

Acknowledgement and About

Acknowledgments

Thanks go to:

- Dimitri van Heesch, author of Doxygen.
- Adam Dunkels, author of protothreads library and Contiki OS kernel.
- NiuYuQing for logo and suggestions.
- My family for supporting and encourage.
- Haroopad, a well-done markdown tool.
- Enya for great voice to listen to while coding.

About

What is this manual for

This manual is used as a reference to RainbowBS(RBS) which is an open-source base library developed for embedded systems especially with limit resources.

How to get source package

RBS has been committed to GitHub.So you can get the newest source package from https://github.com/jacobqwq/RainbowBS/

Future work

This v0.1.0 is the first distributed RBS version.

Although RBS can be used independently, it will also become a basic part of **RainbowSys** according to the plan.

The goal of RainbowSys is to build an open-source flexible and configurable system software written by C language for embedded systems ,especially like low cost micro-controllers which have limit resources.RainbowSys will consist of CMSIS-compatible RTOS,file system, GUI,comunication stack,etc....

So RBS is the first distributed component of RainbowSys and will offer common support for the other components of RainbowSys.

The next distributed component of RainbowSys will be the GUI component named RainbowGX(RGX), which is now being improved continuously together with RBS.And you will see RGX soon.:)



RGX Desk Level Demonstration

Author

Designed by **QWQ** coming from Qingdao, Shandong Prov, China.



Email: jacobqwq@icloud.com

Generated by



Chapter1 Introduction

Table of Contents

- +RainbowBS Features
- ↓ Requirements
- ↓ Licence
- Ψ Conventions

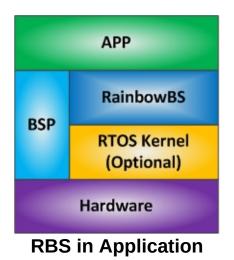
What is **RainbowBS**

RainbowBS(RBS) is an open-source base library developed for embedded systems especially with limit resources.

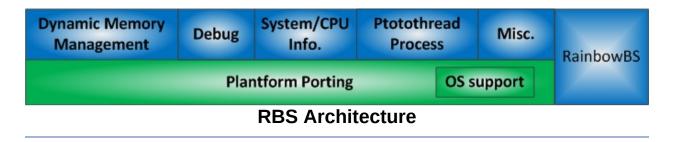
RBS is written by C language in compliance with the ISO/IEC 9899:1999 standard and it is set-up to be highly portable. Therefore it is independent of hardware architecture and can be incorporated into low cost micro-controllers like 8051, AVR, PIC, ARM and etc..., just with small porting code.

This document shows you how to configure and use RBS.

Following diagram shows the RBS position in the entire software architecture:



Following diagram shows the RBS architecture:



RainbowBS Features

• Hardware Features

- Any 8/16/32 bits CPU can use it.
- One timer and very little flash and RAM needed.
- Software Features
 - **Portable**:RBS is written by C language,so it is portable on any C compiler compatible with the C99 standard.
 - **Configurable**:RBS has base components and optional advanced components listed below.

Basic Type	Basic Macros	System/CPU Info.	Memory Operation	Conversion related	
base	base	base	base	base	
Debug	Dynamic Memory Management		Protothread Process Mod		
optional	optional		optional	optional	

- **OS Independence**:RBS can run with or without OS support.It offers thread-safety feature for OS.
- **PC Tools**: RAutoConfig for generating configurable file by UI.

Requirements

- Only CPU,timer,RAM and flash are necessary in the target system.
 The C compiler compatible with the ISO/IEC 9899:1999
- standard(C99).

Licence

RainbowBS is an open-source base library for embedded systems. This is a free software and opened for education, research and commercial developments under license policy of following terms.

Copyright © 2015-2016 QWQ(jacobqwq@icloud.com). All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- The name of the author may not be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL COPYRIGHT HOLDERS AND CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Conventions

Following conventions are used in this document

- text used for normal text
- *item* used for item
- emphasize bold font used to emphasize
- printf() used for code quotes

Generated by doxyden 1.8.9.1



Chapter2 Basic Components

Basic components consist of type,macros,system/CPU information,memory operation and digit conversion.

Table of Contents

 How to port basic components

Type defination refers to basic types

Macros refers to basic macros

System related refers to system releted

CPU related refers to CPU information

Memory operation related refers to memory operation

Digit conversion refers to conversion releted

How to port basic components

System/CPU information need some macros defined in file RainbowBSConf.h.

RBS_CFG_SYS_INFO indicates a string which RBS_GetSysInfo() returns.

RBS_CFG_TICK_RATE indicates the timer tick per second.

RBS_CFG_CPU_WORD_SIZE indicates size which RBS_GetCPUBits() returns.

RBS_CFG_CPU_BYTE_ORDER_L is defined to 0 or 1 which **RBS_IsCPULittleEndian()** uses.

RBS_CFG_CPU_STACK_DOWN is defined to 0 or 1 which **RBS_IsStackGrowDown()** uses.

Generated by doxyden 1.8.9.1



Chapter3 Application Model

Table of Contents

- Application Model Introduction
- *Application Model Supported by RBS
- - $^{\downarrow}$ Scheduling
 - **Unplementation**

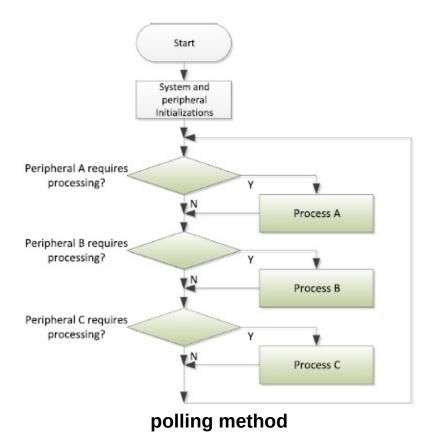
Application Model Introduction

From the view of application developers for an embedded system, there are mainly two ways to construct program for an application.

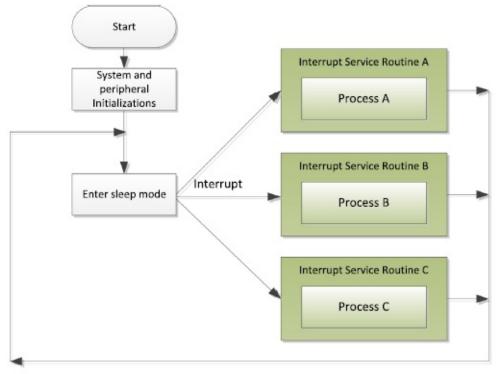
• Event-Driven application model

This model is very suitable for simple applications. Two methods are used, polling or interrupt driven.

For polling method, the processor can wait for data ready to process it, and wait again. The polling method works well for very simple applications, but it has several disadvantages. For example, when the application gets more complex, the polling loop design might get very difficult to maintain. Also, it is difficult to define priorities between different services using polling and you might end up with poor responsiveness, where a peripheral requesting service might need to wait a long time while the processor is handling less important tasks. And lots of energy is wasted during the polling when service is not required.

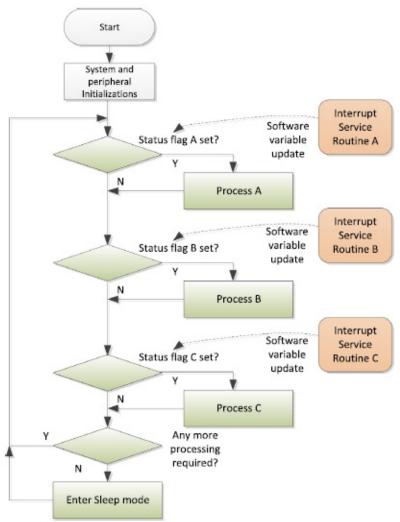


For interrupt driven method, almost all micro-controllers have some sort of sleep mode support to reduce power, in which the peripheral can wake up the processor when it requires a service. Interrupts from different peripherals can be assigned with different interrupt priority levels so that this allows much better responsiveness..



interrupt driven method

In some cases, the processing of data from peripheral services can be partitioned into two parts: the first part needs to be done quickly, and the second part can be carried out a little bit later. In such situations we can use a mixture of interrupt-driven and polling methods to construct the program. When a peripheral requires service, it triggers an interrupt request as in an interrupt-driven application. Once the first part of the interrupt service is carried out, it updates some software variables so that the second part of the service can be executed in the polling-based application code.



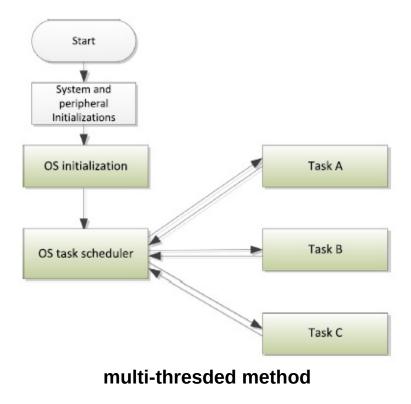
mixed polling and interrupt driven method

Using this arrangement, we can reduce the duration of high-priority interrupt handlers so that lower priority interrupt services can get served quicker. At the same time, the processor can still enter sleep mode to save power when no servicing is needed.

• Multi-Threaded application model

When the applications get more complex, a polling and interruptdriven program structure might not be able to handle the processing requirements. For example, some tasks that can take a long time to execute might need to be processed concurrently. This can be done by dividing the processor's time into a number of time slots and allocating the time slots to these tasks. While it is technically possible to create such an arrangement by manually partitioning the tasks and building a simple scheduler to handle this, it is often impractical to do this in real projects as it is time consuming and can make the program much harder to maintain and debug.

In these applications, a Real-Time Operating System(RTOS) can be used to handle the task scheduling (Figure 2.11). An RTOS allows multiple processes to be executed concurrently, by dividing the processor's time into time slots and allocating the time slots to the processes that require services. A timer is need to handle the timekeeping for the RTOS, and at the end of each time slot, the timer generates a timer interrupt, which triggers the task scheduler and decides if context switching should be carried out. If yes, the current executing process is suspended and the processor executes another process.



Application Model Supported by RBS

There are three optional application model to write your application code based on RBS.

Normal Model

This is an event-driven system model so that an endless loop is needed in main().

OS Model

This is a multi-threaded system model so that an OS kernel is needed.

• Prptothread Process Model

This is an event-driven system with multi-threaded liked blocking property.

Protothreads are a extremely lightweight, stackless type of threads that provides a blocking context on top of an event-driven system, without the overhead of per-thread stacks. The purpose of protothreads is to implement sequential flow of control without complex state machines or full multi-threading. Protothreads provides conditional blocking inside C functions.

The advantage of protothreads over a purely event-driven approach is that protothreads provides a sequential code structure that allows for blocking functions. In purely event-driven systems, blocking must be implemented by manually breaking the function into two pieces one for the piece of code before the blocking call and one for the code after the blocking call. This makes it hard to use control structures such as if() conditionals and while() loops.

The advantage of protothreads over ordinary threads is that a protothread do not require a separate stack. In memory constrained

systems, the overhead of allocating multiple stacks can consume large amounts of the available memory. In contrast, each protothread only requires between two and twelve bytes of state, depending on the architecture.

Protothread Process

The protothreads library was written by Adam Dunkels adam@sics.se with support from Oliver Schmidt ol.sc@web.de.

Protothreads are a extremely lightweight, stackless threads that provides a blocking context on top of an event-driven system, without the overhead of per-thread stacks. The purpose of protothreads is to implement sequential flow of control without using complex state machines or full multi-threading. Protothreads provides conditional blocking inside a C function.

In memory constrained systems, such as deeply embedded systems, traditional multi-threading may have a too large memory overhead. In traditional multi-threading, each thread requires its own stack, that typically is over-provisioned. The stacks may use large parts of the available memory.

The main advantage of protothreads over ordinary threads is that protothreads are very lightweight: a protothread does not require its own stack. Rather, all protothreads run on the same stack and context switching is done by stack rewinding. This is advantageous in memory constrained systems, where a stack for a thread might use a large part of the available memory. A protothread only requires only two bytes of memory per protothread. Moreover, protothreads are implemented in pure C and do not require any machine-specific assembler code.

A protothread runs within a single C function and cannot span over other functions. A protothread may call normal C functions, but cannot block inside a called function. Blocking inside nested function calls is instead made by spawning a separate protothread for each potentially blocking function. The advantage of this approach is that blocking is explicit: the programmer knows exactly which functions that block that which functions the never blocks.

Protothreads are similar to asymmetric co-routines. The main difference is that co-routines uses a separate stack for each co-routine, whereas protothreads are stackless. The most similar mechanism to protothreads are Python generators. These are also stackless constructs, but have a different purpose. Protothreads provides blocking contexts inside a C function, whereas Python generators provide multiple exit points from a generator function.

Detail referances to protothread process protothread process

Local variables

Note

Because protothreads do not save the stack context across a blocking call, local variables are not preserved when the protothread blocks. This means that local variables should be used with utmost care - if in doubt, do not use local variables inside a protothread!

Scheduling

A protothread is driven by repeated calls to the function in which the protothread is running. Each time the function is called, the protothread will run until it blocks or exits. Thus the scheduling of protothreads is done by the application that uses protothreads.

Implementation

Protothreads are implemented using **local continuations**. A local continuation represents the current state of execution at a particular place in the program, but does not provide any call history or local variables. A local continuation can be set in a specific function to capture the state of the function. After a local continuation has been set can be resumed in order to restore the state of the function at the point where the local continuation was set.

Local continuations can be implemented in a variety of ways:

- by using machine specific assembler code.
- by using standard C constructs.
- by using compiler extensions.

The first way works by saving and restoring the processor state, except for stack pointers, and requires between 16 and 32 bytes of memory per protothread. The exact amount of memory required depends on the architecture.

The standard C implementation requires only two bytes of state per protothread and utilizes the C switch() statement in a non-obvious way that is similar to Duff's device. This implementation does, however, impose a slight restriction to the code that uses protothreads in that the code cannot use switch() statements itself.

Certain compilers has C extensions that can be used to implement protothreads. GCC supports label pointers that can be used for this purpose. With this implementation, protothreads require 4 bytes of RAM per protothread.

Generated by doxygen 1.8.9.1



Chapter4 Dynamic Memory Managment

Dynamic memory management is an optional component for memory usage. It offers both **memory block** and **memory pool** mechanism.

Table of Contents

- **↓**DMM Features
- ↓ Memory Block
- ↓ Memory Pool
- ↓Usage
- ↓ Configuration

DMM Features

- memory block offers fixed size memory management.
- memory pool offers variable size memory management.
- both mechanisms support register,byte-alignment,usage check and thread-safety.

Memory Block

• register memory block

Depending on the RBS_CFG_APP_MODEL configuration whether or not using OS,one of two register interfaces should be used before using memory block.

If RBS_CFG_APP_MODEL is configurated to RBS_APP_OS, thread-safety version should be used.

hDMM RBS_DMM_RegisterBlock(char *pName,void *pDM,USIZE Size,USIZE BlockSize,HMUTEX hMutex)

If RBS_CFG_APP_MODEL is configurated to others.

hDMM RBS_DMM_RegisterBlock(char *pName,void *pDM,USIZE Size,USIZE BlockSize)

• allocate and free block

hBLOCK RBS_DMM_AllocZeroBlock(hDMM hDmm)

hBLOCK RBS_DMM_AllocBlock(hDMM hDmm)

BOOL RBS_DMM_FreeBlock(hBLOCK hBlock)

• block usage

void* RBS_DMM_UseHBlock(hBLOCK hBlock)

BOOL RBS_DMM_UnuseHBlock(hBLOCK hBlock)

Memory Pool

• register memory pool

Depending on the RBS_CFG_APP_MODEL configuration whether or not using OS,one of two register interfaces should be used befor using memory pool.

If RBS_CFG_APP_MODEL is configurated to RBS_APP_OS, thread-safety version should be used.

hDMM RBS_DMM_RegisterPool(char *pName,void *pDM,USIZE Size,BOOL bAntiFrag,U16 HCount,HMUTEX hMutex)

If RBS_CFG_APP_MODEL is configurated to others.

hDMM RBS_DMM_RegisterPool(char *pName,void *pDM,USIZE Size,BOOL bAntiFrag,U16 HCount)

• allocate and free pool

hMEM RBS_DMM_AllocZeroMem(hDMM hDmm,USIZE size)

hMEM RBS_DMM_AllocMem(hDMM hDmm,USIZE size)

hMEM RBS_DMM_ReallocMem(hMEM hMem,USIZE size)

BOOL RBS_DMM_FreeMem(hMEM hMem)

• pool usage

void* RBS_DMM_UseHMem(hMEM hMem)

BOOL RBS_DMM_UnuseHMem(hMEM hMem)

USIZE RBS_DMM_GetHMemSize(hMEM hMem)

Usage

In an embedded system, there are some memory types with different size such as internal SRAM integrated in micro-controller, external SDRAM, SRAM, etc... and those memory can be byte-addressed in the same address space. Some of them are used for stacks, while some used for global data and some used for dynamic memory allocation. So the specific usage of those memory are decided by the application.

