PX5 [RTOS]

User Guide

Version 5.0.0.P1

Published January 25, 2023

Table of Contents

A

About this guide

The purpose of this guide is to provide embedded developers with the information necessary to successfully utilize the PX5 Real-Time Operating System (RTOS). This guide is organized into a series of chapters as follows:

Chapter 1: Introducing the PX5 RTOS. This chapter briefly introduces multithreading, pthreads, and the PX5 RTOS with pthreads+ extensions.

Chapter 2: Brief history of POSIX pthreads. This chapter provides a brief history of POSIX pthreads.

Chapter 3: PX5 RTOS Installation and Use. This chapter provides a high-level overview of how to install and use the PX5 RTOS.

Chapter 4: PX5 RTOS Safety and Security. This chapter provides an overview of safety and security in PX5 RTOS applications.

Chapter 5: PX5 RTOS Pointer/Data Verification. This chapter provides an overview of the unique Pointer/Data Verification capabilities of the PX5 RTOS.

Chapter 6: PX5 RTOS Functional Overview. This chapter provides a functional overview of the PX5 RTOS.

Chapter 7: PX5 RTOS API Definitions. This chapter describes the various APIs–both standard pthread APIs and pthreads+ extensions–of the PX5 RTOS.

Conventions

Various conventions are used in this guide. C source code function prototypes and examples are written in `courier` font. Supplemental documentation, API, and parameter names are italicized when discussed. There are also several symbols that are used to highlight important features or topics, as follows:

Symbol	Meaning
ⓘ	This is a general information symbol.
⚠	The caution symbol indicates that the user should be aware of important usage scenarios associated with a specific topic or API.
✖	The danger symbol indicates that the user should be aware of the serious consequences of certain scenarios associated with a specific topic or API.
NP	This symbol indicates the associated API scenario does not result in preemption.
P	This symbol indicates the associated API scenario results in preemption.
S	This symbol indicates the associated API scenario results in suspension.

Feedback

All feedback is greatly appreciated. Please send e-mail feedback to support@px5rtos.com with "*PX5 RTOS User Guide Feedback*" in the subject line.

Chapter

1

Chapter 1: Introducing the PX5 RTOS

The Industrial Grade PX5 RTOS is an advanced, 5th generation RTOS designed for the most demanding embedded applications. Its ultra-small size (< 1KB for minimal use) allows PX5 RTOS to fit into some of the most memory-constrained devices. Its ultrahigh-performance (sub-microsecond context switching and API calls on most microprocessors) and rich determinism make it ideal for the most demanding real-time needs. PX5 RTOS also boasts best-of-class safety and security. The entire PX5 RTOS code base is rigorously tested (100% C statement and branch decision coverage testing for every release). In addition, PX5 RTOS also offers *Pointer/Data Verification (PDV)*, a unique technology for unprecedented run-time function pointer, system object, buffer, and stack verification. Perhaps most important, PX5 RTOS is simple. The API consists of a native implementation of the POSIX pthreads standard, which is well known throughout the industry and makes applications written for PX5 RTOS highly portable to any POSIX pthread implementation–whether that is Linux or even another RTOS supporting the pthread API. The PX5 RTOS also offers optional POSIX pthreads extensions that are designed specifically for deeply embedded, real-time applications.

How the PX5 RTOS benefits you

The PX5 RTOS features benefit embedded development in many ways, including the following:

- Accelerated time-to-market
- Enhanced product quality
- Safer and more secure products
- More portable/reusable application code via industry standard pthreads API
- Reduced engineering training via pthreads API
- Professional support

Why use an RTOS?

Backing up a bit, it's important to briefly discuss why an RTOS is important in the first place. Not all embedded applications require an RTOS. Some elementary applications can perform all of their required processing in a control loop within the C *main* function, as the example illustrates below:

```
int  main(void)
{

    while(1)
    {

        /* Process primary task.  */
        process_primary_task();

        /* Process secondary task.  */
        process_secondary_task();

     }
}
```

Typically, applications less than 64KB in total memory with no network connectivity or physical device I/O may be able to use and even benefit from a simple control loop. The simple control loop technique eliminates the memory and processing cycles required by a more robust RTOS platform. However, it places all the responsibility for allocating processor cycles and meeting real-time requirements on the application code itself. Take, for example, the simple control loop illustrated above. If the *process_primary_task* function has a real-time requirement, the application developer must ensure that the worst-case processing time of the rest of the control loop is small enough to meet that real-time requirement. This may not be an easy task. To accomplish this, the developer must thoroughly analyze and test all code paths before and after the call to *process_primary_task*. Of course, this becomes exponentially more difficult as the code size/complexity increase or if new real-time requirements are added. One might think, "well, I'll just add more calls to *process_primary_task* in the control loop, so I don't have to calculate the entire worst-case processing of the entire loop." This can temporarily help; however, now there is extra processing overhead (extra calls to *process_primary_task* equates to additional overhead). At some point, this additional overhead can be greater than the context switching overhead of an RTOS.

Waiting for peripheral device I/O (blocking) is another difficult issue to overcome with a simple control loop design. Suppose *process_secondary_task* is controlling a peripheral I/O device and has to wait for the peripheral I/O device to complete its operation. Indeed, the real-time requirements in *process_primary_task* could be adversely affected, and this could be even worse if the wait time is not deterministic. In this situation, the application developer would likely be forced to create a state machine inside of *process_secondary_task* such that it wouldn't ever wait, instead setting up a state machine such that it could "find its way back" to the peripheral I/O device operation to check for completion. This state machine processing necessarily adds complexity and also adds overhead. In contrast, if an RTOS is used, the thread implementing *process_secondary_task* can simply suspend (block) in-line on the peripheral I/O device operation. When the I/O completes, the thread resumes immediately where it last executed–no complicated state machine to find its way back, i.e., there is less complexity and overhead.

Finally, development without an RTOS requires every developer to have precise knowledge of the real-time processing of each component. Not only does this become exponentially more difficult as the complexity and real-time requirements increase, it also doesn't scale well as more developers are added to the project. In contrast, an RTOS makes it possible to encapsulate application components into threads of varying priorities. For example, the *process_primary_task* functionality could simply be encapsulated inside the highest priority thread, and thus nothing else in the system would interfere with meeting its real-time requirements. Thus, developers are freed from having to know (and avoid impacting) the system real-time needs, which lets them focus solely on their particular area of interest.

