount of time measured, in same units of (fractional) seconds as the Timer period

b)
$$\text{Time measured} = \frac{\text{Number timer counts}}{\text{Timer frequency}}$$

where

Number timer counts	Number of timer counts measured
Timer frequency	Timer's frequency in some units of counts per second
Time measured	Amount of time measured, in seconds

Example

```
CPU_TS ts_lo;

ts_lo = CPU_TS_GetLo(); /* Get current CPU timestamp. */
```

3.03.02 CPU_TS_Update()

Updates current CPU timestamp.

Prototype

```
void CPU_TS_Update (void);
```

Arguments

None.

Returned Value

None.

Notes / Warnings

- 1) This function enabled **ONLY** if **CPU_CFG_TS_EN** is **DEF_ENABLED** in **cpu_cfg.h** (see Section 3.00).
- 2)
 - a) CPU timestamp **MUST** be updated periodically by some application (or BSP) time handler in order to (adequately) maintain the CPU timestamp time.
 - b) CPU timestamp **MUST** be updated more frequently than timestamp timer overflows; otherwise, CPU timestamp will lose time.

Example

```
void AppPeriodicTimeHandler (void)
{
    :
    CPU_TS_Update(); /* Update current CPU timestamp (see Note #2). */
    :
}
```

3.03.03.01 CPU_TS_TmrInit()

Application-defined function to initialize and start the CPU timestamp's (hardware or software) timer.

Prototype

```
CPU_INT16U CPU_TS_TmrInit (void);
```

Arguments

None.

Returned Value

Number of left-shifts to scale & return (hardware or software) timer as (32-bit) **CPU_TS** data type (see Note #2a1), if necessary.

0 (see Note #2a2), otherwise.

Notes / Warnings

- 1) **CPU_TS_TmrInit()** is an application/BSP function that **MUST** be defined by the developer if either of the following CPU features is enabled (see Section 3.00) :
 - a) CPU timestamps
 - b) CPU interrupts disabled time measurements
- 2)
 - a) Timer count values **MUST** be scaled & returned via (32-bit) **CPU_TS** data type.
 - 1) If timer has less bits, left-shift timer values until the most significant bit of the timer value is shifted into the most significant bit of the return timestamp data type.
 - 2) If timer has more bits, truncate timer values' higher-order bits greater than the return timestamp data type.
 - b) Timer **SHOULD** be an 'up' counter whose values increase with each time count.
 - c) When applicable, timer period **SHOULD** be less than the typical measured time but **MUST** be less than the maximum measured time; otherwise, timer resolution inadequate to measure desired times.

Example Template

```
CPU_INT16U CPU_TS_TmrInit (void)
{
    CPU_INT16U  nbr_left_shifts;

    :
    nbr_left_shifts = /* Insert code to initialize/start CPU timestamp timer (see Note #2). */ ;
    :

    return (nbr_left_shifts);
}
```

3.03.03.02 CPU_TS_TmrRd()

Application-defined function to get current CPU timestamp timer count.

Prototype

```
CPU_TS CPU_TS_TmrRd (void);
```

Arguments

None.

Returned Value

(32-bit) CPU timestamp timer count value (see Notes #2a & #2b).

Notes / Warnings

- 1) `CPU_TS_TmrRd()` is an application/BSP function that **MUST** be defined by the developer if either of the following CPU features is enabled (see Section 3.00) :
 - a) CPU timestamps
 - b) CPU interrupts disabled time measurements
- 2)
 - a) Timer count values **MUST** be scaled & returned via (32-bit) `CPU_TS` data type.
 - 1) If timer has less bits, left-shift timer values until the most significant bit of the timer value is shifted into the most significant bit of the return timestamp data type.
 - 2) If timer has more bits, truncate timer values' higher-order bits greater than the return timestamp data type.
 - b) Timer **SHOULD** be an 'up' counter whose values increase with each time count.
 - c) When applicable, timer period **SHOULD** be less than the typical measured time but **MUST** be less than the maximum measured time; otherwise, timer resolution inadequate to measure desired times.

Example Template

```
CPU_TS CPU_TS_TmrRd (void)
{
    CPU_TS ts_tmr_cnts;

    :
    ts_tmr_cnts = /* Insert code to get/return CPU timestamp timer counts (see Note #2). */ ;
    :

    return (ts_tmr_cnts);
}
```

3.03.04 CPU_TS_to_uSec()

Application-defined function to convert a CPU timestamp from timer counts to microseconds.

Prototype

```
void CPU_TS_to_uSec (CPU_TS      ts_lo_cnts,  
                    CPU_TS      ts_hi_cnts,  
                    CPU_INT32U  *p_ts_lo_usec,  
                    CPU_INT32U  *p_ts_hi_usec);
```

Arguments

ts_lo_cnts	CPU timestamp lower half (in timestamp timer counts [see Note #2aA]).
ts_hi_cnts	CPU timestamp upper half (in timestamp timer counts [see Note #2aA]).
p_ts_lo_usec	Pointer to variable that will receive the converted CPU timestamp's lower half [in microseconds (see Note #2aD)], if available.
p_ts_hi_usec	Pointer to variable that will receive the converted CPU timestamp's upper half [in microseconds (see Note #2aD)], if available.

Returned Value

None.