Unlike traditional memory management like malloc in C library,DMM can manage more than one continuous byte-addressed memory for dynamic memory allocation and can be sensitive to memory usage.

As an example, a micro-controller has 8K-bytes size internal RAM in and external 2M-bytes SDRAM. You want to use 2K-bytes internal RAM and 1M-bytes external SDRAM space for dynamic memory allocation. First of all, you just define two byte array, one for the 2K internal RAM and the other for the 1M-bytes external SDRAM, and the use the register interface to define the two memory space. After register, you can allocate memory from them by the DMM handle returned from register.

Allocated memory will return a memory handle and then you can use the memory handle to use, not use or free the allocated memory.

Using **RBS_DMM_UseHBlock()** Or **RBS_DMM_UseHMem()** to refer to the allocated memory space.

Using **RBS_DMM_UnuseHBlock()** Or BS_DMM_UnuseHMem() to notify the DMM to end the last memory referance.

Using **RBS_DMM_FreeBlock()** Or **RBS_DMM_FreeMem()** to notify the DMM to release the memory.

RBS_DMM_UseHBloc() and BRS_DMM_UnuseHBlock() are used in pair,while **RBS_DMM_UseHMem()** and **RBS_DMM_UnuseHMem()** are in pair.The reason is that each memory handle has a referance counter in internal,and the counter is increased by 1 per usage and decrease by 1 per no usage.So if the counter is not initial value, it will not be released successfully by RBS_DMM_FreeBlock() Or RBS_DMM_FreeMem(). This is called usage check. Developers should keep in mind that after each reference to the allocated memory by RBS_DMM_UseHBlock() Or RBS_DMM_UseHMem(), an RBS_DMM_UnuseHBlock() Or RBS_DMM_UnuseHMem() should be called and if he wants to refer to the memory again, then RBS_DMM_UseHBlock() or BS_DMM_UseHMem() should be called again. With usage check, developers will find memory usage conflicts and bugs earlier. For example, there are two running tasks A and B that both using an allocated memory. If task A is using the memory without calling RBS_DMM_UnuseHBlock() or RBS_DMM_UnuseHMem() and task B call RBS_DMM_FreeBlock() or RBS_DMM_FreeMem(), then an error information will be printed by debug component, telling that it can not release memory handle because it is using.

Remember that usage check is only effective if RBS debug component is enabled, otherwise usage check is disabled. Following is the example code for DMM usage.

```
1 /*define array*/
2 U8 aInternalRAM[2048]
  _attribute__((at(0x20000000)));//2K-bytes
internal RAM starting from 0x20000000
3 U8 aExternalRAM[1024*1024]
  _attribute___((at(0x08011000)));//1M-bytes
external SDRAM starting from 0x08011000
4 /*register*/
5 hDMM hBlockDMM = RBS_DMM_RegisterBlock("Block
DMM", aInternalRAM, sizeof(aInternalRAM), 60);//re
gister a memory block with 60 bytes per block.
6 hDMM hPoolDMM = RBS_DMM_RegisterPool("Pool
DMM", aExternalRAM, sizeof(aExternalRAM), FALSE, 20
0);//register a memory pool with 200 handle
available.
7 /*allocate*/
8 hBLOCK hBlock =
RBS DMM AllocBlock(hBlockDMM);//allocate a
block from hBlockDMM
```

9 hMEM hMem =
 RBS_DMM_AllocMem(hPoolDMM,200);//allocate 200
 bytes from hPoolDMM
10 /*reference*/
11 void *pBlock =
 RBS_DMM_UseHBlock(hBlock);//using hBlock
12 void *pMem = RBS_DMM_UseHMem(hMem);//using
 hMem
13 ...//using memory
14 RBS_DMM_UnuseHBlock(hBlock);//not use hBlock
15 RBS_DMM_UnuseHMem(hMem);//unusing hMem
16 /*free*/
17 RBS_DMM_FreeBlock(hBlock);//release hBlock
18 RBS_DMM_FreeMem(hMem);//release hMem

Configuration

RBS_CFG_DMM_ALIGN is defined for define byte-alignment.3 means 8 bytes,2 means 4 bytes,1 means 2 bytes,0 means 1 bytes.

Allocated memory started address and size are both alignment.So if RBS_CFG_DMM_ALIGN is 2 and you allocated 25 bytes,then you get 28 bytes actually.

Generated by doxyden 1.8.9.1



Chapter5 Debug

Debug is important for software development and RBS contains a useful debug component. Detail references to **debug**

Table of Contents

- ↓ Debug Features
- ↓ Debug Usage
- How to configure debug component
- How to port debug component

Debug Features

- four optional debug levels:none,error,warning,log
- output re-targeting
- thread-safety support

Debug Usage

Four types debug macros are available.

• debug statement

RBS_DEBUG_STA(sta)

RBS_DEBUG_IF(exp,sta)

RBS_DEBUG_ELSIF(sta1,exp,sta2)

RBS_DEBUG_ENDIF(sta)

• debug error output

RBS_DEBUG_ERROR(exp,s)

RBS_DEBUG_ERROR_FORMAT(exp,format,...)

• debug warning output

RBS_DEBUG_WARN(exp,s)

RBS_DEBUG_WARN_FORMAT(exp,format,...)

• debug log output

RBS_DEBUG_LOG(s)

RBS_DEBUG_LOG_FORMAT(format,...)

How to configure debug component

There are two macro configuration for debug component.

- RBS_CFG_DEBUG_LEVEL is the configuration for debug level. You can define RBS_CFG_DEBUG_LEVEL from one of the four optional debug levels macros.
 - RBS_DEBUG_LEVEL_NOCHECK disable all debug statement and output
 - RBS_DEBUG_LEVEL_ERRORS only enable debug statement and debug error output
 - RBS_DEBUG_LEVEL_WARNINGS debug warning output plus BS_DEBUG_LEVEL_ERRORS
 - RBS_DEBUG_LEVEL_ALL debug log output plus BS_DEBUG_LEVEL_WARNINGS
- RBS_CFG_DEBUG_BUFSIZE cofigure the debug buffer size in bytes.Default size is 300 bytes.One output string length should not exceed debug buffer size.

How to port debug component

- Depending to debug level you configure RBS_CFG_DEBUG_LEVEL, different porting functions are needed. And you can re-targeting output by these porting functions.
 - **RBS_DEBUG_LEVEL_ERRORS needs**:

void Port_Printf_Error(const char *s) for debug error output

• RBS_DEBUG_LEVEL_WARNINGS **needs**:

void Port_Printf_Error(const char *s) for debug error output void Port_Printf_Warn(const char *s) for debug warning output

• BS_DEBUG_LEVEL_ALL needs:

void Port_Printf_Error(const char *s) for debug error output void Port_Printf_Warn(const char *s) for debug warning output void Port_Printf_Log(const char *s) for debug log output

• If application model RBS_CFG_APP_MODEL is configured to RBS_APP_OS,then OS lock such as mutex or semaphore should used for thread-safe debug.

Debug error output(RBS_DEBUG_LEVEL_ERRORS),debug warning output(RBS_DEBUG_LEVEL_WARNINGS) and debug log(RBS_DEBUG_LEVEL_ALL) output use the same lock. And three following porting functions should be present.

BOOL Port_Init(void) for creating debug lock BOOL Port_GetMutex(HMUTEX hMutex) for lock debug output BOOL Port_FreeMutex(HMUTEX hMutex) for unlock debug output



Chapter6 Configration

RBS configuration is defined by file **RainbowBSConf.h**. You can write or modify this file or use PC tool RAutoConfig.exe to generate this file.

Table of Contents

- - ↓ System and CPU
 - Application model
 - **↓**DMM
 - ↓ Debug
- Using RAutoConfig

Configuration class

System and CPU

• RBS_CFG_SYS_INFO

define a string for describing the application.

• RBS_CFG_TICK_RATE

timer tick per second. The valid value should be in rang of 1 to 1000 and 1000 should be divided with no remainder by it.

• RBS_CFG_CPU_WORD_SIZE

definition of CPU word size 8bits/16bits/32bits.

• RBS_CFG_CPU_BYTE_ORDER_L

CPU byte order.1 means little-endia and 0 means big-endia.

• RBS_CFG_CPU_STACK_DOWN

stack growth direction.1 means growing to low address and 0 means growing to high address.

Application model

• RBS_CFG_APP_MODEL

one of three application models should be chosen.

- RBS_APP_NONE traditional event-driven model.
- RBS_APP_PTP protothread event-driven model.
- RBS_APP_OS multitask model.OS kernel needed.
- HMUTEX

type of mutex defined by OS used. This has effect only when RBS_CFG_APP_MODEL is configured to RBS_APP_OS.

• Head file name

head file offered by OS used. This has effect only when RBS_CFG_APP_MODEL is configured to RBS_APP_OS.

• RBS_CFG_PTP_N0_PROCESS_NAME

process name.1 means disabled,0 means enabled. This has effect only when RBS_CFG_APP_MODEL is configured to RBS_APP_PTP.

• RBS_CFG_PTP_PROCESS_STATS

used to statistics for events used.1 means enabled,0 means disabled.This has effect only when RBS_CFG_APP_MODEL is configured to RBS_APP_PTP.

• RBS_CFG_PTP_NUMEVENTS

event count. This has effect only when RBS_CFG_APP_MODEL is configured to RBS_APP_PTP.

DMM

• RBS_CFG_DMM_ALIGN

define the alignment bytes for memory address and size.3 means 8 bytes,2 means 4 bytes,1 means 2 bytes,0 means 1 byte.

Debug

• RBS_CFG_DEBUG_LEVEL

debug output level.one of four should be chosen.

- RBS_DEBUG_LEVEL_NOCHECK disable debug output.no running time checks are performed.
- RBS_DEBUG_LEVEL_ERRORS errors are recorded.
- RBS_DEBUG_LEVEL_WARNINGS errors and warnings are recorded.
- RBS_DEBUG_LEVEL_LOG errors, warnings and logs are recorded.
- RBS_CFG_DEBUG_BUFSIZE

output buffer length. This has effect only when RBS_CFG_DEBUG_LEVEL is not configured to RBS_DEBUG_LEVEL_NOCHECK.

Using RAutoConfig

RAutoConfig is in "RainbowBS\RBSTools".

RainbowBS	
System App Model Dynamic Memory Debug	SysInfo.: RainbowSys Tick Rate : 100 CPU Word Size : 32 •
Tick Rate ticks[1,1000] per second. 1000 should be divided with no	Stack Type : Full,Grow Down
	Generate File
	Rainbow AutoConfig

Generated by



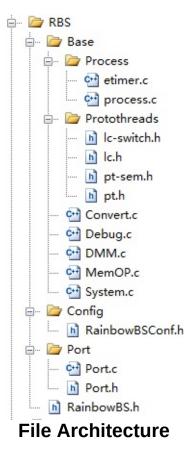
Chapter7 Porting and Example

Currently,RBS has been ported on PC and ARM Cortex-M MCU.Also it can be ported on any other architecture by reference to that.

Table of Contents

- $\operatorname{\Psi}\operatorname{Ported}$ on PC
- [↓]Ported on Cortex-M

Recommend project file architecture as following:



Ported on PC

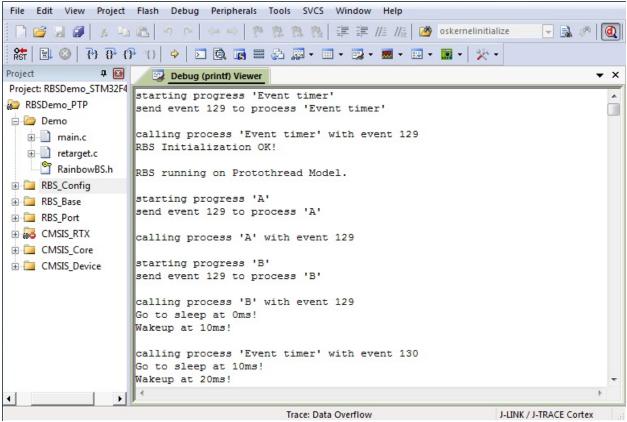
Refer to the example in

"RainbowBS/RBSDemoProjects/X86_VS2008/RBSDemo_VS2008.sln". There are three configuration targets corresponding to the three RBS application models respectively.

Ported on Cortex-M

Refer to the example in

"RainbowBS/RBSDemoProjects/STM32F4_MDK/RBSDemo_STM32F4.u" There are three configuration targets corresponding to the three RBS application models respectively. The standard output is defined to use SWO in the example.You can use Keil-MDK DebugViewer or JLinkSWOViewer(if JLink is used as adaptor) to view.



Output in DebugViewer

File Edit Help		
31 24 23 16 15 8 7 0 Data from stimulus port(s):		
Stay on top Clear		
calling process 'Event timer' with event 130 Go to sleep at 20580ms! Wakeup at 20590ms!		
calling process 'Event timer' with event 130 Go to sleep at 20590ms! Wakeup at 20600ms!		
calling process 'Event timer' with event 130 process 'Event timer' posts event 136 to process 'A' process 'Event timer' posts event 136 to process 'B'		
<u>۲</u>		
Device: STM32F407ZG CPUFreq: 168030 kF SWOFreq: 6000 kHz 227196 bytes		

Output in JLinkSWOViewer

Generated by



Modules

Here is a list of all modules:

	[detail level 12]
RBS configration	
basic types	
basic macros	
system releted	
CPU information	
memory operation	
conversion releted	
dynamic memory management	
protothread process	
Event timers	
debug	
▼ Pt	
Local continuations	
Protothread semaphores	

Generated by doxyden 1.8.9.1



RBS configration

Macros

Macros

#define RBS_CFG_SYS_INFO "RainbowSys"

#define RBS_CFG_TICK_RATE 100

#define **RBS_CFG_CPU_WORD_SIZE** 32 /* CPU word size */

#define **RBS_CFG_CPU_BYTE_ORDER_L** 1 /* CPU byte order */

#define **RBS_CFG_CPU_STACK_DOWN** 1 /* stack growth direction */

#define **RBS_APP_NONE** 0

#define **RBS_APP_PTP** 1

#define **RBS_APP_OS** 2

#define **RBS_CFG_APP_MODEL** RBS_APP_PTP

#define **RBS_CFG_PTP_NO_PROCESS_NAME** 0

#define RBS_CFG_PTP_PROCESS_STATS 0

#define RBS_CFG_PTP_NUMEVENTS 32

#define RBS_CFG_DMM_ALIGN 2

#define **RBS_DEBUG_LEVEL_NOCHECK** 0 /* No running time checks are performed */

#define **RBS_DEBUG_LEVEL_ERRORS** 1 /* Errors are recorded */

#define **RBS_DEBUG_LEVEL_WARNINGS** 2 /* Errors and Warnings are recorded */

#define **RBS_DEBUG_LEVEL_LOG** 3 /* Errors,Warnings and logs are recorded */

#define RBS_CFG_DEBUG_BUFSIZE 300

#define **RBS_CFG_DEBUG_LEVEL** RBS_DEBUG_LEVEL_LOG

Detailed Description

Generated by



basic types

Typedefs

Typedefs

typedef uint8_t BOOL

typedef int8_t **I8**

typedef uint8_t U8

typedef int16_t **I16**

typedef uint16_t **U16**

typedef int32_t **I32**

typedef uint32_t **U32**

typedef int64_t **I64**

typedef uint64_t **U64**

typedef size_t **USIZE**

typedef ptrdiff_t IPTRDIFF

typedef U64 TICK

Detailed Description

basic types defination.

Generated by



basic macros

Macros

Macros

- #define FALSE Ou
- #define TRUE 1u
- #define USE_PARA(para) (para = (para))
- #define **MIN**(v0, v1) (((v0) > (v1))? (v1): (v0))
- #define MAX(v0, v1) (((v0) > (v1))? (v0) : (v1))
- #define **ABS**(v) (((v) >= 0) ? (v) : (-(v)))
- #define **SWAP**(a, b) (a = (a) + (b), b = (a) (b), a = (a) (b))
- #define COUNT_OF(a) (sizeof(a)/sizeof(a[0]))
- #define _STR(a) #a
- #define STR(a) _STR(a)
- #define _CONS(a, b) a##b
- #define CONS(a, b) _CONS(a,b)
- #define ALIGN_F(pointer, power2) (((IPTRDIFF)(pointer) + ((IPTRDIFF)((power2) 1))) & (~((IPTRDIFF)((power2) 1))))
- #define **ALIGN_B**(pointer, power2) ((IPTRDIFF)(pointer) & (~ ((IPTRDIFF)((power2) 1))))

Detailed Description

basic macros defination.

Generated by



Data Structures | Functions

system releted

Data Structures

struct tTime

Time structure. More...