The following is an example of the control loop converted to operate using the pthread API:

```
pthread_t primary_thread_handle;


void *  process_primary_task(void *  argument)
{

struct sched_param param;


      /* Assuming priority 31 is highest.  */
      param.sched_priority = 31;

      /* Raise the priority of the real-time thread. */
      pthread_setschedparam(pthread_self(),SCHED_OTHER, &param);

       while(1)
       {

          /* Perform third real-time duties and suspend until more
             work needs to be done.  */
       }
}

int     main(void)
{


      /* Create a thread for the real-time requirements associated with the
         primary task. */
      pthread_create(&primary_thread_handle, NULL, process_primary_task, NULL);

      /* Relinquish to the primary thread. */
      sched_yield();

      while(1)
      {
           /* Perform the secondary task from the main thread.  */
           process_secondary_task();
      }
}
```

The example shows how easy it is to convert the processing of a control loop implementation into two threads under an RTOS–where the main loop is now a lower priority thread, and the processing in *process_primary_task* is contained within a higher-priority thread that will preempt whenever it has real-time processing to accomplish. The beauty of an RTOS-based design is that boundless new functionality can be added without adversely affecting the real-time processing in *process_primary_task*, assuming, of course, that it remains the highest priority thread.

In summary, an RTOS greatly reduces the complexity of an application–especially in terms of meeting real-time requirements and elimination of state machines necessitated from a lack of in-line suspension. It's possible an RTOS can even reduce overhead–in situations where the control loop deficiencies have necessitated a significant amount of logic to manage the real-time requirements of the application. Here are some advantages of using an RTOS in bullet form:

- Enhances application real-time responsiveness

- Reduces complexity and makes development easier
- Easier to divide an application into more manageable pieces
- Enables more features and project developers
- Possible reduction of overhead via elimination of polling and state machines
- Achieve concurrency by enabling other processing while waiting for blocking I/O
- Enable true parallel processing in symmetric multiprocessing RTOS environments

The PX5 RTOS is here to help!

As mentioned at the beginning of this chapter, PX5 RTOS combines best-of-class RTOS technology with the industry standard pthread API. In addition, the PX5 RTOS provides optional pthreads+ extensions, which add functional enhancements designed to complement the POSIX pthreads API to better address embedded multithreading requirements. In summary, the PX5 RTOS provides embedded developers the best of both worlds–best-of-class RTOS technology united with the industry standard pthreads API!

Chapter

2

Chapter 2: Brief history of POSIX pthreads

The Portable Operating System Interface (POSIX) is a family of standards defined and maintained by IEEE. The basic idea behind POSIX is application portability across different hardware and operating systems, providing the operating system adheres to the POSIX API standard. Originally found in UNIX, many of today's operating systems support POSIX - most significantly, all Embedded Linux distributions.

The first POSIX standard was the IEEE Std 1003.1-1988 specification and was released in 1988. The latest POSIX standard was released in 2017 (POSIX.1-2017 IEEE Std 1003.1-2017). Originally, all multitasking in POSIX was process based. Communicating and switching between processes required significant overhead as well as significant hardware resources (virtual memory support, large amounts of memory, super fast processors, etc.). In order to achieve reduced overhead, the POSIX pthreads specification was introduced in the IEEE Std 1003.1c-1995 specification of 1995. This concept of lightweight multithreading – including thread synchronization and communication primitives – provided a useful new paradigm to application developers. With POSIX pthreads, developers could share global variables and data structures between threads using low-overhead mutual exclusion to coordinate the access.

The lightweight nature of the POSIX pthreads API make it especially ideal for resource constrained embedded devices. The POSIX pthread APIs are also relatively intuitive and easy to use. Take, for example the *pthread_create* API to create a new thread with the entry function of "*my_thread_entry*":

pthread_create*(&my_thread_handle, NULL, my_thread_entry, NULL);*

The pthread create API effectively requires only two elements from the application developer – a thread handle and an entry function for the thread. Notably, the *pthread_create* API is much simpler than most RTOS thread/task creation APIs – many of which require more parameters as well as the accompanying complexity.

The POSIX pthread standard is also robust. It specifies a plethora of services, including mutual exclusion, synchronization, and communication primitives. There are also additional POSIX (not pthreads per-se) APIs that are applicable to resource constrained embedded systems, including semaphores, message queues, signals, and time-related services.

Why pthreads for Embedded?

According to some recent surveys, Embedded Linux accounts for nearly 70% of embedded development. As mentioned previously, Embedded Linux is based on the POSIX pthread API, which makes pthreads the most well-known and used API in the embedded industry. Historically, there are many proprietary RTOS APIs in the embedded space–many of them are quite good. However, they all require significant developer training, which limits the use (and adoption) of the underlying RTOS. Proprietary RTOS APIs make application code less portable. Furthermore, many device makers have both Linux and RTOS-based devices, servicing different target markets and price points. An RTOS based on the industry standard POSIX pthreads API solves these problems – reducing developer training and making application code portable across a wide range of platforms.

Chapter
3

Chapter 3: PX5 RTOS Installation and Use

Installing and using the PX5 RTOS is ultrasimple. Starting from a simple C *main* program, there are conceptually three simple steps to install and start using the PX5 RTOS. Of course, the exact installation is processor/tool specific. Please review the binding layer documentation for more details.

3 Simple Steps

1. Place the PX5 RTOS distribution into your C *main* project source directory.

2. The PX5 RTOS distribution contains two main source files, namely ***px5.c*** and ***px5_binding.s,*** as well as supporting C header files. Simply add both of them to your project (IDE or makefile).

3. Modify your C main program to include ***pthread.h*** and call *px5_pthread_start* in your main program. Next, create a *while(1)* loop, since *px5_pthread_start* upscales the *main* function into your system's first thread!