Notes / Warnings

- 1) **CPU_TS_to_uSec()** is an application/BSP function that **MAY** be optionally defined by the developer when either of the following CPU features is enabled (see Section 3.00) :
 - a) CPU timestamps
 - b) CPU interrupts disabled time measurements
- 2) a) The amount of time measured by CPU timestamps is calculated by either of the following equations :

$$\begin{aligned} 1) \text{ Time measured} &= \text{Number timer counts} * \frac{10^6 \text{ microseconds}}{1 \text{ second}} * \text{Timer period} \\ 2) \text{ Time measured} &= \frac{\text{Number timer counts}}{\text{Timer frequency}} * \frac{10^6 \text{ microseconds}}{1 \text{ second}} \end{aligned}$$

where

- | | |
|------------------------|------------------------------------------------------|
| A) Number timer counts | Number of timer counts measured |
| B) Timer frequency | Timer's frequency in some units of counts per second |
| C) Timer period | Timer's period in some units of (fractional) seconds |
| D) Time measured | Amount of time measured, in microseconds |

- b) Timer period **SHOULD** be less than the typical measured time but **MUST** be less than the maximum measured time; otherwise, timer resolution inadequate to measure desired times.
- c) Specific implementations may convert any number of **CPU_TS** bits, up to 64, into microseconds.

Example Template

```
void CPU_TS_to_uSec (CPU_TS      ts_lo_cnts,
                    CPU_TS      ts_hi_cnts,
                    CPU_INT32U *p_ts_lo_usec,
                    CPU_INT32U *p_ts_hi_usec)
{
    CPU_INT32U  ts_lo_usec;
    CPU_INT32U  ts_hi_usec;

    :
    :           /* Insert code to convert CPU timestamp into microseconds (see Note #2). */
    :

    if (p_ts_lo_usec != (CPU_INT32U *)0) {
        *p_ts_lo_usec = (CPU_INT32U )ts_lo_usec;
    }
    if (p_ts_hi_usec != (CPU_INT32U *)0) {
        *p_ts_hi_usec = (CPU_INT32U )ts_hi_usec;
    }
}
```

3.04 CPU Interrupts Disable Time Measurements

When enabled, the maximum amount of time interrupts are disabled during calls to `CPU_CRITICAL_ENTER()`/`CPU_CRITICAL_EXIT()` is measured and saved. There are two maximum interrupts disable time measurements, one resetable and the other non-resetable, both measured in units of CPU timestamps (see Section 3.03).

Note that the interrupts disable time measurement feature requires that the application/developer provide CPU timestamp timer functions (see Sections 3.03.03).

3.04.01 CPU_IntDisMeasMaxGet()

Gets (non-resetable) maximum interrupts disabled time.

Prototype

```
CPU_TS CPU_IntDisMeasMaxGet (void);
```

Arguments

None.

Returned Value

(Non-resetable) maximum interrupts disabled time (in CPU timestamp timer counts).

Notes / Warnings

- 1) This function enabled **ONLY** if `CPU_CFG_INT_DIS_MEAS_EN` is `#define`'d in `cpu_cfg.h` (see Section 3.00).

Example

```
CPU_TS time_max_cnts;

time_max_cnts = CPU_IntDisMeasMaxGet(); /* Get maximum interrupts disabled time. */
```

3.04.02 CPU_IntDisMeasMaxCurGet()

Gets current/resetable maximum interrupts disabled time.

Prototype

```
CPU_TS CPU_IntDisMeasMaxCurGet (void);
```

Arguments

None.

Returned Value

Current maximum interrupts disabled time (in CPU timestamp timer counts).

Notes / Warnings

- 1) This function enabled **ONLY** if **CPU_CFG_INT_DIS_MEAS_EN** is **#define**'d in **cpu_cfg.h** (see Section 3.00).

Example

```
CPU_TS time_max_cnts;  
  
time_max_cnts = CPU_IntDisMeasMaxCurGet(); /* Get current maximum interrupts disabled time. */
```

3.04.03 CPU_IntDisMeasMaxCurReset()

Resets current maximum interrupts disabled time.

Prototype

```
CPU_TS CPU_IntDisMeasMaxCurReset (void);
```

Arguments

None.

Returned Value

Maximum interrupts disabled time (in CPU timestamp timer counts) before resetting.

Notes / Warnings

- 1) This function enabled **ONLY** if **CPU_CFG_INT_DIS_MEAS_EN** is **#define**'d in **cpu_cfg.h** (see Section 3.00).

Example

```
CPU_TS time_max_cnts;  
  
time_max_cnts = CPU_IntDisMeasMaxCurReset(); /* Reset current maximum interrupts disabled time. */
```

3.05 CPU_CntLeadZeros()

Counts the number of contiguous, most-significant, leading zero bits in a data value.

Prototype

```
CPU_DATA CPU_CntLeadZeros (CPU_DATA val);
```

Arguments

val Data value to count leading zero bits.

Returned Value

None.

Notes / Warnings

- 1) This function implemented in `cpu_core.c` if `CPU_CFG_LEAD_ZEROS_ASM_PRESENT` is **NOT** `#define`'d in `cpu_cfg.h`, and **SHOULD** be implemented in `cpu_a.asm` (or `cpu_a.s`) if `CPU_CFG_LEAD_ZEROS_ASM_PRESENT` is `#define`'d in `cpu_cfg.h` (see Section 3.00).

Example

```
CPU_DATA val;  
CPU_DATA nbr_lead_zeros;  
  
val = 0x0643A718;  
nbr_lead_zeros = CPU_CntLeadZeros(val);
```

Appendix A

μC/CPU Licensing Policy

You need to obtain an 'Object Code Distribution License' to embed μC/CPU 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:

Micrium

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