Functions

BOOL **RBS_Init (void)** Initialize RainbowBS. More...

system information

const char * RBS_GetVersionString (void) Get RainbowBS version string. More...

const char * **RBS_GetSysInfo (void)** Get system description string. More...

time releated

TICK **RBS_GetTickCount** (void) Get system tick. More...

U64 **RBS_GetRunTime (void)** Get system running time(ms). More...

void RBS_GetLocalTime (tTime *ptTime) Get system local time. More...

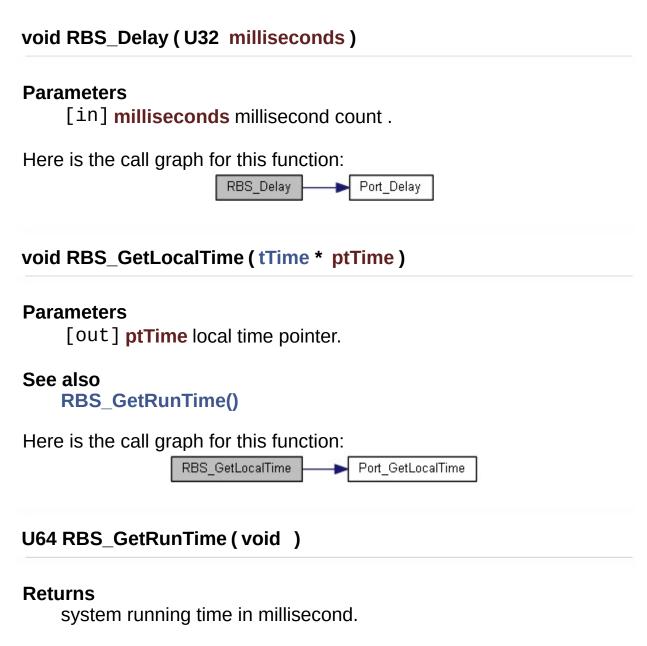
void **RBS_Delay** (U32 milliseconds) Delay some milliseconds. More...

#define RBS_TICK_MS (1000u/RBS_CFG_TICK_RATE)

Detailed Description

Implementation of system releted.

Function Documentation



See also

RBS_GetLocalTime()

Here is the call graph for this function:

RBS_GetRunTime Port_GetTickCount

const char* RBS_GetSysInfo (void)

Returns

system description string.

See also

RBS_GetVersionString()

TICK RBS_GetTickCount (void)

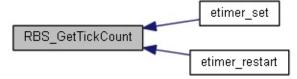
Returns

system tick.

Here is the call graph for this function:

RBS_GetTickCount Port_GetTickCount

Here is the caller graph for this function:



const char* RBS_GetVersionString (void)

Returns

version string.

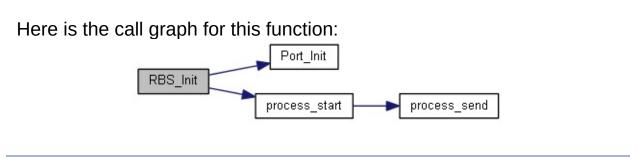
See also

RBS_GetSysInfo()

BOOL RBS_Init (void)

Return values TRUE successfully.

FALSE failed.



Generated by



Data Fields

tTime Struct Reference

system releted

Time structure.

#include <RainbowBS.h>

Collaboration diagram for tTime:

tTime
+ year + month + day + day_of_week + hour + minute + second + milliseconds

Data Fields

U16	year
U8	month
U8	day
U8	day_of_week
U8	hour
U8	minute
U8	second
U16	milliseconds

Field Documentation

U8 day
day of month[1,31]
U8 day_of_week
day of week[0,6]
U8 hour
hour[0,23]
U16 milliseconds
milliseconds[0,999]
U8 minute
minute[0,59]
U8 month
month[1,12]
U8 second

second[0,59]

U16 year

year[1601,30827]

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

• RainbowBS.h

Generated by doxyden 1.8.9.1



Functions

CPU information

Functions

- U8 **RBS_GetCPUBits (void)** CPU word-width. More...
- BOOL **RBS_IsCPULittleEndian (void)** Check if CPU is little-endian. More...
- BOOL **RBS_IsStackGrowDown (void)** Check if stack grows down. More...

Detailed Description

Information about CPU.

Function Documentation

U8 RBS_GetCPUBits (void)

Returns

CPU word-width(8/16/32/64).

BOOL RBS_IsCPULittleEndian (void)

Return values

TRUE little-endian. **FALSE** big-endian.

BOOL RBS_IsStackGrowDown (void)

Return values

TRUE growth down. **FALSE** growth up.

Generated by doxyden 1.8.9.1



Functions

memory operation

Functions

- void **RBS_MemSet8** (U8 *pDes, U8 fill, USIZE num) Set memory by byte unit. More...
- void **RBS_MemSet16** (U16 *pDes, U16 fill, USIZE num) Set memory by two-bytes unit. More...
- void **RBS_MemSet32** (U32 *pDes, U32 fill, USIZE num) Set memory by four-bytes unit. More...
- void **RBS_MemCpy8** (U8 *pDes, const U8 *pSrc, USIZE num) Copy memory by byte unit. More...
- U16 **RBS_Read16L** (const U8 **ppData) Read a 16-bits entity in little-endian. More...
- U32 RBS_Read32L (const U8 **ppData) Read a 32-bits entity in little-endian. More...
- void **RBS_Write16L** (U8 **ppData, U16 data) Write a 16-bits entity in little-endian. More...
- void **RBS_Write32L** (U8 **ppData, U32 data) Write a 32-bits entity in little-endian. More...
- U16 RBS_Read16B (const U8 **ppData) Read a 16-bits entity in big-endian. More...
- U32 **RBS_Read32B** (const U8 **ppData) Read a 32-bits entity in big-endian. More...
- void **RBS_Write16B** (U8 **ppData, U16 data) Write a 16-bits entity in big-endian. More...

void **RBS_Write32B** (U8 **ppData, U32 data) Write a 32-bits entity in big-endian. More...

Detailed Description

Memory set and copy operation.

Function Documentation

void RBS_MemCpy8 (U8 *	pDes,
	const U8 *	pSrc,
	USIZE	num
)		

Parameters

[out] **pDes** destination pointer.

- [in] **pSrc** source pointer.
- [in] **num** total units.

Here is the caller graph for this function:

RBS_MemCpy8 < RBS_DMM_ReallocMem

void RBS_MemSet16 (U16 * pDes, U16 fill, USIZE num)

Parameters [out]pDes destination pointer(two-bytes alignment). [in] fill two-bytes filled. [in] num total units.	
void RBS_MemSet32 (U32 * pDes, U32 fill, USIZE num)	

Parameters

[out] **pDes** destination pointer(four-bytes alignment).

[in] fill four-bytes filled.

[in] **num** total units.

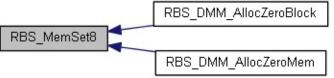
```
void RBS_MemSet8 (U8 * pDes,
U8 fill,
USIZE num
)
```

Parameters

[out] **pDes** destination pointer.

- [in] fill byte filled.
- [in] **num** total units.

Here is the caller graph for this function:



U16 RBS_Read16B (const U8 ** ppData)

Parameters

[in, out] **ppData** data pointer, added 2 bytes after calling.

Returns

data entity

U16 RBS_Read16L (const U8 ** ppData)

Parameters

[in, out] **ppData** data pointer, added 2 bytes after calling.

Returns

data entity

U32 RBS_Read32B (const U8 ** ppData)

Parameters

[in, out] **ppData** data pointer, added 4 bytes after calling.

Returns

data entity

U32 RBS_Read32L (const U8 ** ppData)

Parameters

[in, out] **ppData** data pointer, added 4 bytes after calling.

Returns

data entity

void RBS_Write16B (U8 ** ppData, U16 data

)

Parameters

[in, out] ppData data pointer,added 2 bytes after calling. [in] data data to be wtitten.

void RBS_Write16L (U8 ** ppData, U16 data)

Parameters

[in, out] **ppData** data pointer, added 2 bytes after calling.

[in] **data** data to be wtitten.

void RBS_Write32B (U8 ** ppData, U32 data)

Parameters

[in, out] ppData data pointer,added 4 bytes after calling. [in] data data to be wtitten.

void RBS_Write32L (U8 ** ppData, U32 data)

Parameters

[in, out] ppData data pointer,added 4 bytes after calling. [in] data data to be wtitten.

Generated by doxyden 1.8.9.1



conversion releted

convert digit to string

enum ePOW { SHEX, SBINARY, SDECIMAL } Specific hex, binary or decimal. More...

U8 RBS_Number2String (U32 value, ePOW ePow, U8 length, char *pText)

Convert unsigned digit to ASCII string. More...

trigonometric releted

int **RBS_sin** (int angle) Calculate sine. More...

int **RBS_cos** (int angle) Calculate cosine. More...

int **RBS_tan (int angle)** Calculate tangent. More...

int **RBS_ctan** (int angle) Calculate cotangent. More...

#define ANG_45DEG 1024

#define ANG_90DEG (2*ANG_45DEG)

#define ANG_135DEG (3*ANG_45DEG)

#define ANG_180DEG (4*ANG_45DEG)

#define ANG_225DEG (5*ANG_45DEG)

#define ANG_270DEG (6*ANG_45DEG)

#define ANG_315DEG (7*ANG_45DEG)

#define ANG_360DEG (8*ANG_45DEG)

Detailed Description

Enumeration Type Documentation

enum ePOW

Enumerator	
SHEX	hex
SBINARY	binary
SDECIMAL	decimal

Function Documentation

int RBS_cos (int angle)

This function calculates cosine without using float-point numbers. It use a table to look up for the appropriate value.

Parameters

[in] angle angle=degrees*ANG_45DEG/45,degrees=angle*45/AN

Returns

cos(degrees)*1024.

int RBS_ctan (int angle)

This function calculates cotangent without using float-point numbers. It us constant table to look up for the appropriate value.

Parameters

[in] angle angle=degrees*ANG_45DEG/45,degrees=angle*45/AN

Returns

cotan(degrees)*1024.

U8 RBS_Number2String (U32	value,
	ePOW	ePow,
	U8	length,
	char *	pText
)		

Parameters

[in] value unsigned digit.

- [in] **ePow** choose hex, binary or decimal.
- [in] **length** Max.bits from lowest bit.If Len is 0,choosing Min.characters automatically.

[out] **pText** output string buffer.

Returns

character number.

See also

ePOW

int RBS_sin (int angle)

This function calculates sine without using float-point numbers. It use a c table to look up for the approximate value.

Parameters

[in] angle angle=degrees*ANG_45DEG/45,degrees=angle*45/AN

Returns

sin(degrees)*1024.

Example:

1 //calculate sine of 30 degrees,the return valu
511 which

- 2 //is approximately equal to sin30*1024 which i
- 3 int value = RBS_sin(30*ANG_45DEG/45);

int RBS_tan (int angle)

This function calculates tangent without using float-point numbers. It use table to look up for the appropriate value.

Parameters

[in] **angle** angle=degrees*ANG_45DEG/45,degrees=angle*45/AN

Returns

tan(degrees)*1024.

Generated by



Macros | Typedefs | Functions

dynamic memory management

Macros

#define **HDMM_NULL** NULL

#define **HBLOCK_NULL** NULL

#define **HMEM_NULL** NULL

Typedefs

typedef void * hDMM

typedef void * hBLOCK

typedef void * **hMEM**

Functions

- hDMM RBS_DMM_RegisterBlock (char *pName, void *pDM, USIZE size, USIZE block_size, HMUTEX hMutex) Register the memory area as a dynamic block memory area. More...
- hBLOCK RBS_DMM_AllocZeroBlock (hDMM hDmm) Allocate a free block with zero initialization from the dynamic block memory area. More...
- hBLOCK RBS_DMM_AllocBlock (hDMM hDmm) Allocate a free block from the dynamic block memory area. More...
 - BOOL **RBS_DMM_FreeBlock** (hBLOCK hBlock) Free the allocated block. More...
 - void * **RBS_DMM_UseHBlock** (hBLOCK hBlock) Get the allocated block pointer. More...
 - BOOL **RBS_DMM_UnuseHBlock** (hBLOCK hBlock) Unuse the allocated block area. More...
 - hDMM RBS_DMM_RegisterPool (char *pName, void *pDM, USIZE size, BOOL bAntiFrag, U16 handle_count, HMUTEX hMutex) Register the memory area as a dynamic pool memory area. More...
 - hMEM RBS_DMM_AllocZeroMem (hDMM hDmm, USIZE size) Allocate a free memory with zero initialization from the dynamic memory area. More...
 - hMEM RBS_DMM_AllocMem (hDMM hDmm, USIZE size) Allocate a free memory from the dynamic memory area. More...

- hMEM **RBS_DMM_ReallocMem** (hMEM hMem, USIZE size) Reallocate a free memory from the same dynamic memory area. More...
- BOOL **RBS_DMM_FreeMem (hMEM hMem)** Free the allocated memory. More...
- void * **RBS_DMM_UseHMem (hMEM hMem)** Get the allocated memory pointer. More...
- BOOL **RBS_DMM_UnuseHMem (hMEM hMem)** Unuse the allocated memory area. More...
- USIZE **RBS_DMM_GetHMemSize** (hMEM hMem) Get the allocated memory size. More...

Detailed Description

Dynamic memory management for both memory poll and block.

Function Documentation

hBLOCK RBS_DMM_AllocBlock (hDMM hDmm)

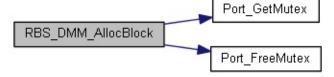
Parameters

[in] **hDmm** handle of dynamic block memory area.

Return values

HBLOCK_NULL failed. others handle of dynamic memory area.

Here is the call graph for this function:



Here is the caller graph for this function:



hMEM RBS_DMM_AllocMem (hDMM hDmm, USIZE size

Parameters

[in] **hDmm** handle of dynamic memory area.

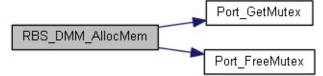
)

[in] **size** memory area bytes.

Return values

HMEM_NULL failed.othershandle of dynamic memory area.

Here is the call graph for this function:



Here is the caller graph for this function:

RBS_DMM_AllocMem RBS_DMM_AllocZeroMem

hBLOCK RBS_DMM_AllocZeroBlock (hDMM hDmm)

Parameters

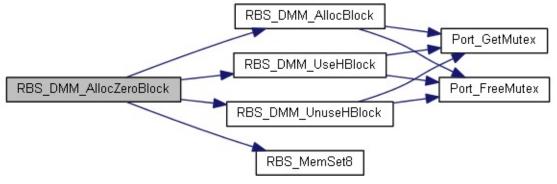
[in] **hDmm** handle of dynamic block memory area.

Return values

HBLOCK_NULL failed.

others handle of dynamic memory area.

Here is the call graph for this function:



hMEM RBS_DMM_AllocZeroMem (hDMM hDmm, USIZE size)

Parameters

[in] **hDmm** handle of dynamic memory area.

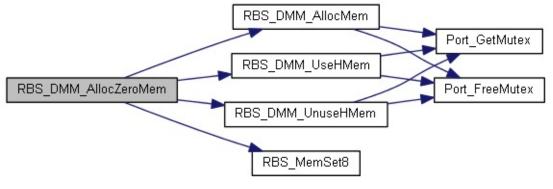
[in] **size** memory area bytes.

Return values

HMEM_NULL failed.

others handle of dynamic memory area.

Here is the call graph for this function:



BOOL RBS_DMM_FreeBlock (hBLOCK hBlock)

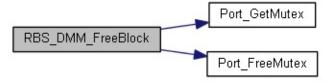
Parameters

[in] **hBlock** handle of a allocated block.

Return values

TRUE successful. **FALSE** failed.

Here is the call graph for this function:



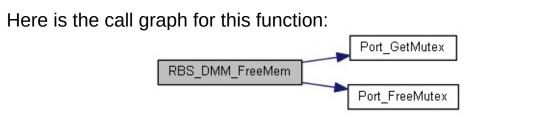
BOOL RBS_DMM_FreeMem (hMEM hMem)

Parameters

[in] **hMem** handle of a allocated memory.

Return values

TRUE successful. **FALSE** failed.



USIZE RBS_DMM_GetHMemSize (hMEM hMem)

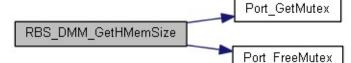
Parameters

[in] **hMem** handle of the allocated memory.

Returns

memory area size.

Here is the call graph for this function:



hMEM RBS_DMM_ReallocMem (hMEM hMem, USIZE size)

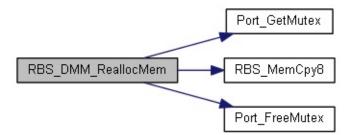
Parameters

[in] hMem handle of an allocated memory. [in] size memory area bytes.

Return values

HMEM_NULL failed.othershandle of dynamic memory area.