   Your C main program should now look something like the following (PX5 RTOS specific additions in red):

```
#include “pthread.h”

int   main(void)
{
    /* Call PX5 RTOS initialization.  */
    px5_pthread_start(1, NULL, NULL);

    /* We are now in the context of a thread.  */
    while(1)
    {

        /* All other PX5 API calls are now available!  */
    }
}
```

After these three easy steps, you should be up and running with the PX5 RTOS! To enable timer related services, simply add a call to *px5_timer_interrupt_process* from within the periodic timer interrupt handler.

Typically, there are no linker control file changes required to install and use the PX5 RTOS. There shouldn't be any project setting changes either if the PX5 RTOS source code is placed in the same directory as the C main function source. Otherwise, if the PX5 RTOS source is placed in another location, you will need to update the C include paths of your project to point to the PX5 RTOS header files.

Please review the *px5_pthread_start* API defined later in this guide. This API provides parameters for run-time identification and memory for object creation. It also performs important error checking – including verification of the binding between the *px5.c* C source and the *px5_binding.s* assembly code. The return code of this API should always be checked. For the sake of clarity, we omitted checking for API return code errors in this simple example.

PX5_RTOS_Binding_User_Guide.pdf

As mentioned previously, please review the *PX5_RTOS_Binding_User_Guide.pdf* for more detailed information on specific information pertaining to the operation of the PX5 RTOS with the processor and development tool you are using – including binding-specific configuration options. This guide is also where you will find information about ready-to-run example(s) specific to your processor and development tool, including guidance on how to enable periodic timer-related services.

Defined PX5 Symbol

If you include any of the PX5 RTOS include files, the symbol *PX5* is defined. This is useful for cross-platform applications such that conditionals can be placed around PX5 RTOS extensions.

Configuration Options for the PX5 RTOS

There are a multiple compile-time configuration options for building and using the PX5 RTOS. The following describes each option in detail (note that processor/tool specific configuration options are defined in the *PX5_RTOS_Binding_User_Guide.pdf* guide):

Build Option	**Meaning**
PX5_CANCELLATION_POINTS_DISABLE	When defined, the cancellation points in the blocking APIs are disabled, resulting in improved performance. This option is not defined by default.

PX5_DEFAULT_PRIORITY	When defined, this value overrides the default threads priority for thread creation (priority 16 is the default thread create priority).
PX5_DEFAULT_SIGNAL_MASK	When defined, this value overrides the default signal bit mask of the main thread (all signals masked - 0xFFFFFFFF).
PX5_DEFAULT_STACK_SIZE	When defined, this value overrides the default stack size, which is processor-specific (typically on the order of 1K bytes).
PX5_FUNCTION_POINTER_VERIFY_ENABLE	When defined, all function pointers used internally in the PX5 RTOS are evaluated against the verification code that was established when they were setup (using PDV). This mechanism helps early detection of memory corruption – both intentional and non-intentional. This option is not defined by default.
PX5_MEMORYPOOL_VERIFY_ENABLE	When defined, all internal PX5 RTOS memory pool linked-lists are evaluated against the verification code that was established when the memory pool is created. The verification also occurs as memory from the pool is allocated or released. This mechanism helps early detection of memory corruption, most usually associated with the application writing past the allocated memory. This option is not defined by default.
PX5_OBJECT_VERIFY_ENABLE	When defined, all internal PX5 RTOS objects (including the global PX5 RTOS data) are evaluated against

	the verification code that was established when they were created (using PDV). This mechanism helps early detection of memory corruption – both intentional and non-intentional. This option is not defined by default.
PX5_PARAMETER_CHECKING_DISABLE	When defined, basic parameter checking is disabled (by default parameter checking is enabled). This can be used on a per-file basis, i.e., parameter checking can be enabled (default) for some application files and disabled via this option in other files. This option is not defined by default.
PX5_SPECIFIC_ERRNO	When defined, the PX5 RTOS does not redefine *errno* in the applicaton code. Instead, the *px5_errno* symbol is remapped to retrieve the thread-specific API error information. The application may also use the *px5_errno_get* API directly. This define is useful in situations where multiple software entities have their own definition of errno. This option is not defined by default.
PX5_STACK_CHECK_ENABLE	When defined, the PX5 RTOS performs stack size checking, including update of the minimal available stack and determining if the stack has overflowed or is in imminent danger of an overflow. This option is not defined by default.
PX5_STACK_VERIFY_ENABLE	When defined, the PX5 RTOS performs stack integrity checking

	(using PDV), including verification of the function call return address when possible. This option is not defined by default.
PX5_TICKS_PER_SECOND	By default, this option is defined as 1000, representing a 1ms timer interrupt frequency. However, the application may use a different timer interrupt frequency, just as long as this define is adjusted accordingly.
PX5_TIME_REMAPPING_DISABLE	When defined, the PX5 RTOS does not remap user time types (e.g., *time_t* and *timespec*) to the PX5 RTOS equivalents. If this is defined, the application code would have to prepend px5_ to any time type referenced, e.g., *px5_time_t* and *px5_timespec*. This define is useful in situations where multiple software entities have their own definition of time types. This option is not defined by default.

Using Configuration Options

The PX5 RTOS configuration options mentioned previously may be defined via project setting of via -D compiler command line options. Alternatively, the px5_user_config.h in the PX5 RTOS distribution is dedicated for application use, i.e., it is a safe place to define configuration options for the PX5 RTOS since it does not change with each new version of the PX5 RTOS.

⚠ *It is highly recommended to compile the application code with the same configuration options as used to compile the PX5 RTOS source files (px5.c and px5_binding.s).*

Troubleshooting

The PX5 RTOS is designed for ease-of-use and reliability, so it's not likely you will experience issues. However, if you do experience issues, here are some basic troubleshooting suggestions that may help:

1. Make sure that the processor/tool-specific examples mentioned in the *PX5_RTOS_Binding_User_Guide.pdf* (or supplemental documentation) are executing normally in your environment.

2. Your application code should always check the return status on all API calls.

3. Be suspicious of any recent change(s).

4. Please make sure that each thread has a large enough stack allocated, which means that each thread stack must be large enough to hold the worst-case C function call depth with memory sufficient for all local variables. Please review the stack checking APIs and functionality described later in this guide for assistance. Thread stack corruption is often the cause of unusual run-time problems in any RTOS-based application.