Here is the call graph for this function:



hDMM RBS_DMM_RegisterBlock (char * pName, void * pDM, USIZE size, USIZE block_size, HMUTEX hMutex)

Parameters

[in] pNameblock memory name.[in] pDMpointer to the block memory area.[in] sizeblock memory area size(bytes).[in] block_sizeeach block size.[in] hMutexmemory block mutex for thread-safe.If NULL,no thread-safe support.

Return values

HDMM_NULL failed.

other handle of dynamic block memory area.

Here is the call graph for this function:



RBS_DMM_RegisterBlock

Here is the caller graph for this function:

hDMM RBS_DMM_RegisterPool (cha	ar * pName,
voi	d * pDM,
USI	IZE size,
BO	OL bAntiFrag,
U16	6 handle_count,
HM	UTEX hMutex
)	

Parameters

[in]	pName	memory pool name.
[in]	рDМ	pointer to the memory area.
[in]	size	memory area size(bytes).
[in]	bAntiFrag	enable defragmentation.
[in]	handle_count	count of memory handles.
[in]	hMutex	memory pool mutex for thread-safety.If NULL,no thread-safe support.

Return values

NotHDMM_NULL successful.HDMM_NULL failed.

Here is the call graph for this function:



Here is the caller graph for this function:



BOOL RBS_DMM_UnuseHBlock (hBLOCK hBlock)

Parameters

[in] **hBlock** handle of a allocated block.

Returns

block area pointer.

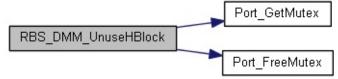
See also

RBS_DMM_UseHBlock()

Note

used in pair with RBS_DMM_UseHBlock()

Here is the call graph for this function:



Here is the caller graph for this function:

RBS_DMM_UnuseHBlock RBS_DMM_AllocZeroBlock

BOOL RBS_DMM_UnuseHMem (hMEM hMem)

Parameters

[in] **hMem** handle of a allocated memory.

Returns

memory area pointer.

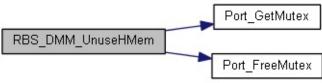
See also

RBS_DMM_UseHMem()

Note

used in pair with RBS_DMM_UseHMem()

Here is the call graph for this function:



Here is the caller graph for this function:



void* RBS_DMM_UseHBlock (hBLOCK hBlock)

Parameters

[in] **hBlock** handle of a allocated block.

Returns

block area pointer.

See also

RBS_DMM_UnuseHBlock()

Note

used in pair with RBS_DMM_UnuseHBlock()

Here is the call graph for this function:

RBS DMM UseHBlock

Port_FreeMutex

Port GetMutex

Here is the caller graph for this function:

RBS_DMM_UseHBlock RBS_DMM_AllocZeroBlock

void* RBS_DMM_UseHMem (hMEM hMem)

Parameters

[in] **hMem** handle of a allocated memory.

Returns

memory area pointer.

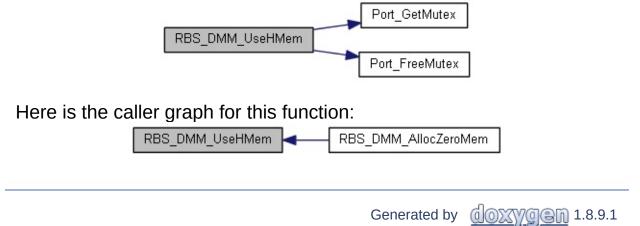
See also

RBS_DMM_UnuseHMem()

Note

used in pair with **RBS_DMM_UnuseHMem()**

Here is the call graph for this function:





Data Structures | Macros | Typedefs

protothread process

Data Structures

struct process_struct Process structure(one process consists one protothread). More...

Macros

#define **PROCESS_NONE** NULL

#define **PROCESS_BROADCAST** NULL

Typedefs

typedef U8 process_event_t

typedef void * process_data_t

typedef struct process_struct tPROCESS Process structure(one process consists one protothread). More... predefined event type

#define **PROCESS_EVENT_NONE** 0x80

#define **PROCESS_EVENT_INIT** 0x81

#define **PROCESS_EVENT_POLL** 0x82

#define **PROCESS_EVENT_EXIT** 0x83

#define **PROCESS_EVENT_SERVICE_REMOVED** 0x84

#define **PROCESS_EVENT_CONTINUE** 0x85

#define **PROCESS_EVENT_MSG** 0x86

#define **PROCESS_EVENT_EXITED** 0x87

#define **PROCESS_EVENT_TIMER** 0x88

#define **PROCESS_EVENT_COM** 0x89

#define **PROCESS_EVENT_MAX** 0x8a

process declaration and definition

- #define **PROCESS_THREAD**(name, ev, data) Define the body of a process. More...
- #define **PROCESS_NAME(name)** extern **tPROCESS** name Declare the name of a process. More...
- #define **PROCESS**(name, strname) Define a process. More...

semaphore declaration and definition

#define SEM_NAME(name) extern struct pt_sem sem_##name Declare the name of a semaphore. More...

#define SEM(name, count) Define a semaphore. More...

process protothread functions

- #define PROCESS_BEGIN() Define the beginning of a process. More...
- #define PROCESS_END() Define the end of a process. More...
- #define PROCESS_WAIT_EVENT() Wait for an event to be posted to the process. More...
- #define PROCESS_WAIT_EVENT_UNTIL(c) Wait for an event to be posted to the process, with an extra condition. More...
- #define **PROCESS_YIELD()** Yield the currently running process.
- #define PROCESS_YIELD_UNTIL(c) Yield the currently running process until a condition occurs. More...
- #define PROCESS_WAIT_UNTIL(c) Wait for a condition to occur. More...
- #define PROCESS_WAIT_WHILE(c) Wait for a condition not occur. More...
- #define PROCESS_EXIT() Exit the currently running process.
- #define PROCESS_PT_SPAWN(pt, thread) Spawn a protothread from the process. More...

#define PROCESS_PAUSE()

Yield the process for a short while. More...

#define PROCESS_WAIT_SEM(name)

#define PROCESS_SIGNAL_SEM(name)

poll and exit handlers

#define **PROCESS_POLLHANDLER**(handler) if(ev == PROCESS_EVENT_POLL) { handler; } Specify an action when a process is polled. More...

#define **PROCESS_EXITHANDLER**(handler) if(ev == PROCESS_EVENT_EXIT) { handler; } Specify an action when a process exits. More...

process functions called from application programs

BOOL process_start (tPROCESS *ptProcess, process_data arg)

Start a process. More...

BOOL process_post (tPROCESS *ptProcess, process_even ev, void *data) Post an asynchronous event to one or all processes. More...

void process_send (tPROCESS *ptProcess, process_ever ev, void *data) Send a synchronous event to a process. More...

- void process_exit (tPROCESS *ptProcess) Cause a process to exit. More...
- process_event_t process_alloc_event (void) Allocate a global event number. More...
 - #define PROCESS_CURRENT() process_current Get a pointer to the currently running process. More...

#define #tmp_current = **PROCESS_CURRENT()**;process_current = **process** Switch context to another process. More...

#define	PROCESS_CONTEXT_END(ptProcess)	process_cu
	= tmp_current; }	
	End a context switch. More	

process functions called from device drivers.

BOOL process_poll (tPROCESS *ptProcess) Request a process to be polled. More...

functions called by the system and boot-up code

- U16 process_run (void) Run the system once - call poll handlers and process one event. More...
- BOOL process_is_running (tPROCESS *ptProcess) Check if a process is running. More...

U16 process_nevents (void)

Number of events waiting to be processed. More...

Detailed Description

A process in consists of a single **protothread**.

Macro Definition Documentation

#define PROCESS (name, strname)

This macro defines a process. The process has two names: the variable of the process structure, which is used by the C program, and a human readable string name, which is used when debugging. A configuration option allows removal of the readable name to save RAM.

Parameters

[in] name The variable name of the process structure. [in] strname The string representation of the process' name.

Note

definition must be global.

#define PROCESS_BEGIN ()

This macro defines the beginning of a process, and must always appear in a **PROCESS_THREAD()** definition. The **PROCESS_END()** macro must come at the end of the process.

{ tPROCESS *tmp_cur #define PROCESS_CURRENT(); PROCESS_CONTEXT_BEGIN (ptProcess) = ptProcess

This function switch context to the specified process and executes the cc that process. Typical use of this function is to switch context in services, processes. Each **PROCESS_CONTEXT_BEGIN()** must be followed by t

PROCESS_CONTEXT_END() macro to end the context switch.

Parameters

[in] **ptProcess** The process to use as context

See also

PROCESS_CONTEXT_END() PROCESS_CURRENT()

Example:

1 PROCESS_CONTEXT_BEGIN(&test_process); 2 etimer_set(&timer, CLOCK_SECOND);

3 PROCESS_CONTEXT_END(&test_process);

#define process_current = PROCESS_CONTEXT_END (ptProcess) tmp_current; }

This function ends a context switch and changes back to the previous process.

Parameters

[in] **ptProcess** The process used in the context switch

See also

PROCESS_CONTEXT_START()

#define PROCESS_CURRENT() process_current

This macro get a pointer to the currently running process. Typically, this macro is used to post an event to the current process with **process_post()**.

#define PROCESS_END ()

This macro defines the end of a process. It must appear in a

PROCESS_THREAD() definition and must always be included. The process exits when the **PROCESS_END()** macro is reached.

if(ev == #define PROCESS_EVENT_EXIT) { PROCESS_EXITHANDLER (handler) handler; }

Note

This declaration must come immediately before the **PROCESS_BEGIN()** macro.

Parameters

handler The action to be performed.

#define PROCESS_NAME (name) extern tPROCESS name

This macro is typically used in header files to declare the name of a process that is implemented in the C file.

#define PROCESS_PAUSE()

This macro yields the currently running process for a short while, thus letting other processes run before the process continues.

if(ev == #define PROCESS_EVENT_POLL) { PROCESS_POLLHANDLER (handler) handler; }

Note

This declaration must come immediately before the **PROCESS_BEGIN()** macro.

Parameters

handler The action to be performed.

#define PROCESS_PT_SPAWN (pt, thread

.....

)

Parameters

pt[inout] The protothread state (struct pt) for the new protothread

thread[in] The call to the protothread function.

See also

PT_SPAWN()

#define PROCESS_SIGNAL_SEM (name)

Signal a semaphore

This macro carries out the "signal" operation on the semaphore.

Parameters

[in] name semaphore name

Note

It's the application's responsibility to notify the waiting process if semaphore is avilable.

See also

PROCESS_WAIT_SEM()

#define PROCESS_THREAD (name,
	ev,
	data
)	

This macro is used to define the body (protothread) of a process. The

process is called whenever an event occurs in the system, A process always start with the **PROCESS_BEGIN()** macro and end with the **PROCESS_END()** macro.

#define PROCESS_WAIT_EVENT()

This macro blocks the currently running process until the process receives an event.

#define PROCESS_WAIT_EVENT_UNTIL(c)

This macro is similar to **PROCESS_WAIT_EVENT()** in that it blocks the currently running process until the process receives an event. But **PROCESS_WAIT_EVENT_UNTIL()** takes an extra condition which must be true for the process to continue.

Parameters

[in] **c** The condition that must be true for the process to continue.

See also

PT_WAIT_UNTIL()

#define PROCESS_WAIT_SEM (name)

Wait for a semaphore

This macro carries out the "wait" operation on the semaphore.

Parameters

[in] **name** semaphore name

Note

It's the application's responsibility to notify the waiting process if semaphore is avilable.

See also

PROCESS_SIGNAL_SEM()

#define PROCESS_WAIT_UNTIL (c)

This macro does not guarantee that the process yields, and should therefore be used with care. In most cases, **PROCESS_WAIT_EVENT()**, **PROCESS_WAIT_EVENT_UNTIL()**, **PROCESS_YIELD()** or **PROCESS_YIELD_UNTIL()** should be used instead.

Parameters

[in] **c** The condition that must be true for the process to continue

#define PROCESS_WAIT_WHILE(c)

This macro does not guarantee that the process yields, and should therefore be used with care. In most cases, **PROCESS_WAIT_EVENT()**, **PROCESS_WAIT_EVENT_UNTIL()**, **PROCESS_YIELD()** or **PROCESS_YIELD_UNTIL()** should be used instead.

Parameters

[in] **c** The condition that must be false for the process to continue.

#define PROCESS_YIELD_UNTIL (c)

This macro is different from **PROCESS_WAIT_UNTIL()** in that **PROCESS_YIELD_UNTIL()** is guaranteed to always yield at least once. This ensures that the process does not end up in an infinite loop and monopolizing the CPU.

Parameters

[in] **c** The condition to wait for.

#define SEM (name, count)

Parameters

name The variable name of the semaphore structure. **count** The count value of the semaphore.

Note

definition must be global.

#define SEM_NAME (name) extern struct pt_sem sem_##name

This macro is typically used in header files to declare the name of a semaphore that is implemented in the C file.

Typedef Documentation

typedef struct process_struct tPROCESS

Note

application code should not change any member of the process structure for they are maintained by system inside.

Function Documentation

process_event_t process_alloc_event(void)

event numbers above 128 are global and may be posted from one process to another. This function allocates one such event number.

Note

There currently is no way to deallocate an allocated event number.

Returns

The allocated event number

void process_exit (tPROCESS * ptProcess)

This function causes a process to exit. The process can either be the currently executing process, or another process that is currently running.

Parameters

[in, out] **ptProcess** The process that is to be exited

See also PROCESS_CURRENT()

BOOL process_is_running (tPROCESS * ptProcess)

This function checks if a specific process is running. A process can be on running state after calling **process_start()** normally.

Parameters

[in] **ptProcess** The process pointer.

Return values

TRUE if the process is running. **FALSE** if the process is not running.

Here is the caller graph for this function:



U16 process_nevents (void)

Returns

Number of events that are currently waiting to be processed.

BOOL process_poll (tPROCESS * ptProcess)

This function typically is called from an interrupt handler to cause a process to be polled.

Parameters

[in, out] **ptProcess** A pointer to the process' process structure.

Return values

TRUE poll successfully.

FALSE if the process is not running.

Here is the call graph for this function:

process_poll ____ process_is_running

Here is the caller graph for this function:

process_poll < ____ etimer_request_poll

BOOL process_post (tPROCESS *	ptProcess,
process_event_t	ev,
void *	data

)

This function posts an asynchronous event to one or all processes. The handing of the event is deferred until the target process is scheduled by the kernel. An event can be broadcast to all processes, in which case all processes in the system will be scheduled to handle the event.

Parameters

[in,out] ptProcess	The process to which the event should be
-	posted, or PROCESS_BROADCAST if the
	event should be posted to all processes.
[in] ev	The event to be posted.
[in,out] <mark>data</mark>	The auxiliary data to be sent with the event

Return values

TRUE The event could be posted.

FALSE The event queue was full and the event could not be posted.

See also

2 3

process_send()

U16 process_run (void)

This function should be called repeatedly from the main() program to actually run the system. It calls the necessary poll handlers, and processes one event. The function returns the number of events that are waiting in the event queue so that the caller may choose to put the CPU to sleep when there are no pending events.

Example about how to use protothread process model for event-driven system:

```
1 int main(void) {
```

- RBS_Init();//RBS initialization
- process_start();//start process

```
4 ...
5 while (1) {
6 do {
7 ...//do some optional user code
8 } while(process_run() > 0);
9 sleep();//go to sleep
10 }
11 }
```

Returns

The number of events that are currently waiting to be processed.

void process_send (tPROCESS *	ptProcess,
process_event_t	ev,
void *	data
)	

This function sends a synchronous event to one processes. On opposite of **process_post()**, the handing of the event is completed immediately after calling.

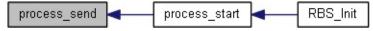
Parameters

[in, out] ptProcess A pointer to the process' process structure.		
[in]	ev	The event to be posted.
[in,out]	data	A pointer to additional data that is posted together with the event.

See also

process_post()

Here is the caller graph for this function:



BOOL process_start (tPROCESS * ptProcess, process_data_t arg

)	
Parameters [in,out]pt [in] ar	g	A pointer to a process structure. An argument pointer that can be passed to the new process

Return values

TRUE successful.

FALSE Try to start a process that is already running.

Here is the call graph for this function:

 process_start
 process_send

 Here is the caller graph for this function:

 process_start
 RBS_Init

Generated by



Public Member Functions | Data Fields

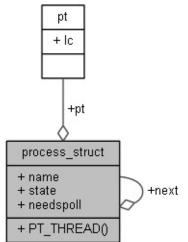
process_struct Struct Reference

protothread process

Process structure(one process consists one protothread). More...

#include <RainbowBS.h>

Collaboration diagram for process_struct:



Public Member Functions

PT_THREAD ((*thread)(struct pt *, process_event_t, process_data_t))

Data Fields

struct process_struct *	next
const char *	name
struct pt	pt
U8	state
BOOL	needspoll

Detailed Description

Note

application code should not change any member of the process structure for they are maintained by system inside.

Field Documentation

const char* name

process name string

BOOL needspoll

indicate if a process has high privilege to be called,

See also process_poll()

struct process_struct* next

pointer to the next process, all processes are in a process list

U8 state

indicate process state

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

• RainbowBS.h

Generated by



Event timers

Data Structures | Typedefs

Data Structures

struct etime_struct

Event timer structure. More...

Typedefs

typedef struct etime_struct tETIME

Event timer structure. More...

etimer functions called from application programs

- void etimer_set (tETIME *ptEtime, TICK interval) Set an event timer. More...
- void etimer_reset (tETIME *ptEtime) Reset an event timer with the same interval as was previously set. More...
- void etimer_restart (tETIME *ptEtime) Restart an event timer from the current point in time. More...
- void etimer_adjust (tETIME *ptEtime, int timediff) Adjust the expiration time for an event timer. More...
- TICK etimer_expiration_time (tETIME *ptEtime) Get the expiration time for the event timer. More...
- TICK etimer_start_time (tETIME *ptEtime) Get the start time for the event timer. More...
- BOOL etimer_expired (tETIME *ptEtime) Check if an event timer has expired. More...
 - void etimer_stop (tETIME *ptEtime) Stop a pending event timer. More...

etimer functions called from timer interrupts, by the system

void etimer_request_poll (void) Make the event timer aware that the clock has changed. More...

BOOL etimer_pending (void)

Check if there are any non-expired event timers. More...

Detailed Description

Event timers provides a way to generate timed events. An event timer will post an event to the process that set the timer when the event timer expires.

An event timer is declared as a struct etimer and all access to the event timer is made by a pointer to the declared event timer.

Typedef Documentation

typedef struct etime_struct tETIME

This structure is used for declaring a timer. The timer must be set with **etimer_set()** before it can be used.

Function Documentation

void etimer_adjust (tETIME * ptEtime, int timediff)

This function is used to adjust the time the event timer will expire. It can be used to synchronize periodic timers without the need to restart the timer or change the timer interval.

Parameters

[in, out] **ptEtime** A pointer to the event timer.

[in] **timediff** The time difference to adjust the expiration time with.

Note

This function should only be used for small adjustments. For large adjustments use **etimer_set()** instead.

A periodic timer will drift unless the **etimer_reset()** function is used.

See also

etimer_set()
etimer_reset()

TICK etimer_expiration_time (tETIME * ptEtime)

This function returns the expiration time for an event timer.

Parameters

[in, out] **ptEtime** A pointer to the event timer.

Returns

The expiration time for the event timer.

BOOL etimer_expired (tETIME * ptEtime)

This function tests if an event timer has expired and returns true or false depending on its status.

Parameters

[in] **ptEtime** A pointer to the event timer.

Return values

TRUE if the timer has expired. **FALSE** if the timer has not expired.

BOOL etimer_pending (void)

This function checks if there are any active event timers that have not expired.

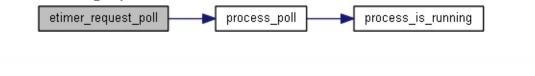
Return values

TRUE if there are active event timers. **FALSE** if there are no active timers.

void etimer_request_poll (void)

This function is used to inform the event timer module that the system clock has been updated. Typically, this function would be called from the timer interrupt handler when the clock has ticked.

Here is the call graph for this function:



void etimer_reset (tETIME * ptEtime)

This function resets the event timer with the same interval that was given to the event timer with the **etimer_set()** function. The start point of the interval is the exact time that the event timer last expired. Therefore, this function will cause the timer to be stable over time, unlike the **etimer_restart()** function.

Parameters

[in, out] **ptEtime** A pointer to the event timer.

See also

etimer_restart()

void etimer_restart (tETIME * ptEtime)

This function restarts the event timer with the same interval that was given to the **etimer_set()** function. The event timer will start at the current time.

Parameters

[in, out] **ptEtime** A pointer to the event timer.

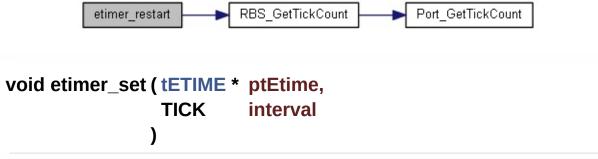
Note

A periodic timer will drift if this function is used to reset it. For periodic timers, use the **etimer_reset()** function instead.

See also

etimer_reset()

Here is the call graph for this function:



This function is used to set an event timer for a time sometime in the

future. When the event timer expires, the event PROCESS_EVENT_TIMER will be posted to the process that called the **etimer_set()** function.

Parameters

[in, out] **ptEtime** A pointer to the event timer

[in] **interval** The interval before the timer expires.

Here is the call graph for this function:



TICK etimer_start_time (tETIME * ptEtime)

This function returns the start time (when the timer was last set) for an event timer.

Parameters

[in, out] **ptEtime** A pointer to the event timer

Returns

The start time for the event timer.

void etimer_stop (tETIME * ptEtime)

This function stops an event timer that has previously been set with **etimer_set()** or **etimer_reset()**. After this function has been called, the event timer will not emit any event when it expires.

Parameters

[in, out] **ptEtime** A pointer to the pending event timer.

Generated by



Data Fields

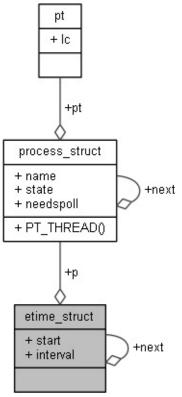
etime_struct Struct Reference

Event timers

Event timer structure. More...

#include <RainbowBS.h>

Collaboration diagram for etime_struct:



Data Fields

TICK start

TICK interval

struct etime_struct * next

tPROCESS * p

Detailed Description

This structure is used for declaring a timer. The timer must be set with **etimer_set()** before it can be used.

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

• RainbowBS.h

Generated by doxygen 1.8.9.1



debug

statement macros

statement macros for debug

- #define RBS_DEBUG_STA(sta) sta C statement sta is compiled if enable debug. More...
- #define **RBS_DEBUG_IF**(exp, sta) if (exp) { sta C statement if is compiled if enable debug. More...
- #define RBS_DEBUG_ELSIF(sta1, exp, sta2) sta1;} else if (exp) {
 sta2
 C statement else if is compiled if enable debug. More...
- #define RBS_DEBUG_ENDIF(sta) sta;}
 C statement } for if is compiled if enable debug. More...

error macro function

error macros for debug output

#define **RBS_DEBUG_ERROR**(exp, s, sta) Output error information. More...

#define RBS_DEBUG_ERROR_FORMAT(exp, format, sta, ...) Output format error information conditionally. More... warning macro function

warning macros for debug output

#define RBS_DEBUG_WARN(exp, s) Output warning information conditionally. More...

#define RBS_DEBUG_WARN_FORMAT(exp, format, ...) Output format warning information conditionally. More...

log macro function

log macros for debug output

#define RBS_DEBUG_LOG(s) Output format log information. More...

#define RBS_DEBUG_LOG_FORMAT(format, ...) Output format log information. More...

Detailed Description

used for system or application debug.

Macro Definition Documentation

This macro equals to sta1;} else if (exp) { sta2

Parameters

sta1 C statement.exp C expression.sta2 C statement.

Note

this should used with **RBS_DEBUG_IF()** and **RBS_DEBUG_ENDIF()**.

See also

RBS_DEBUG_IF() RBS_DEBUG_ENDIF()

#define RBS_DEBUG_ENDIF (sta) sta;}

This macro equals to sta1; }

Parameters

sta C statement.

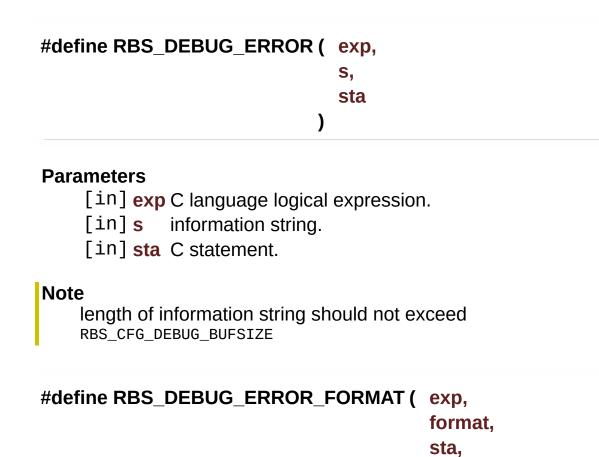
Note

this should used with **RBS_DEBUG_IF()** and **RBS_DEBUG_ELSIF()**.

See also

RBS_DEBUG_IF()

RBS_DEBUG_ELSIF()



Parameters

[in] **exp** C language logical expression.

. . .

)

[in] **format** format information string.

- [in] **sta** C statement.
- [in] ... parameters.

Note

length of information string should not exceed RBS_CFG_DEBUG_BUFSIZE

#define RBS_DEBUG_IF (exp,

This macro equals to if (exp) { sta

Parameters

exp C expression. sta C statement.

Note

this should used with **RBS_DEBUG_ELSIF()** and **RBS_DEBUG_ENDIF()**.

)

See also

RBS_DEBUG_ELSIF() RBS_DEBUG_ENDIF()

#define RBS_DEBUG_LOG (s)

Parameters

[in] **s** information string.

Note

length of information string should not exceed RBS_CFG_DEBUG_BUFSIZE

#define RBS_DEBUG_LOG_FORMAT (format,

Parameters

[in] format format information string. [in] ... parameters.

Note

length of information string should not exceed

#define RBS_DEBUG_STA (sta) sta

Parameters

sta C statement

#define RBS_DEBUG_WARN (exp,

)

S

Parameters

[in] exp C language logical expression.
[in] s information string.

Note

length of information string should not exceed RBS_CFG_DEBUG_BUFSIZE

#define RBS_DEBUG_WARN_FORMAT (exp,

format,

)

Parameters

- [in] **exp** C language logical expression.
- [in] **format** format information string.
- [in] ... parameters.

Note

length of information string should not exceed RBS_CFG_DEBUG_BUFSIZE

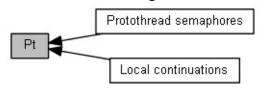
Generated by



Modules | Files | Data Structures | Macros

Collaboration diagram for Pt:

Pt



Modules

Local continuations

Protothread semaphores

Files

file pt.h

Data Structures

struct pt

Macros

#define **PT_WAITING** 0

#define **PT_YIELDED** 1

#define PT_EXITED 2

#define PT_ENDED 3

Initialization

#define PT_INIT(pt)

Declaration and definition

#define PT_THREAD(name_args)

#define **PT_BEGIN(pt)**

#define PT_END(pt)

Blocked wait

#define PT_WAIT_UNTIL(pt, condition)

#define PT_WAIT_WHILE(pt, cond)

Hierarchical protothreads

#define **PT_WAIT_THREAD(pt**, thread)

#define PT_SPAWN(pt, child, thread)

Exiting and restarting

#define PT_RESTART(pt)

#define **PT_EXIT(pt)**

Calling a protothread

#define **PT_SCHEDULE(f)**

Yielding from a protothread

#define PT_YIELD(pt)

#define PT_YIELD_UNTIL(pt, cond) Yield from the protothread until a condition occurs. More...

Detailed Description

Macro Definition Documentation

#define PT_BEGIN (pt)

Declare the start of a protothread inside the C function implementing the protothread.

This macro is used to declare the starting point of a protothread. It should be placed at the start of the function in which the protothread runs. All C statements above the **PT_BEGIN()** invokation will be executed each time the protothread is scheduled.

Parameters

pt A pointer to the protothread control structure.

#define PT_END (pt)

Declare the end of a protothread.

This macro is used for declaring that a protothread ends. It must always be used together with a matching **PT_BEGIN()** macro.

Parameters

pt A pointer to the protothread control structure.

#define PT_EXIT(pt)

Exit the protothread.

This macro causes the protothread to exit. If the protothread was spawned by another protothread, the parent protothread will become unblocked and can continue to run.

Parameters

pt A pointer to the protothread control structure.

#define PT_INIT (pt)

Initialize a protothread.

Initializes a protothread. Initialization must be done prior to starting to execute the protothread.

Parameters

pt A pointer to the protothread control structure.

See also PT SPAWN()

#define PT RESTART (pt)

Restart the protothread.

This macro will block and cause the running protothread to restart its execution at the place of the **PT_BEGIN()** call.

Parameters

pt A pointer to the protothread control structure.

#define PT_SCHEDULE(f)

Schedule a protothread.

This function shedules a protothread. The return value of the function is non-zero if the protothread is running or zero if the protothread has

exited.

Parameters

f The call to the C function implementing the protothread to be scheduled

```
#define PT_SPAWN ( pt,
child,
thread
)
```

Spawn a child protothread and wait until it exits.

This macro spawns a child protothread and waits until it exits. The macro can only be used within a protothread.

Parameters

pt A pointer to the protothread control structure.child A pointer to the child protothread's control structure.thread The child protothread with arguments

#define PT_THREAD (name_args)

Declaration of a protothread.

This macro is used to declare a protothread. All protothreads must be declared with this macro.

Parameters

name_args The name and arguments of the C function implementing the protothread.

```
#define PT_WAIT_THREAD ( pt,
thread
)
```

Block and wait until a child protothread completes.

This macro schedules a child protothread. The current protothread will block until the child protothread completes.

Note

The child protothread must be manually initialized with the **PT_INIT()** function before this function is used.

Parameters

pt A pointer to the protothread control structure.**thread** The child protothread with arguments

See also

PT_SPAWN()

#define PT_WAIT_UNTIL (pt,

condition

)

Block and wait until condition is true.

This macro blocks the protothread until the specified condition is true.

Parameters

pt A pointer to the protothread control structure. **condition** The condition.

```
#define PT_WAIT_WHILE ( pt,
cond
)
```

Block and wait while condition is true.

This function blocks and waits while condition is true. See

PT_WAIT_UNTIL().

Parameters

pt A pointer to the protothread control structure. **cond** The condition.

#define PT_YIELD (pt)

Yield from the current protothread.

This function will yield the protothread, thereby allowing other processing to take place in the system.

Parameters

pt A pointer to the protothread control structure.

)

#define PT_YIELD_UNTIL (pt,

cond

Parameters

pt A pointer to the protothread control structure.

cond The condition.

This function will yield the protothr specified condition evaluates to true



Files | Macros | Typedefs

Local continuations

Pt

Collaboration diagram for Local continuations:

Pt Local continuations

Files

file	lc-addrlabels.h
file	lc-switch.h
file	lc.h

Macros

- #define **LC_INIT**(s) s = NULL
- #define LC_RESUME(s)

#define LC_CONCAT2(s1, s2) s1##s2

#define LC_CONCAT(s1, s2) LC_CONCAT2(s1, s2)

#define LC_SET(s)

#define **LC_END**(s)

#define $LC_INIT(s) s = 0;$

#define **LC_RESUME**(s) switch(s) { case 0:

#define **LC_SET**(s) s = __LINE__; case __LINE__:

#define LC_END(s) }

#define __LC_H__

Typedefs

typedef void * lc_t

typedef unsigned short lc_t

Detailed Description

Local continuations form the basis for implementing protothreads. A local continuation can be *set* in a specific function to capture the state of the function. After a local continuation has been set can be *resumed* in order to restore the state of the function at the point where the local continuation was set.

Macro Definition Documentation

```
#define LC_RESUME( s)
```

Value:

#define LC_SET (s)

Value:

 \setminus



RainbowBS RBSSource

Base Protothreads

Macros | Typedefs

lc-addrlabels.h File Reference

Pt » Local continuations

Macros

- #define LC_INIT(s) s = NULL
- #define LC_RESUME(s)

#define LC_CONCAT2(s1, s2) s1##s2

#define LC_CONCAT(s1, s2) LC_CONCAT2(s1, s2)

#define LC_SET(s)

#define **LC_END**(s)

Typedefs

typedef void * lc_t

Detailed Description

Implementation of local continuations based on the "Labels as values" feature of gcc

Author

Adam Dunkels adam@sics.se

This implementation of local continuations is based on a special feature of the GCC C compiler called "labels as values". This feature allows assigning pointers with the address of the code corresponding to a particular C label.

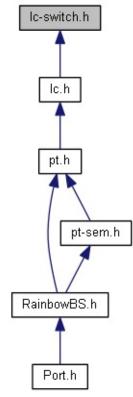
For more information, see the GCC documentation: http://gcc.gnu.org/onlinedocs/gcc/Labels-as-Values.html

RainbowBS Manual v0.2 Written by QWQ(jacobqwq@icloud.com)	L.O					
RainbowBS RBSSource Base Protothreads						
lc-switch h Eile	Macros Typedefs					

Ic-switch.h File Reference

Pt » Local continuations

This graph shows which files directly or indirectly include this file:



Macros

#define $LC_INIT(s) s = 0;$

#define LC_RESUME(s) switch(s) { case 0:

#define LC_SET(s) s = __LINE__; case __LINE__:

#define LC_END(s) }

Typedefs

typedef unsigned short lc_t

Detailed Description

Implementation of local continuations based on switch() statment

Author

Adam Dunkels adam@sics.se

This implementation of local continuations uses the C switch() statement to resume execution of a function somewhere inside the function's body. The implementation is based on the fact that switch() statements are able to jump directly into the bodies of control structures such as if() or while() statmenets.