5. If a ready thread isn't running, ensure that one or more higher priority threads eventually suspend such that lower priority threads are given a chance to execute. Remember to use as few priorities as possible, which will reduce context-switching overhead as well as the potential for thread starvation.

6. If time-related services are not working, verify that the periodic timer interrupt is occurring and you are calling *px5_timer_interrupt_process* from within the interrupt handler.

7. Set a breakpoint on the PX5 RTOS central error handling function *px5_internal_central_error_process*. If the breakpoint is hit, the type of error, executing threads, current interrupt level, as well as the caller, should provide valuable debug information.

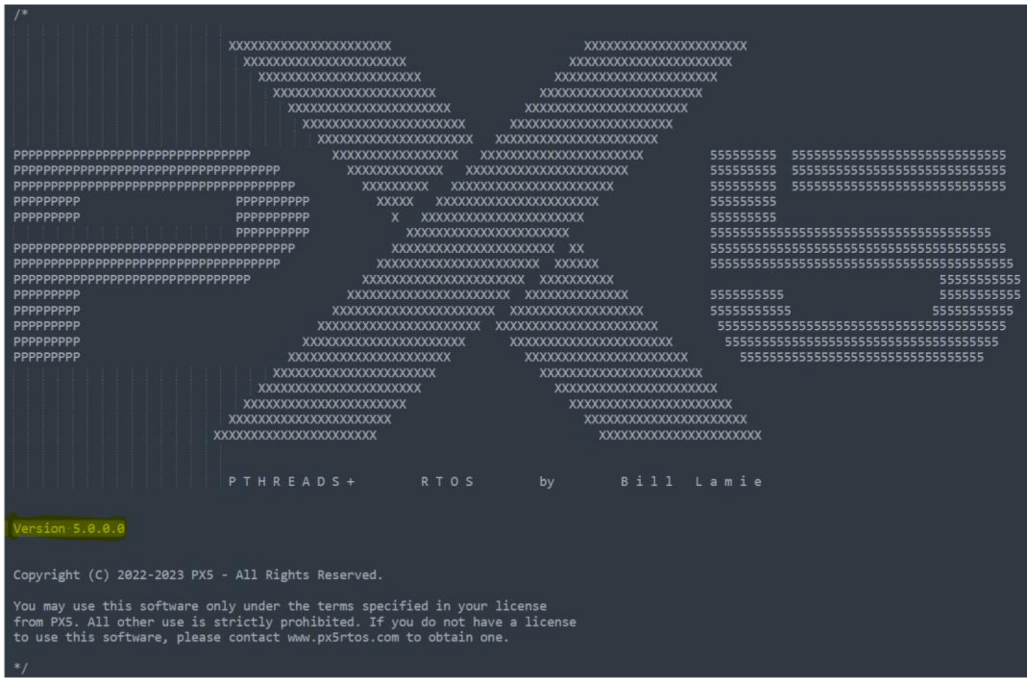
Version Information

The PX5 RTOS version identification is comprised of four numbers separated by periods in the general format of V.M.U.P, with the following meaning:

Version Number	**Meaning**
V	Major version
M	Minor version

U	Update version
P	Patch version

The specific version of PX5 you are using can be found near the top of any source file, as shown below:



```
/*

                    P T H R E A D S +     R T O S     by     B i l l   L a m i e

Version 5.0.0.0

Copyright (C) 2022-2023 PX5 - All Rights Reserved.

You may use this software only under the terms specified in your license
from PX5. All other use is strictly prohibited. If you do not have a license
to use this software, please contact www.px5rtos.com to obtain one.

*/
```

The following constants in px5.h also contain the version:

```
#define PX5_MAJOR_VERSION                          5
#define PX5_MINOR_VERSION                          0
#define PX5_UPDATE_VERSION                         0
#define PX5_PATCH_VERSION                          0
```

Long Term Support (LTS)

We support major versions for five years (the first number in the version ID). Minor versions (represented by the second number) are supported for two years.

Contact Us

Please feel free to contact us with any problem – we are glad to help! To make the request as efficient as possible, please provide the following information:

1. Your name and company name

2. Your *px5_binding.h* and *px5.h* header files

3. Brief description of the problem

4. Screen shots and any data collection (sometimes a memory display of the *px5_globals* data structure is helpful)

The best place to initiate a support ticket is on the PX5 web site:

www.px5rtos.com/support

You may also send an e-mail to:

support@px5rtos.com

You may also contact us by mail and telephone:

PX5
11440 West Bernardo Court, Suite 300
San Diego, CA   92127
Phone: +1 (858) 753-1715

Chapter
4

Chapter 4: PX5 RTOS Safety and Security

Today, safety and security for embedded devices are paramount. Although safety and security are distinct areas, in the embedded PX5 RTOS context, they have a significant degree of overlap. Memory corruption – either intentional or un-intentional is the most common source of safety and security issues in embedded devices. This is also where the PX5 RTOS can make a significant difference. That said, the PX5 RTOS safety and security features are a piece of a greater defense-in-depth solution that includes the PX5 RTOS, application software, device hardware, and other network/cloud entities and their configuration/settings.

Exactly what safety and security mean differ depending on the application. The safety and security requirements for any specific application are ultimately a combination of the attack surface as well as what is practical – both from a technological and business standpoint. Stated another way, embedded safety and security isn't a one-size-fits-all but rather a deliberate risk-benefit analysis based on each specific use case.

Hardware Safety & Security Features

As for hardware safety and security, there are a wide variety of embedded processors with an even greater variety of safety and security features, the most common of which are:

1. Anti-tampering. This hardware feature protects the firmware IP of the device from unauthorized access. It is generally outside the scope of the PX5 RTOS or application firmware but an important consideration when selecting hardware.

2. Lock-step execution. This hardware feature employs multiple processors executing the same code with the same data. The goal being that exact code execution is guaranteed. Such hardware is mostly found in safety critical applications. This is effectively invisible to the PX5 RTOS or most of the application firmware.

3. Anti-glitch. This hardware feature employs circuitry to prevent an attacker from causing abnormal program execution via manipulating power or

other system signals. This too is generally outside the scope of the PX5 RTOS or application firmware but may be an important consideration when selecting hardware.

4. Execute only from flash. Most microcontrollers (MCUs) execute instructions from flash. Some of these MCUs are able to prohibit execution from RAM, which is recommended to help prevent dynamic insertion of malicious code in remote execution attacks. This sometimes requires the application firmware to enable, but otherwise is invisible to the PX5 RTOS or the application.

5. Hardware stack limit. Some processors have a stack limit register that guards against memory corruption caused by stack overflow. This is generally implemented as an additional register and is setup by the PX5 RTOS on each thread context switch.

6. Hardware watchdog timer. Many processors have a non-maskable hardware watchdog timer. This feature acts as a fail-safe. Under normal operation, the application code resets the watchdog on a regular basis and always before its expiration. During abnormal execution, the watchdog is likely not reset, thus leading to a non-maskable interrupt, which halts the abnormal execution. Typically, applications will simply reset after a watchdog expiration.

7. True Random Number Generator (TRNG). Having a TRNG or even more basic Random Number Generator (RNG) in hardware is very beneficial. This is most important for networked devices, but it’s also useful for the PX5 RTOS – especially for the unique PX5 RTOS Pointer/Data Verification (PDV) feature.

8. Memory Management Unit (MMU). This hardware feature enables access restrictions of various regions of memory. This feature is typically only available on larger, more powerful processors. This is most often setup once by the application firmware after reset to map and protect various memory regions.

9. Memory Protection Unit (MPU). This hardware feature is similar to the MMU but found in smaller, more resource constrained devices. Again, this is typically setup by the application firmware after reset to protect various memory regions. In cases where there isn’t stack limit registers, the MPU can be used to setup a protected block at the top of each thread’s stack in order to prevent stack overflow. This functionality would need to be accomplished inside the PX5 RTOS thread context switching logic.

10. Secure Element (SE) or Trusted Platform Module (TPM). For devices that are network connected, having a SE or TMP allow credentials or other secrets to be securely isolated from the main application, therefore can greatly increase the network security of the device and is therefore highly recommended.

Of course, each of the hardware safety and security features mentioned have an associated cost – in circuitry, power consumption, size, etc. These costs must go through the risk-benefit analysis mentioned previously.

Software Safety & Security Features

Some amount of software support is required to utilize the hardware safety and security features, e.g., setting up the hardware stack limit feature or the MPU to protect certain memory areas. The PX5 RTOS binding layer is where such support is typically located. The application firmware may also have logic to support the various hardware safety and security features.

PX5 RTOS Pointer/Data Verification (PDV)

As for software-only safety and security measures, the PX5 RTOS provides Pointer/Data Verification (PDV), which is a unique software-only technique to help detect corruption of important data like function pointers, function return addresses, internal system objects, allocated memory, etc. PDV utilizes the pointer or data value, the storage location of the verification code, and the unique run-time identification provided to *px5_pthread_start* to create a verification code (fingerprint) for each important data element during its initialization. Before the important data is used, it is authenticated against the verification code. If corruption is detected, the application is alerted via the PX5 RTOS central error handing, at which point the application can take the necessary measures to respond to the memory corruption. PDV helps detect memory corruption early and greatly reduces the chance of unwanted execution associated with function pointer corruption.

Run-Time Stack Checking & Verification

Additional software-only stack safety and security measures are offered by the PX5 RTOS. When enabled, run-time stack checking examines the current stack pointer upon function entry to check for overflow or imminent overflow, as well as keeping track of the minimal available amount of stack memory. Simply build the PX5 RTOS source with *PX5_STACK_CHECKING_ENABLE,* and each thread's stack is checked throughout PX5 RTOS execution. The application may also utilize the *px5_pthread_stack_check* API to perform stack checking from the application C code. Stack verification utilizes PDV to help verify stack integrity – specifically the caller return address on the stack when supported by the compiler – before returning to the caller of a function. To enable stack verification, simply build the PX5 RTOS source with *PX5_STACK_VERIFY_ENABLE* defined.

If a stack corruption or overflow is detected, the central error handling function is called with a fatal error. If stack checking detects a stack that is at risk of overflow, the central error handling is called with the appropriate advisory error.

Application Best Practices

In addition to PDV and run-time stack checking, there are also additional application *best practices* for enhanced safety and security, as follows:

1. Harden the device firmware. The PX5 RTOS was implemented in a Test Driven Development (TDD) manner, which basically means the tests are written before the actual code. Furthermore, by design, the PX5 RTOS code base must achieve 100% statement and 100% branch/decision coverage testing before each minor release. We recommend the application firmware take a similar approach – there is never enough testing. The more vetted the software is, the safer and more secure it is.

2. Leverage static analysis and related tools. In addition to the hardening mentioned previously, it's a good idea to leverage static analysis tools as well as penetration testing and fuzzing tools. These tools help find subtle issues in advance, which is often much easier than debugging fielded devices.

3. Use PX5 RTOS Pointer/Data Verification (PDV). The PX5 RTOS optionally (as determined by *px5.c* build options) uses PDV to verify function pointers, stack integrity, internal system objects, and allocated memory. Through API extensions, the application is also able to utilize PDV to verify its important function pointers and data. By using PVD, memory corruption can be detected early, and the possibility of unwanted program execution can be greatly reduced – including malicious remote execution.

4. Use an adequate (or larger) stack size. Stack overflow is the number one cause of memory corruption in embedded systems. Each thread stack must have enough memory for its worst-case function call nesting – including all local variables in each function. If not, the stack may overflow into the memory directly preceding the stack. This problem can be mitigated by using hardware stack limit features or, alternatively, the MPU/MMU to guard the area directly above the stack. The PX5 RTOS stack checking features are also helpful in preventing stack overflow issues.

5. Use the PX5 RTOS run-time stack checking. The PX5 RTOS run-time stack checking is an easy way to detect (and correct) stack overflows.

6. Explicitly specify and check buffer sizes. In all functions where a buffer is supplied, it is important for the caller to explicitly provide the size of the buffer and the callee to explicitly check the size to avoid overrun. The PX5 RTOS does this internally, and the application firmware should as well.