This implementation borrows heavily from Simon Tatham's coroutines implementation in C:

http://www.chiark.greenend.org.uk/~sgtatham/coroutines.html

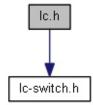
	RainbowBS Manual v0.1.0	
	Written by QWQ(jacobqwq@icloud.com)	
Rainbov	$ BS\rangle$ RBSSource \rangle Base \rangle Protothreads \rangle	

Ic.h File Reference

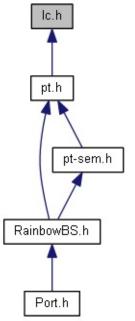
Pt » Local continuations

#include "lc-switch.h"

Include dependency graph for lc.h:



This graph shows which files directly or indirectly include this file:



Macros

Detailed Description

Local continuations

Author

Adam Dunkels adam@sics.se

Generated by



Files | Data Structures | Macros

Protothread semaphores

Pt

Collaboration diagram for Protothread semaphores:

Files

file pt-sem.h

Data Structures

struct pt_sem

Macros

#define PT_SEM_INIT(s, c)

#define PT_SEM_WAIT(pt, s)

#define PT_SEM_SIGNAL(pt, s)

Detailed Description

This module implements counting semaphores on top of protothreads. Semaphores are a synchronization primitive that provide two operations: "wait" and "signal". The "wait" operation checks the semaphore counter and blocks the thread if the counter is zero. The "signal" operation increases the semaphore counter but does not block. If another thread has blocked waiting for the semaphore that is signalled, the blocked thread will become runnable again.

Semaphores can be used to implement other, more structured, synchronization primitives such as monitors and message queues/bounded buffers (see below).

The following example shows how the producer-consumer problem, also known as the bounded buffer problem, can be solved using protothreads and semaphores. Notes on the program follow after the example.

```
#include "pt-sem.h"
#define NUM_ITEMS 32
#define BUFSIZE 8
static struct pt_sem mutex, full, empty;
PT_THREAD(producer(struct pt *pt))
{
    static int produced;
    PT_BEGIN(pt);
    for(produced = 0; produced < NUM_ITEMS; ++produced)
        {
        PT_SEM_WAIT(pt, &full);
        PT_SEM_WAIT(pt, &mutex);
        add_to_buffer(produce_item());
        PT_SEM_SIGNAL(pt, &mutex);
    }
}
</pre>
```

```
PT_SEM_SIGNAL(pt, &empty);
 }
PT_END(pt);
}
PT_THREAD(consumer(struct pt *pt))
{
static int consumed;
PT_BEGIN(pt);
for(consumed = 0; consumed < NUM_ITEMS; ++consumed)</pre>
    {
PT_SEM_WAIT(pt, &empty);
PT_SEM_WAIT(pt, &mutex);
    consume_item(get_from_buffer());
PT_SEM_SIGNAL(pt, &mutex);
PT_SEM_SIGNAL(pt, &full);
 }
PT_END(pt);
}
PT_THREAD(driver_thread(struct pt *pt))
{
static struct pt pt_producer, pt_consumer;
PT_BEGIN(pt);
PT_SEM_INIT(&empty, 0);
PT_SEM_INIT(&full, BUFSIZE);
PT_SEM_INIT(&mutex, 1);
PT_INIT(&pt_producer);
PT_INIT(&pt_consumer);
PT_WAIT_THREAD(pt, producer(&pt_producer) &
         consumer(&pt_consumer));
```

```
PT_END(pt);
}
```

The program uses three protothreads: one protothread that implements the consumer, one thread that implements the producer, and one protothread that drives the two other protothreads. The program uses three semaphores: "full", "empty" and "mutex". The "mutex" semaphore is used to provide mutual exclusion for the buffer, the "empty" semaphore is used to block the consumer is the buffer is empty, and the "full" semaphore is used to block the producer is the buffer is full.

The "driver_thread" holds two protothread state variables, "pt_producer" and "pt_consumer". It is important to note that both these variables are declared as *static*. If the static keyword is not used, both variables are stored on the stack. Since protothreads do not store the stack, these variables may be overwritten during a protothread wait operation. Similarly, both the "consumer" and "producer" protothreads declare their local variables as static, to avoid them being stored on the stack.

Macro Definition Documentation

)

```
#define PT_SEM_INIT ( s, c
```

Initialize a semaphore

This macro initializes a semaphore with a value for the counter. Internally, the semaphores use an "unsigned int" to represent the counter, and therefore the "count" argument should be within range of an unsigned int.

Parameters

- s (struct pt_sem *) A pointer to the pt_sem struct representing the semaphore
- c (unsigned int) The initial count of the semaphore.

#define PT_SEM_SIGNAL (pt,
	S
)	

Signal a semaphore

This macro carries out the "signal" operation on the semaphore. The signal operation increments the counter inside the semaphore, which eventually will cause waiting protothreads to continue executing.

Parameters

- pt (struct pt *) A pointer to the protothread (struct pt) in which the operation is executed.
- s (struct pt_sem *) A pointer to the pt_sem struct representing
 the semaphore

#define PT_SEM_WAIT(pt, s)

Wait for a semaphore

This macro carries out the "wait" operation on the semaphore. The wait operation causes the protothread to block while the counter is zero. When the counter reaches a value larger than zero, the protothread will continue.

Parameters

- pt (struct pt *) A pointer to the protothread (struct pt) in which the operation is executed.
- s (struct pt_sem *) A pointer to the pt_sem struct representing the semaphore

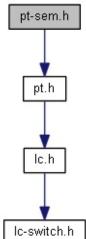
	Rainbo	wB3	S Manual vo.1.0	
	Written by Q	NQ(ja	cobqwq@icloud.com)	
RainbowBS ABSSource Base Protothreads				

pt-sem.h File Reference

Pt » Protothread semaphores

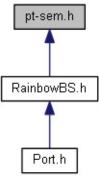
#include "pt.h"

Include dependency graph for pt-sem.h:



Data Structures | Macros

This graph shows which files directly or indirectly include this file:



Data Structures

struct pt_sem

Macros

#define PT_SEM_INIT(s, c)

#define PT_SEM_WAIT(pt, s)

#define PT_SEM_SIGNAL(pt, s)

Detailed Description

Couting semaphores implemented on protothreads

Author

Adam Dunkels adam@sics.se



Data Fields

pt_sem Struct Reference

Pt » Protothread semaphores

Collaboration diagram for pt_sem:



Data Fields

unsigned int count

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

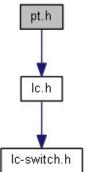
• pt-sem.h

	RainbowBS Manual	
	Written by QWQ(jacobqwq@icloud.	com)
Rainbow	BS RBSSource Base Protothreads	
		Data Structures Macros

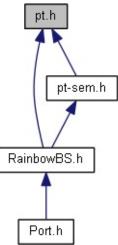
pt.h File Reference

#include "lc.h"

Include dependency graph for pt.h:



This graph shows which files directly or indirectly include this file:



Data Structures

struct pt

Macros

#define **PT_WAITING** 0

#define **PT_YIELDED** 1

#define **PT_EXITED** 2

#define **PT_ENDED** 3

Initialization

#define PT_INIT(pt)

Declaration and definition

#define PT_THREAD(name_args)

#define PT_BEGIN(pt)

#define PT_END(pt)

Blocked wait

#define PT_WAIT_UNTIL(pt, condition)

#define PT_WAIT_WHILE(pt, cond)

Hierarchical protothreads #define PT_WAIT_THREAD(pt, thread)

#define **PT_SPAWN(pt**, child, thread)

Exiting and restarting
#define PT_RESTART(pt)

#define PT_EXIT(pt)

Calling a protothread #define PT_SCHEDULE(f)

Yielding from a protothread #define PT_YIELD(pt)

#define PT_YIELD_UNTIL(pt, cond) Yield from the protothread until a condition occurs. More...

Detailed Description

Protothreads implementation.

Author

Adam Dunkels adam@sics.se

Generated by



pt Struct Reference

Collaboration diagram for pt:



lc_t lc

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

• pt.h

Generated by



Data Structures

Here are the data structures with brief descriptions:

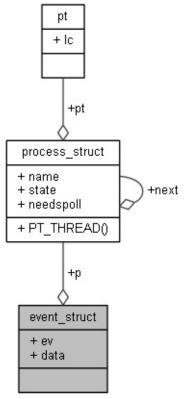
Cetime_struct	Event timer structure
<pre>event_struct</pre>	
MEM_struct	
<pre>process_struct</pre>	Process structure(one process consists one protothread)
la pt	
pt_sem	
tBLOCK	
C tDMMHEAD	
la tTime	Time structure

Generated by



event_struct Struct Reference

Collaboration diagram for event_struct:



process_event_t ev

process_data_t data

tPROCESS * p

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

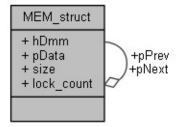
• process.c

Generated by doxygen 1.8.9.1



MEM_struct Struct Reference

Collaboration diagram for MEM_struct:



hDMM	hDmm
U8 *	pData
USIZE	size
struct MEM_struct *	pNext
struct MEM_struct *	pPrev
U8	lock_count

Field Documentation

hDMM hDmm

DMM handle

U8 lock_count

0:not allocated,1:allocated,[2,255]:being used lock_count-1 times.

U8* pData

pointer of memory area

struct MEM_struct* pNext

next handle in linked list

struct MEM_struct* pPrev

prev handle in linked list

USIZE size

usable size of allocated block

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

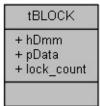
• DMM.c

Generated by



tBLOCK Struct Reference

Collaboration diagram for tBLOCK:



hDMM hDmm

U8 * pData

U8 lock_count

Field Documentation

hDMM hDmm

DMM handle

U8 lock_count

0:not allocated,1:allocated,[2,255]:being used lock_count-1 times

U8* pData

pointer of data area

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

• DMM.c

Generated by doxyden 1.8.9.1



tDMMHEAD Struct Reference

Data Fields

Collaboration diagram for tDMMHEAD:

eDMTYPE eType

char * pName

void * pObjectArrayStart

void * pObjectArrayLast

void * **pFreeObject**

USIZE free_object_count

USIZE free_bytes

U8 * pDataStart

U8 * pDataEnd

USIZE **block_size**

HMUTEX hMutex

Field Documentation

USIZE block_size

block size if eType is DM_BLOCK

eDMTYPE eType

DMM type

USIZE free_bytes

count of free bytes if eType is DM_POOL or DM_POOL_AUTO

USIZE free_object_count

count of free array(tBLOCK or tMEM) objects

HMUTEX hMutex

lock

U8* pDataEnd

pointer of data end

U8* pDataStart

pointer of data head

void* pFreeObject

pointer for accelerating allocation

char* pName

DMM name

void* pObjectArrayLast

pointer of the last array(tBLOCK or tMEM) object

void* pObjectArrayStart

pointer of the first array(tBLOCK or tMEM) object

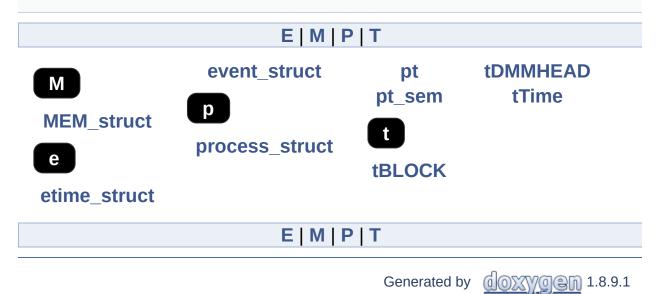
The documentation for this struct was generated from the following file:

• DMM.c

Generated by doxyden 1.8.9.1



Data Structure Index





Here is a list of all documented struct and union fields with links to the struct/union documentation for each field:

- b -

- block_size : tDMMHEAD
- d -
 - day : tTime
 - day_of_week : tTime

- e -

- eType : tDMMHEAD
- f -
 - free_bytes : **tDMMHEAD**
 - free_object_count : tDMMHEAD

- h -

- hDmm : MEM_struct , tBLOCK
- hMutex : tDMMHEAD
- hour : tTime

- | -

• lock_count : MEM_struct , tBLOCK

- m -

• milliseconds : tTime

- minute : tTime
- month : tTime

- n -

- name : process_struct
- needspoll : process_struct
- next : process_struct

- p -

- pData : MEM_struct , tBLOCK
- pDataEnd : tDMMHEAD
- pDataStart : tDMMHEAD
- pFreeObject : tDMMHEAD
- pName : tDMMHEAD
- pNext : MEM_struct
- pObjectArrayLast : tDMMHEAD
- pObjectArrayStart : tDMMHEAD
- pPrev : MEM_struct

- S -

- second : tTime
- size : MEM_struct
- state : process_struct

-у-

• year : tTime





- b -

- block_size : tDMMHEAD
- d -
 - day : tTime
 - day_of_week : tTime

- e -

- eType : tDMMHEAD
- f -
 - free bytes : tDMMHEAD
 - free_object_count : tDMMHEAD

- h -

- hDmm : MEM_struct , tBLOCK
- hMutex : tDMMHEAD
- hour : tTime

- | -

• lock_count : MEM_struct , tBLOCK

- m -

- milliseconds : tTime
- minute : tTime

• month : tTime

- n -

- name : process_struct
- needspoll : process_struct
- next : process_struct

- p -

- pData : MEM_struct , tBLOCK
- pDataEnd : tDMMHEAD
- pDataStart : tDMMHEAD
- pFreeObject : tDMMHEAD
- pName : tDMMHEAD
- pNext : MEM_struct
- pObjectArrayLast : tDMMHEAD
- pObjectArrayStart : tDMMHEAD
- pPrev : MEM_struct

- S -

- second : tTime
- size : MEM_struct
- state : process_struct

-у-

• year : tTime

Generated by



File List

Here is a list of all documented files with brief descriptions:

🖹 Ic-addrlabels.h	
Ic-switch.h	
lc.h	
Port.h	
🗎 pt-sem.h	
🖹 pt.h	
🖹 RainbowBS.h	
RainbowBSConf.h	

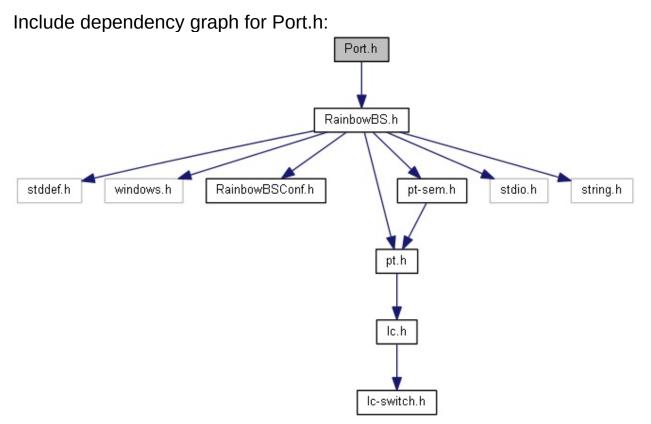
Generated by doxyden 1.8.9.1



Port.h File Reference

Functions

#include "RainbowBS.h"



Functions

BOOL Port_Init (void) Initialize hardware. More...

TICK Port_GetTickCount (void) Get system tick. More...

void Port_GetLocalTime (tTime *pTime) Get system local time. More...

void **Port_Delay (U32 millisec)** Delay some millisecond. More...

void **Port_Printf_Error** (const char *s) Output error information. More...

- void **Port_Printf_Warn** (const char *s) Output warning information. More...
- void **Port_Printf_Log** (const char *s) Output log information. More...
- BOOL **Port_GetMutex (HMUTEX hMutex)** Wait for a mutex. More...
- BOOL **Port_FreeMutex (HMUTEX hMutex)** Release a mutex. More...

Detailed Description

Implementation of porting.

Author

QWQ jacobqwq@icloud.com

Function Documentation

void Port_Delay (U32 millisec)

Parameters

[in] millisec millisecond count.