7. Be mindful of more likely areas of memory corruption. When a thread stack overflows, it generally corrupts the memory immediately *preceding* the thread stack memory (in most architectures, thread stack grows toward lower addresses). In contrast, buffer overflows are more likely to corrupt memory immediately *following* the buffer. Knowing this, it might be safer to avoid placing critical data before stacks or immediately following buffers.

8. Use PDV to place markers before stacks and after data buffers to help detect memory corruption caused by stack and buffer overflows. Please see the *px5_pthread_pdv_ ** APIs for more details.

9. Be careful with function pointers. Function pointers provide an easy path to unwanted program execution – both unintentional and intentional. For example, it's not good practice to place function pointers inside buffers since a buffer overflow could overwrite the function pointer. This is the easiest way for an attacker to initiate unwanted remote execution. Of course, the PDV feature of the PX5 RTOS can be used to verify application function pointers before they are called, which helps mitigate this issue.

10. Test, test, and test. The PX5 RTOS was implemented in a Test Driven Development (TDD) manner, which basically means the tests are written before the actual code. Furthermore, by design, the PX5 RTOS code base must achieve 100% statement and 100% branch/decision coverage testing before each minor release. We recommend the application firmware take a similar approach – there is never enough testing. The more vetted the software is, the safer and more secure it is.

Chapter
5

Chapter 5: PX5 RTOS Pointer/Data Verification

Pointer/Data Verification is a software-only technique to help detect and mitigate both intentional and accidental memory corruption. The basic idea is that for important information, a verification code is created and stored in memory. Before the important information is used, the verification code is generated again and compared with what was stored previously. If they are not the same, memory corruption has occurred, and the PX5 RTOS immediately alerts the application by calling the central error handling function. It's important to note that the application can define exactly what happens in the central error handling.

Default Verification Code

The default formula for generating the verification code can be defined by the application. However, by default, the verification code is a combination of a run-time identification (secret) passed to the PX5 RTOS in the px5_pthread_start API, along with the value of the important information and the address to store the generated code. The default formula looks something like this:

Verification Code = ((Data Value) + (Address to Store Code) + (Secret)) ^ (Secret)

For verification codes that are run-time unique, we recommend using a True Random Number Generator (TRNG) if available in hardware. With use of a TRNG, the verification code for each important data element has a temporal property, i.e., it will be unique for each execution of the application running on top of the PX5 RTOS. This makes it much harder for hackers to successfully insert malicious information, such as function pointers for remote execution attacks. The address to store the verification code provides a special property to the verification code. It's unlikely that any two images will have the same exact memory layout, which again makes it more difficult for hackers to successfully change important information without detection.

Note that he default verification code can be changed by overriding the PX5 RTOS macro PX5_POINTER_DATA_VERIFY_CODE_COMPUTE to any formula desired by the application.

Important Information Verified

The PX5 RTOS provides optional PDV protection over a series of important internal information, including the following:

- All function pointers used in the PX5 RTOS
- Global data of the PX5 RTOS
- Internal system structures with the PX5 RTOS (threads, queues, etc.)
- Return addresses on internal PX5 RTOS functions
- Metadata pointers used for memory management
- API's for application-specific use of PDV

Enabling PDV

As mentioned, the use of PDV is optional and is not-enabled by default. In addition, it can be individually enabled for specific PX5 RTOS areas. To enable PDV, the PX5 RTOS source should be built with the following defines (depending on the exact verification requested):

PX5_FUNCTION_POINTER_VERIFY_ENABLE	When defined, all function pointers used internally in the PX5 RTOS are evaluated against the verification code that was established when they were setup. With this enabled, it's much harder for hackers to insert rogue function pointers in remote execution attacks.
PX5_OBJECT_VERIFY_ENABLE	When defined, all internal PX5 RTOS objects (including the global PX5 RTOS data) are evaluated against the verification code that was established when they were created. This mechanism facilitates early detection of memory corruption – both intentional and non-intentional.

PX5_MEMORYPOOL_VERIFY_ENABLE	When defined, all internal PX5 RTOS memory pool linked-lists are evaluated against the verification code that was established when the memory pool is created. The verification also occurs memory from the pool is allocated or released. This mechanism helps early detection of memory corruption, most usually associated with the application writing past the allocated memory.
PX5_STACK_VERIFY_ENABLE	When defined, the PX5 RTOS performs stack integrity checking, including verification of the function call return address when possible (when supported by the compiler).

PDV Overhead

The amount of overhead associated with using PDV depends on CPU architecture and the compiler but is generally minimal. Assuming the default verification code generation as described previously, the assembly code to generate the verification code is only a couple of instructions on a typical Arm Cortex-M architecture, as follows:

```
ADDS   R3, R2, R0
EORS   R3, R3, R4
```