Here is the caller graph for this function:

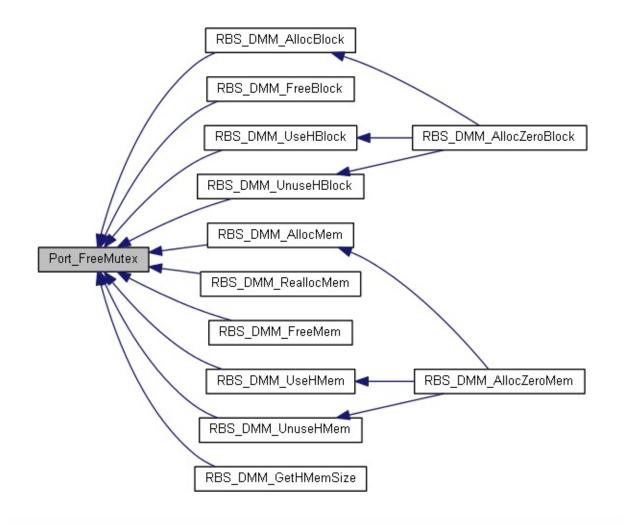


BOOL Port_FreeMutex (HMUTEX hMutex)

Parameters

[in] **hMutex** handle of mutex.

Here is the caller graph for this function:



void Port_GetLocalTime (tTime * pTime)

Parameters

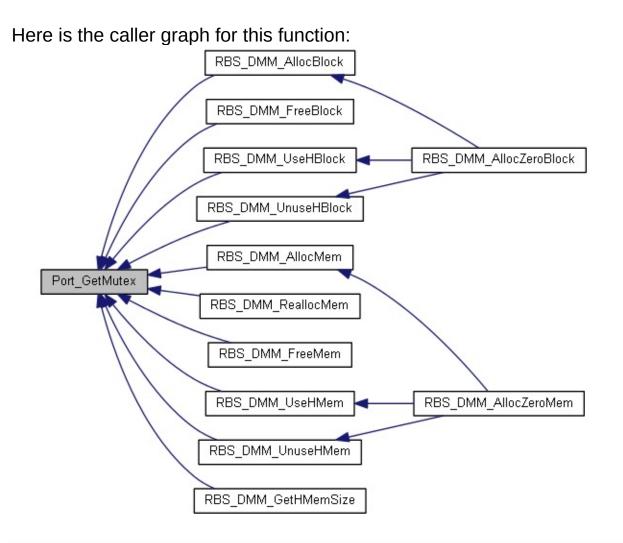
[out] **pTime** time structure.

Here is the caller graph for this function:

BOOL Port_GetMutex (HMUTEX hMutex)

Parameters

[in] **hMutex** handle of mutex.

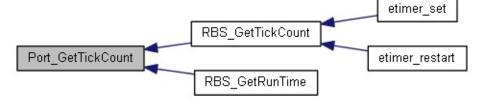


TICK Port_GetTickCount (void)

Returns

system tick.

Here is the caller graph for this function:



BOOL Port_Init (void)

Return values

TRUE successful. **FALSE** failed.

Here is the caller graph for this function:



void Port_Printf_Error (const char * s)

Parameters

[in] s string.

void Port_Printf_Log (const char * s)

Parameters

[in] **s** string.

void Port_Printf_Warn (const char * s)

Parameters

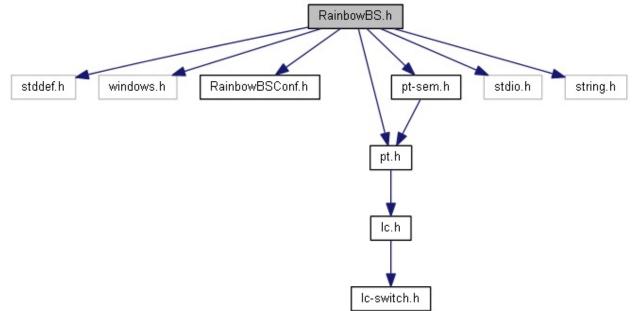
[in] s string.

Generated by doxyden 1.8.9.1

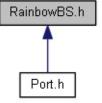


#include	<stddef.h> #include</stddef.h>	<windows.h></windows.h>
#include	"RainbowBSConf.h"	
#include	"pt.h"	
#include	"pt-sem.h"	
#include	<stdio.h></stdio.h>	
#include	<string.h></string.h>	

Include dependency graph for RainbowBS.h:



This graph shows which files directly or indirectly include this file:



Data Structures

struct tTime Time structure. More...

struct process_struct Process structure(one process consists one protothread). More...

struct etime_struct Event timer structure. More...

Macros

- #define **RBS_VERSION** "0.1.0"
- #define **RBS_VERNUM** 0x0100
- #define RBS_VER_MAJOR 0
- #define **RBS_VER_MINOR** 1
- #define RBS_VER_REVISION 0
- #define **RBS_VER_SUBREVISION** 0
- #define FALSE Ou
- #define TRUE 1u
- #define **USE_PARA**(para) (para = (para))
- #define MIN(v0, v1) (((v0) > (v1))? (v1): (v0))
- #define MAX(v0, v1) (((v0) > (v1))? (v0) : (v1))
- #define **ABS**(v) (((v) >= 0) ? (v) : (-(v)))
- #define **SWAP**(a, b) (a = (a) + (b), b = (a) (b), a = (a) (b))
- #define COUNT_OF(a) (sizeof(a)/sizeof(a[0]))
- #define _**STR**(a) #a
- #define STR(a) _STR(a)
- #define _CONS(a, b) a##b

#define CONS(a, b) _CONS(a,b)

#define ALIGN_F(pointer, power2) (((IPTRDIFF)(pointer) + ((IPTRDIFF)((power2) - 1))) & (~((IPTRDIFF)((power2) - 1))))

#define **ALIGN_B**(pointer, power2) ((IPTRDIFF)(pointer) & (~ ((IPTRDIFF)((power2) - 1))))

#define **HDMM_NULL** NULL

#define **HBLOCK_NULL** NULL

#define HMEM_NULL NULL

#define **PROCESS_NONE** NULL

#define PROCESS_BROADCAST NULL

#define **PROCESS_NAME_STRING**(process) ((NULL == (process)) ? "No Name" : (process)->name)

predefined event type
#define PROCESS EVENT NONE 0x80

#define **PROCESS EVENT INIT** 0x81

#define **PROCESS_EVENT_POLL** 0x82

#define **PROCESS_EVENT_EXIT** 0x83

#define **PROCESS_EVENT_SERVICE_REMOVED** 0x84

#define **PROCESS_EVENT_CONTINUE** 0x85

#define **PROCESS_EVENT_MSG** 0x86

#define **PROCESS_EVENT_EXITED** 0x87

#define **PROCESS_EVENT_TIMER** 0x88

#define **PROCESS_EVENT_COM** 0x89

#define **PROCESS_EVENT_MAX** 0x8a

process declaration and definition

#define **PROCESS_THREAD**(name, ev, data) Define the body of a process. More...

#define **PROCESS_NAME(name)** extern **tPROCESS** name Declare the name of a process. More...

#define **PROCESS**(name, strname) Define a process. More...

semaphore declaration and definition

#define SEM_NAME(name) extern struct pt_sem sem_##name Declare the name of a semaphore. More...

#define SEM(name, count) Define a semaphore. More...

process protothread functions

#define PROCESS_BEGIN() Define the beginning of a process. More...

#define PROCESS_END() Define the end of a process. More...

#define PROCESS_WAIT_EVENT() Wait for an event to be posted to the process. More...

#define	PROCESS_WAIT_EVENT_UNTIL(c) Wait for an event to be posted to the process, with an extra condition. More	
#define	PROCESS_YIELD() Yield the currently running process.	
#define	PROCESS_YIELD_UNTIL(c) Yield the currently running process until a condition occurs. More	
#define	PROCESS_WAIT_UNTIL(c) Wait for a condition to occur. More	
#define	PROCESS_WAIT_WHILE(c) Wait for a condition not occur. More	
#define	PROCESS_EXIT() Exit the currently running process.	
#define	PROCESS_PT_SPAWN(pt, thread) Spawn a protothread from the process. More	
#define	PROCESS_PAUSE() Yield the process for a short while. More	
#define	PROCESS_WAIT_SEM(name)	
#define	PROCESS_SIGNAL_SEM(name)	
poll and exit handlers		
Hdafina	PROCESS_POLLHANDLER(handler) if(ev ==	
#define	PROCESS_EVENT_POLL) { handler; }	

Specify an action when a process is polled. More...

#define **PROCESS_EXITHANDLER**(handler) if(ev ==

PROCESS_EVENT_EXIT) { handler; } Specify an action when a process exits. More...

statement macros

statement macros for debug

#define RBS_DEBUG_STA(sta) sta C statement sta is compiled if enable debug. More...

#define RBS_DEBUG_IF(exp, sta) if (exp) { sta C statement if is compiled if enable debug. More...

#define RBS_DEBUG_ELSIF(sta1, exp, sta2) sta1;} else if (exp) { sta2 C statement else if is compiled if enable debug. More...

#define RBS_DEBUG_ENDIF(sta) sta;}
C statement } for if is compiled if enable debug. More...

error macro function

error macros for debug output

#define **RBS_DEBUG_ERROR**(exp, s, sta) Output error information. More...

#define **RBS_DEBUG_ERROR_FORMAT(exp, format, sta, ...)** Output format error information conditionally. More...

warning macro function

warning macros for debug output

#define RBS_DEBUG_WARN(exp, s)

Output warning information conditionally. More...

#define RBS_DEBUG_WARN_FORMAT(exp, format, ...) Output format warning information conditionally. More...

log macro function

log macros for debug output

#define RBS_DEBUG_LOG(s) Output format log information. More...

#define RBS_DEBUG_LOG_FORMAT(format, ...) Output format log information. More...

Typedefs

typedef uint8_t	BOOL
typedef int8_t	18
typedef uint8_t	U8
typedef int16_t	116
typedef uint16_t	U16
typedef int32_t	132
typedef uint32_t	U32
typedef int64_t	164
typedef uint64_t	U64
typedef size_t	USIZE
typedef ptrdiff_t	IPTRDIFF
typedef U64	TICK
typedef void *	hDMM
typedef void *	hBLOCK
typedef void *	hMEM
typedef U8	process_event_t
typedef void *	process_data_t

typedef struct process_struct	tPROCESS Process structure(one process consists one protothread). More
typodof struct atime, struct	

typedef struct etime_struct tETIME Event timer structure. More...

Functions

- BOOL RBS_Init (void) Initialize RainbowBS. More...
 - U8 RBS_GetCPUBits (void) CPU word-width. More...
- BOOL **RBS_IsCPULittleEndian (void)** Check if CPU is little-endian. More...
- BOOL **RBS_IsStackGrowDown (void)** Check if stack grows down. More...
 - void **RBS_MemSet8** (U8 *pDes, U8 fill, USIZE num) Set memory by byte unit. More...
 - void **RBS_MemSet16** (U16 *pDes, U16 fill, USIZE num) Set memory by two-bytes unit. More...
 - void **RBS_MemSet32** (U32 *pDes, U32 fill, USIZE num) Set memory by four-bytes unit. More...
 - void **RBS_MemCpy8** (U8 *pDes, const U8 *pSrc, USIZE num) Copy memory by byte unit. More...
 - U16 RBS_Read16L (const U8 **ppData) Read a 16-bits entity in little-endian. More...
 - U32 **RBS_Read32L** (const U8 **ppData) Read a 32-bits entity in little-endian. More...
 - void **RBS_Write16L** (U8 **ppData, U16 data) Write a 16-bits entity in little-endian. More...

void **RBS_Write32L** (U8 **ppData, U32 data) Write a 32-bits entity in little-endian. More...

U16 **RBS_Read16B** (const U8 **ppData) Read a 16-bits entity in big-endian. More...

U32 **RBS_Read32B** (const U8 **ppData) Read a 32-bits entity in big-endian. More...

void **RBS_Write16B** (U8 **ppData, U16 data) Write a 16-bits entity in big-endian. More...

void **RBS_Write32B** (U8 **ppData, U32 data) Write a 32-bits entity in big-endian. More...

hDMM **RBS_DMM_RegisterBlock** (char *pName, void *pDM, USIZE size, USIZE block_size, HMUTEX hMutex) Register the memory area as a dynamic block memory area. More...

hBLOCK RBS_DMM_AllocZeroBlock (hDMM hDmm) Allocate a free block with zero initialization from the dynamic block memory area. More...

hBLOCK RBS_DMM_AllocBlock (hDMM hDmm) Allocate a free block from the dynamic block memory area. More...

BOOL **RBS_DMM_FreeBlock** (hBLOCK hBlock) Free the allocated block. More...

void * **RBS_DMM_UseHBlock** (hBLOCK hBlock) Get the allocated block pointer. More...

BOOL **RBS_DMM_UnuseHBlock** (hBLOCK hBlock) Unuse the allocated block area. More...

RBS DMM RegisterPool (char *pName, void *pDM, hDMM USIZE size, BOOL bAntiFrag, U16 handle count, HMUTEX hMutex) Register the memory area as a dynamic pool memory area. More... hMEM **RBS DMM AllocZeroMem** (hDMM hDmm, USIZE size) Allocate a free memory with zero initialization from the dynamic memory area. More... hMEM **RBS DMM AllocMem** (hDMM hDmm, USIZE size) Allocate a free memory from the dynamic memory area. More... hMEM **RBS DMM ReallocMem** (hMEM hMem, USIZE size) Reallocate a free memory from the same dynamic memory area. More... BOOL **RBS DMM FreeMem** (hMEM hMem) Free the allocated memory. More... void * RBS_DMM_UseHMem (hMEM hMem) Get the allocated memory pointer. More... BOOL **RBS DMM UnuseHMem** (hMEM hMem) Unuse the allocated memory area. More...

USIZE **RBS_DMM_GetHMemSize** (hMEM hMem) Get the allocated memory size. More...

system information

const char * **RBS_GetVersionString (void)** Get RainbowBS version string. More...

const char * **RBS_GetSysInfo** (void)

Get system description string. More...

process functions called from device drivers.

BOOL process_poll (tPROCESS *ptProcess) Request a process to be polled. More...

functions called by the system and boot-up code

- U16 process_run (void) Run the system once - call poll handlers and process one event. More...
- BOOL process_is_running (tPROCESS *ptProcess) Check if a process is running. More...

U16 process_nevents (void) Number of events waiting to be processed. More...

etimer functions called from application programs

- void etimer_set (tETIME *ptEtime, TICK interval) Set an event timer. More...
- void etimer_reset (tETIME *ptEtime) Reset an event timer with the same interval as was previously set. More...
- void etimer_restart (tETIME *ptEtime) Restart an event timer from the current point in time. More...
- void etimer_adjust (tETIME *ptEtime, int timediff) Adjust the expiration time for an event timer. More...
- TICK etimer_expiration_time (tETIME *ptEtime) Get the expiration time for the event timer. More...

TICK etimer_start_time (tETIME *ptEtime) Get the start time for the event timer. More...

BOOL etimer_expired (tETIME *ptEtime) Check if an event timer has expired. More...

void etimer_stop (tETIME *ptEtime) Stop a pending event timer. More...

etimer functions called from timer interrupts, by the system

void etimer_request_poll (void) Make the event timer aware that the clock has changed. More...

BOOL etimer_pending (void)

Check if there are any non-expired event timers. More...

time releated

#define RBS_TICK_MS (1000u/RBS_CFG_TICK_RATE)

- TICK **RBS_GetTickCount (void)** Get system tick. More...
 - U64 **RBS_GetRunTime (void)** Get system running time(ms). More...
- void **RBS_GetLocalTime (tTime *ptTime)** Get system local time. More...
- void **RBS_Delay** (U32 milliseconds) Delay some milliseconds. More...

convert digit to string

enum ePOW { SHEX, SBINARY, SDECIMAL } Specific hex, binary or decimal. More...

U8 RBS_Number2String (U32 value, ePOW ePow, U8 length, char *pText)

Convert unsigned digit to ASCII string. More...

trigonometric releted

#define ANG_45DEG 1024

#define ANG_90DEG (2*ANG_45DEG)

#define ANG_135DEG (3*ANG_45DEG)

#define **ANG_180DEG** (4*ANG 45DEG)

#define ANG_225DEG (5*ANG_45DEG)

#define ANG_270DEG (6*ANG_45DEG)

#define ANG_315DEG (7*ANG_45DEG)

#define ANG_360DEG (8*ANG_45DEG)

int **RBS_sin** (int angle) Calculate sine. More...

int **RBS_cos** (int angle) Calculate cosine. More...

int **RBS_tan (int angle)** Calculate tangent. More...

int **RBS_ctan** (int angle) Calculate cotangent. More...

process functions called from application programs

#define **PROCESS_CURRENT()** process_current Get a pointer to the currently running process. More...

#define *tmp_current = **PROCESS_CURRENT**();process_current = ptProcess Switch context to another process. More...

#define PROCESS_CONTEXT_END(ptProcess) process_cui
= tmp_current; }
End a context switch. More...

tPROCESS * process_current

BOOL process_start (tPROCESS *ptProcess, process_data arg) Start a process. More...

BOOL process_post (tPROCESS *ptProcess, process_even ev, void *data) Post an asynchronous event to one or all processes. More...