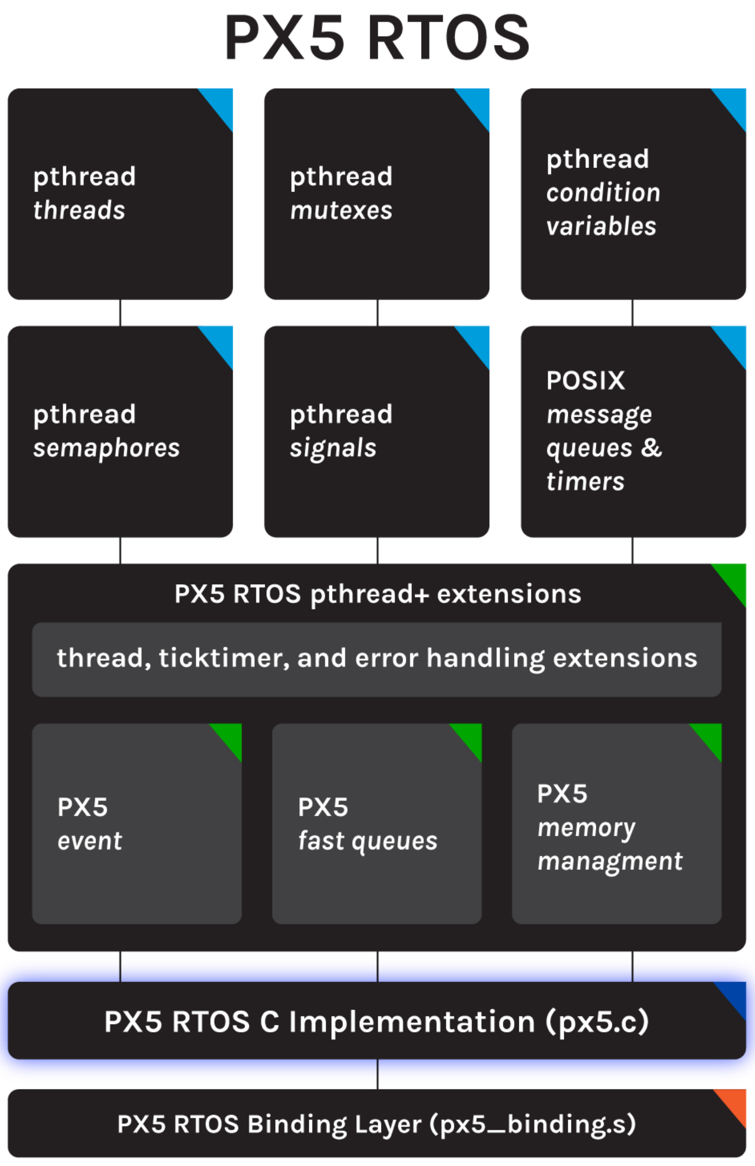
This code assumes that R0 contains the important data value, R2 contains the address to store the verification code, and R3 contains the run-time secret. It's reasonable to assume that each register might require a load instruction (LDR), and there will be one store instruction (STR) to store the code. Given all of that, to build and store the default verification code takes roughly six assembly instructions.

To verify the code, another six assembly instructions are required, along with another three instructions to load the previously stored code, compare it, and branch to either the "okay" path or to the central error handling.

Chapter
6

Chapter 6: PX5 RTOS Functional Overview

This chapter provides a complete functional overview of the PX5 RTOS. The following block diagram provides a high-level overview of the PX5 RTOS:


PX5 RTOS
pthread threads
pthread mutexes
pthread condition variables
pthread semaphores
pthread signals
POSIX message queues & timers
PX5 RTOS pthread+ extensions
thread, ticktimer, and error handling extensions
PX5 event
PX5 fast queues
PX5 memory managment
PX5 RTOS C Implementation (px5.c)
PX5 RTOS Binding Layer (px5_binding.s)

Each feature of the PX5 RTOS are discussed in the following paragraphs.

Composition

The PX5 RTOS is comprised of two main source files, namely *px5.c* and *px5_binding.s*. As shown in the block diagram above, almost all of the PX5 RTOS functionality is implemented in ANSI C and resides in *px5.c*. All development tool and processor-specific logic is contained in assembly-language *px5_binding.s* file. The binding file is intentionally kept to a minimum in order to enhance PX5 RTOS portability. Detailed information regarding the binding layer assembly file for each unique processor and development tool support for the PX5 RTOS can be found in the accompanying *PX5_RTOS_Binding_User_Guide.pdf* document.

Namespace

All global symbols in the PX5 RTOS – both functions and global data – have names with *px5_* prepended. Hence, looking at a linker load map, the PX5 RTOS symbols are easily identified.

Native pthreads Implementation

The PX5 RTOS is a native implementation of the POSIX pthreads standards. By native, we mean that the pthreads API is not a layer on top of another RTOS API but instead is implemented directly. For example, the API *pthread_mutex_lock* is an actual C function implemented in *px5.c*, containing all of the mutex lock code necessary for operation. There are no other layers or internal RTOS primitives that this code relies on or is built on top of. The result is the fastest and smallest possible implementation of *pthread* APIs.

pthreads API Extensions

The POSIX pthreads standard provides services to create robust multithreaded applications, most commonly on Linux or Embedded Linux platforms. That said, embedded development is often more demanding, requiring advanced real-time capabilities – in terms of both performance and functionality. The PX5 RTOS implements the core pthreads APIs in the most real-time, deterministic way possible, satisfying the advanced real-time demands for embedded systems. As for functionality, the PX5 RTOS offers extensions to the pthreads standards – called pthreads+. These extensions include adding new functionality to the standard API set, as well as completely new functionality, e.g., event flags, fast queues, and memory management.

Internal Memory Management

The PX5 RTOS does not require dynamic memory allocation for its operation. However, each new system object (thread, mutex, semaphore, etc.) created by the application does require memory for its internal control structure. Some objects also have additional memory requirements, e.g., each thread requires as stack and each message queue require memory to store messages. The PX5 RTOS object memory requirements can be satisfied in the following ways:

1. One-time, sequential allocation for each object creation/initialization from the memory supplied to the *px5_pthread_start* API. This is the easiest approach – as well as the lowest overhead and fastest. For systems that are not dynamically destroying and re-creating system objects, this approach is sufficient.

2. Application explicitly provides memory for each object. Through attribute extensions, each object in the PX5 RTOS provides the application the ability to explicitly supply the memory required for the object's creation. If this approach is used throughout, no additional memory is required by the PX5 RTOS. In addition, this approach allows the application to specifically designate where every object's memory is located. Please see the following APIs for more information:

    pthread_attr_setstackaddr
    px5_mq_extendattr_setcontroladdr
    px5_mq_extendattr_setqueueaddr
    px5_pthread_attr_setcontroladdr
    px5_pthread_condattr_setcontroladdr
    px5_pthread_mutexattr_setcontroladdr
    px5_pthread_ticktimerattr_setcontroladdr
    px5_semattr_setcontroladdr

3. Dynamic memory management of the memory supplied to the *px5_pthread_start* API. This approach relies on the memory supplied to *px5_pthread_start*. However, it is fully managed underneath via a PX5 RTOS variable-length memory pool. This facilitates dynamic destruction and re-creation of objects. Please see the *px5_memory_manager_enable* API for more details.

4. Application override of the internal memory management. This approach allows the application to override the internal PX5 RTOS memory management – giving the application complete control of dynamic memory allocation within the PX5 RTOS. Please see the *px5_memory_manager_get* and *px5_memory_manager_set* API for more details.