- void process_send (tPROCESS *ptProcess, process_ever ev, void *data) Send a synchronous event to a process. More...
- void process_exit (tPROCESS *ptProcess) Cause a process to exit. More...
- process_event_t process_alloc_event (void) Allocate a global event number. More...

Detailed Description

RainbowBS Interface.

Author

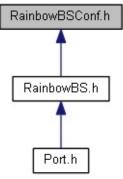
QWQ jacobqwq@icloud.com

Generated by



RainbowBSConf.h File Reference

This graph shows which files directly or indirectly include this file:



Macros

#define RBS_CFG_SYS_INFO "RainbowSys"

#define RBS_CFG_TICK_RATE 100

#define **RBS_CFG_CPU_WORD_SIZE** 32 /* CPU word size */

#define **RBS_CFG_CPU_BYTE_ORDER_L** 1 /* CPU byte order */

#define **RBS_CFG_CPU_STACK_DOWN** 1 /* stack growth direction */

#define **RBS_APP_NONE** 0

#define **RBS_APP_PTP** 1

#define **RBS_APP_OS** 2

#define **RBS_CFG_APP_MODEL** RBS_APP_PTP

#define **RBS_CFG_PTP_NO_PROCESS_NAME** 0

#define RBS_CFG_PTP_PROCESS_STATS 0

#define RBS_CFG_PTP_NUMEVENTS 32

#define RBS_CFG_DMM_ALIGN 2

#define **RBS_DEBUG_LEVEL_NOCHECK** 0 /* No running time checks are performed */

#define **RBS_DEBUG_LEVEL_ERRORS** 1 /* Errors are recorded */

#define **RBS_DEBUG_LEVEL_WARNINGS** 2 /* Errors and Warnings are recorded */

#define **RBS_DEBUG_LEVEL_LOG** 3 /* Errors,Warnings and logs are recorded */

#define RBS_CFG_DEBUG_BUFSIZE 300

#define **RBS_CFG_DEBUG_LEVEL** RBS_DEBUG_LEVEL_LOG

Detailed Description

RainbowBS Configuration.

Author

QWQ jacobqwq@icloud.com

Generated by



Here is a list of all documented functions, variables, defines, enums, and typedefs with links to the documentation:

- e -

- ePOW : RainbowBS.h
- etimer_adjust() : RainbowBS.h
- etimer_expiration_time() : RainbowBS.h
- etimer_expired() : RainbowBS.h
- etimer_pending() : RainbowBS.h
- etimer_request_poll() : RainbowBS.h
- etimer_reset() : RainbowBS.h
- etimer_restart() : RainbowBS.h
- etimer_set() : RainbowBS.h
- etimer_start_time() : RainbowBS.h
- etimer_stop() : RainbowBS.h

- p -

- Port_Delay() : Port.h
- Port_FreeMutex() : Port.h
- Port_GetLocalTime() : Port.h
- Port_GetMutex() : Port.h
- Port_GetTickCount() : Port.h
- Port_Init() : Port.h
- Port_Printf_Error() : Port.h
- Port_Printf_Log() : Port.h
- Port_Printf_Warn() : Port.h
- PROCESS : RainbowBS.h
- process_alloc_event() : RainbowBS.h
- PROCESS_BEGIN : RainbowBS.h
- PROCESS_CONTEXT_BEGIN : RainbowBS.h
- PROCESS_CONTEXT_END : RainbowBS.h
- PROCESS_CURRENT : RainbowBS.h
- PROCESS_END : RainbowBS.h
- process_exit() : RainbowBS.h

- PROCESS_EXIT : RainbowBS.h
- PROCESS_EXITHANDLER : RainbowBS.h
- process_is_running() : RainbowBS.h
- PROCESS_NAME : RainbowBS.h
- process_nevents() : RainbowBS.h
- PROCESS_PAUSE : RainbowBS.h
- process_poll() : RainbowBS.h
- PROCESS_POLLHANDLER : RainbowBS.h
- process_post() : RainbowBS.h
- PROCESS_PT_SPAWN : RainbowBS.h
- process_run() : RainbowBS.h
- process_send() : RainbowBS.h
- PROCESS_SIGNAL_SEM : RainbowBS.h
- process_start() : RainbowBS.h
- PROCESS_THREAD : RainbowBS.h
- PROCESS_WAIT_EVENT : RainbowBS.h
- PROCESS_WAIT_EVENT_UNTIL : RainbowBS.h
- PROCESS_WAIT_SEM : RainbowBS.h
- PROCESS_WAIT_UNTIL : RainbowBS.h
- PROCESS_WAIT_WHILE : RainbowBS.h
- PROCESS_YIELD : RainbowBS.h
- PROCESS_YIELD_UNTIL : RainbowBS.h
- PT_BEGIN : pt.h
- PT_END : pt.h
- PT_EXIT : pt.h
- PT_INIT : pt.h
- PT_RESTART : pt.h
- PT_SCHEDULE : pt.h
- PT_SEM_INIT : pt-sem.h
- PT_SEM_SIGNAL : pt-sem.h
- PT_SEM_WAIT : pt-sem.h
- PT_SPAWN : pt.h
- PT_THREAD : pt.h
- PT_WAIT_THREAD : pt.h
- PT_WAIT_UNTIL : pt.h
- PT_WAIT_WHILE : pt.h
- PT_YIELD : pt.h
- PT_YIELD_UNTIL : pt.h

- RBS_cos() : RainbowBS.h
- RBS_ctan() : RainbowBS.h
- RBS_DEBUG_ELSIF : RainbowBS.h
- RBS_DEBUG_ENDIF : RainbowBS.h
- RBS_DEBUG_ERROR : RainbowBS.h
- RBS_DEBUG_ERROR_FORMAT : RainbowBS.h
- RBS_DEBUG_IF : RainbowBS.h
- RBS_DEBUG_LOG : RainbowBS.h
- RBS_DEBUG_LOG_FORMAT : RainbowBS.h
- RBS_DEBUG_STA : RainbowBS.h
- RBS_DEBUG_WARN : RainbowBS.h
- RBS_DEBUG_WARN_FORMAT : RainbowBS.h
- RBS_Delay() : RainbowBS.h
- RBS_DMM_AllocBlock() : RainbowBS.h
- RBS_DMM_AllocMem() : RainbowBS.h
- RBS_DMM_AllocZeroBlock() : RainbowBS.h
- RBS_DMM_AllocZeroMem() : RainbowBS.h
- RBS_DMM_FreeBlock() : RainbowBS.h
- RBS_DMM_FreeMem() : RainbowBS.h
- RBS_DMM_GetHMemSize() : RainbowBS.h
- RBS_DMM_ReallocMem() : RainbowBS.h
- RBS_DMM_RegisterBlock() : RainbowBS.h
- RBS_DMM_RegisterPool() : RainbowBS.h
- RBS_DMM_UnuseHBlock() : RainbowBS.h
- RBS_DMM_UnuseHMem() : RainbowBS.h
- RBS_DMM_UseHBlock() : RainbowBS.h
- RBS_DMM_UseHMem() : RainbowBS.h
- RBS_GetCPUBits() : RainbowBS.h
- RBS_GetLocalTime() : RainbowBS.h
- RBS_GetRunTime() : RainbowBS.h
- RBS_GetSysInfo() : RainbowBS.h
- RBS_GetTickCount() : RainbowBS.h
- RBS_GetVersionString() : RainbowBS.h
- RBS_Init() : RainbowBS.h
- RBS_IsCPULittleEndian() : RainbowBS.h
- RBS_IsStackGrowDown() : RainbowBS.h
- RBS_MemCpy8() : RainbowBS.h
- RBS_MemSet16() : RainbowBS.h
- RBS_MemSet32() : RainbowBS.h
- RBS_MemSet8() : RainbowBS.h

- RBS_Number2String() : RainbowBS.h
- RBS_Read16B() : RainbowBS.h
- RBS_Read16L() : RainbowBS.h
- RBS_Read32B() : RainbowBS.h
- RBS_Read32L() : RainbowBS.h
- RBS_sin() : RainbowBS.h
- RBS_tan() : RainbowBS.h
- RBS_Write16B() : RainbowBS.h
- RBS_Write16L() : RainbowBS.h
- RBS_Write32B() : RainbowBS.h
- RBS_Write32L() : RainbowBS.h

- S -

- SBINARY : RainbowBS.h
- SDECIMAL : RainbowBS.h
- SEM : RainbowBS.h
- SEM_NAME : RainbowBS.h
- SHEX : RainbowBS.h
- t -
 - tETIME : RainbowBS.h
 - tPROCESS : RainbowBS.h

Generated by doxyden 1.8.9.1



- e -

- etimer_adjust() : RainbowBS.h
- etimer_expiration_time() : RainbowBS.h
- etimer_expired() : RainbowBS.h
- etimer_pending() : RainbowBS.h
- etimer_request_poll() : RainbowBS.h
- etimer_reset() : RainbowBS.h
- etimer_restart() : RainbowBS.h
- etimer_set() : RainbowBS.h
- etimer_start_time() : RainbowBS.h
- etimer_stop() : RainbowBS.h

- p -

- Port_Delay() : Port.h
- Port_FreeMutex() : Port.h
- Port_GetLocalTime() : Port.h
- Port_GetMutex() : Port.h
- Port_GetTickCount() : Port.h
- Port_Init() : Port.h
- Port_Printf_Error() : Port.h
- Port_Printf_Log() : Port.h
- Port_Printf_Warn() : Port.h
- process_alloc_event() : RainbowBS.h
- process_exit() : RainbowBS.h
- process_is_running() : RainbowBS.h
- process_nevents() : RainbowBS.h
- process_poll() : RainbowBS.h
- process_post() : RainbowBS.h
- process_run() : RainbowBS.h
- process_send() : RainbowBS.h
- process_start() : RainbowBS.h

- r -
 - RBS_cos() : RainbowBS.h
 - RBS_ctan() : RainbowBS.h
 - RBS_Delay() : RainbowBS.h
 - RBS_DMM_AllocBlock() : RainbowBS.h
 - RBS_DMM_AllocMem() : RainbowBS.h
 - RBS_DMM_AllocZeroBlock() : RainbowBS.h
 - RBS_DMM_AllocZeroMem() : RainbowBS.h
 - RBS_DMM_FreeBlock() : RainbowBS.h
 - RBS_DMM_FreeMem() : RainbowBS.h
 - RBS_DMM_GetHMemSize() : RainbowBS.h
 - RBS_DMM_ReallocMem() : RainbowBS.h
 - RBS_DMM_RegisterBlock() : RainbowBS.h
 - RBS_DMM_RegisterPool() : RainbowBS.h
 - RBS_DMM_UnuseHBlock() : RainbowBS.h
 - RBS_DMM_UnuseHMem() : RainbowBS.h
 - RBS_DMM_UseHBlock() : RainbowBS.h
 - RBS_DMM_UseHMem() : RainbowBS.h
 - RBS_GetCPUBits() : RainbowBS.h
 - RBS_GetLocalTime() : RainbowBS.h
 - RBS_GetRunTime() : RainbowBS.h
 - RBS_GetSysInfo() : RainbowBS.h
 - RBS GetTickCount() : RainbowBS.h
 - RBS_GetVersionString() : RainbowBS.h
 - RBS_Init() : RainbowBS.h
 - RBS_IsCPULittleEndian() : RainbowBS.h
 - RBS_IsStackGrowDown() : RainbowBS.h
 - RBS_MemCpy8() : RainbowBS.h
 - RBS_MemSet16() : RainbowBS.h
 - RBS_MemSet32() : RainbowBS.h
 - RBS_MemSet8() : RainbowBS.h
 - RBS_Number2String() : RainbowBS.h
 - RBS_Read16B(): RainbowBS.h
 - RBS_Read16L(): RainbowBS.h
 - RBS_Read32B(): RainbowBS.h
 - RBS_Read32L(): RainbowBS.h
 - RBS_sin() : RainbowBS.h
 - RBS_tan() : RainbowBS.h
 - RBS_Write16B(): RainbowBS.h

- RBS_Write16L() : RainbowBS.h
 RBS_Write32B() : RainbowBS.h
 RBS_Write32L() : RainbowBS.h

Generated by doxygen 1.8.9.1



- tETIME : RainbowBS.h
- tPROCESS : RainbowBS.h

Generated by doxyden 1.8.9.1



• ePOW : RainbowBS.h

Generated by



- SBINARY : RainbowBS.h
- SDECIMAL : RainbowBS.h
- SHEX : RainbowBS.h

Generated by doxyden 1.8.9.1



- p -

- PROCESS : RainbowBS.h
- PROCESS_BEGIN : RainbowBS.h
- PROCESS_CONTEXT_BEGIN : RainbowBS.h
- PROCESS_CONTEXT_END : RainbowBS.h
- PROCESS_CURRENT : RainbowBS.h
- PROCESS END : RainbowBS.h
- PROCESS_EXIT : RainbowBS.h
- PROCESS_EXITHANDLER : RainbowBS.h
- PROCESS_NAME : RainbowBS.h
- PROCESS_PAUSE : RainbowBS.h
- PROCESS_POLLHANDLER : RainbowBS.h
- PROCESS_PT_SPAWN : RainbowBS.h
- PROCESS_SIGNAL_SEM : RainbowBS.h
- PROCESS_THREAD : RainbowBS.h
- PROCESS_WAIT_EVENT : RainbowBS.h
- PROCESS_WAIT_EVENT_UNTIL : RainbowBS.h
- PROCESS_WAIT_SEM : RainbowBS.h
- PROCESS_WAIT_UNTIL : RainbowBS.h
- PROCESS_WAIT_WHILE : RainbowBS.h
- PROCESS_YIELD : RainbowBS.h
- PROCESS_YIELD_UNTIL : RainbowBS.h
- PT_BEGIN : pt.h
- PT_END : pt.h
- PT_EXIT : pt.h
- PT_INIT : pt.h
- PT_RESTART : pt.h
- PT_SCHEDULE : pt.h
- PT_SEM_INIT : pt-sem.h
- PT_SEM_SIGNAL : pt-sem.h
- PT_SEM_WAIT : pt-sem.h
- PT_SPAWN : pt.h
- PT_THREAD : pt.h

- PT_WAIT_THREAD : pt.h
- PT_WAIT_UNTIL : pt.h
- PT_WAIT_WHILE : pt.h
- PT_YIELD : pt.h
- PT_YIELD_UNTIL : pt.h

- r -

- RBS_DEBUG_ELSIF : RainbowBS.h
- RBS_DEBUG_ENDIF : RainbowBS.h
- RBS_DEBUG_ERROR : RainbowBS.h
- RBS_DEBUG_ERROR_FORMAT : RainbowBS.h
- RBS_DEBUG_IF : RainbowBS.h
- RBS_DEBUG_LOG : RainbowBS.h
- RBS_DEBUG_LOG_FORMAT : RainbowBS.h
- RBS_DEBUG_STA : RainbowBS.h
- RBS_DEBUG_WARN : RainbowBS.h
- RBS_DEBUG_WARN_FORMAT : RainbowBS.h

- S -

- SEM : RainbowBS.h
- SEM_NAME : RainbowBS.h

Generated by doxyden 1.8.9.1



RainbowBS Directory Reference

Directory dependency graph for RainbowBS:

RBSSource Rain	bowBS
----------------	-------

Directories

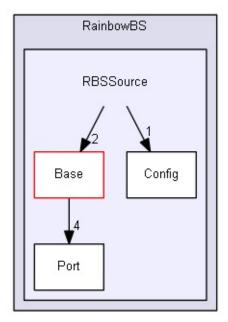
directory **RBSSource**

Generated by



RBSSource Directory Reference

Directory dependency graph for RBSSource:



Directories

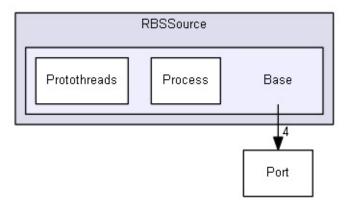
directory	Base
directory	Config
directory	Port

file RainbowBS.h



Base Directory Reference

Directory dependency graph for Base:



Directories

directory **Process**

directory **Protothreads**

file	Convert.c		
file	Debug.c		
	-		
file	DMM.c		
file	MemOP.c		
file	System.c		
		Generated by	<u>doxygen</u> 1.8.9.1



Protothreads Directory Reference

Directory dependency graph for Protothreads:

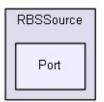
Base	
Protothreads	

file	e Ic-addrlabels.h	
file	e Ic-switch.h	
file	e lc.h	
file	e pt-sem.h	
file	e pt.h	
		Generated by



Port Directory Reference

Directory dependency graph for Port:



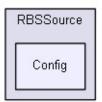
file Port.c

file Port.h



Config Directory Reference

Directory dependency graph for Config:



file RainbowBSConf.h



Process Directory Reference

Directory dependency graph for Process:

Base	
Process	

file etimer.c

file process.c