Run-time Stack Checking/Verification

Memory corruption via stack overflow is the leading cause for erroneous program execution in most embedded applications. In addition to taking advantage of hardware stack limit protections for overrun, the PX5 RTOS provides run-time stack checking. Simply build the PX5 RTOS source with *PX5_STACK_CHECK_ENABLE,* and each thread's stack size is checked throughout PX5 RTOS execution. The application may also utilize the *px5_pthread_stack_check* API to perform stack checking from the application's C code.

Stack integrity checking is also available. Building the PX5 RTOS source code with PX5_STACK_VERIFY_ENABLE defined enables run-time integrity checking of each thread's stack. This integrity checking – via the PX5 RTOS PDV technology - includes verification of the function caller's return address, when accessible via the compiler.

If a stack corruption or overflow is detected, the central error handling function is called with a fatal error. If stack checking detects a stack that is at risk of overflow, the central error handling is called with the appropriate advisory error.

Central Error Handling

The PX5 RTOS provides central error handling, meaning that all system errors are routed to a single, internal handler. System errors are grouped into three basic categories as follows:

PX5_LEVEL_3_ERROR
PX5_LEVEL_2_ERROR
PX5_LEVEL_1_ERROR

Level 3 errors are considered fatal errors and are generally not recoverable. Level 2 errors are serious errors that could soon result in a fatal error. Level 1 errors are the least serious error and are often the result of an invalid API parameter. The default processing inside of PX5 is to simply register the error information received and return. However, the user is able to augment the error processing for each level by overriding these symbols (typically, this would be done in the *px5_user_config.h*):

PX5_LEVEL_3_ERROR_PROCESSING
PX5_LEVEL_2_ERROR_PROCESSING
PX5_LEVEL_1_ERROR_PROCESSING

During development, it's also good practice to set a breakpoint on the *px5_internal_central_error_process* routine.

Scheduling Policy

There are typically three scheduling policies offered in association with POSIX pthreads, as follows:

SCHED_FIFO	This is the basic priority based, preemptive scheduling policy where higher-priority threads preempt lower-priority threads. Aside from preemption, each thread runs until it is blocked, completes, or is terminated.
SCHED_RR	This simply adds time-slicing to *SCHED_FIFO*.
SCHED_OTHER	This is the implementation defined scheduling policy in pthreads, which is how the PX5 RTOS identifies its scheduling policy.

The PX5 RTOS employs a priority-based preemptive scheduling policy, which matches closely with SCHED_RR. However, since the PX5 RTOS implements per-thread time-slicing as well as various pthread extensions, the *pthread_attr_getschedparam* API returns SCHED_OTHER to indicate it is implementation defined. The PX5 RTOS does not support dynamic changing of the scheduling policy.

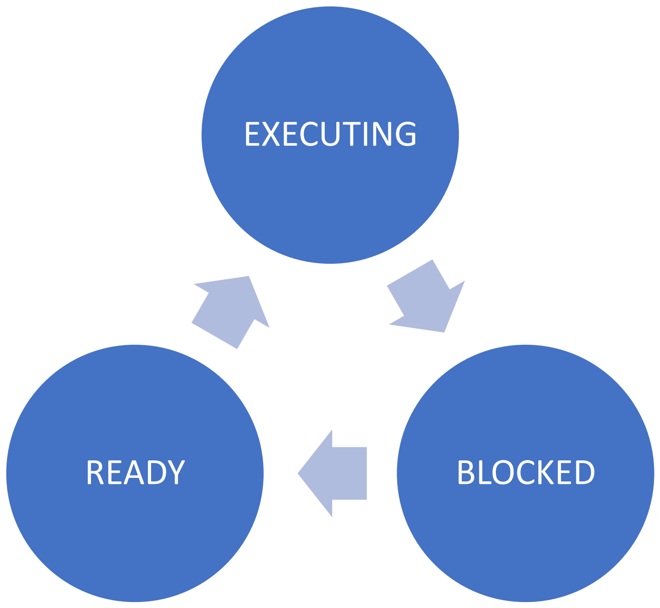
Thread Priorities

The PX5 RTOS supports 32 thread priority levels, ranging from 0 through 31. Priority level 0 is the lowest priority level (least important). Conversely, priority level 31 is the highest (most important). Preemption occurs (thread context switch) when a higher priority thread becomes ready during the execution of a lower priority thread.

Thread States

Each thread in the PX5 RTOS is in one of four high-level states – EXECUTING, READY, BLOCKED, or FINISHED. For single core processors, there is only one thread in the EXECUTING state, which generally represents the highest-priority, ready thread.

Threads that are in the READY state are waiting for their turn to enter the EXECUTING state. Threads that are in the BLOCKED state are waiting for an PX5 RTOS API call or system event from another thread or Interrupt Service Routine (ISR) that satisfies their previous request that caused them to enter the BLOCKED state. Once unblocked they enter the READY state. Threads in the FINISHED state have called *pthread_exit*, returned from their entry function, or werecanceled. The following diagram shows the typical thread state transitions in the PX5 RTOS:



Note that the FINISHED state is not shown. Once in the FINISHED state, the thread can never become ready or execute again, i.e., there are no transitions from the FINISHED state.

System Objects

The PX5 RTOS provides support for the most popular POSIX objects, including *condition variables*, *message queues*, *mutexes*, *semaphores*, *signals*, and *timers*. There is no compile-time limit on the number of system objects an application may have. The only limit is the amount of memory available in the application.

The PX5 RTOS does not create any objects itself, i.e., there are no hidden system threads or any other mutex. The only object created by the PX5 RTOS is the initial “main” thread, which ultimately is an application thread.

As mentioned previously, the memory required for object control blocks and associated memory areas is under complete control of the application.

Condition Variables

The POSIX pthreads condition variable is a communication object that has built-in synchronization via direct association with a mutex object. Threads can block on a condition variable (even with timeout) to wait for data or some state to be reached. When the data or state is reached, this can be signaled or broadcasted by another thread or threads. If a condition variable is signaled, the highest priority waiting thread is resumed. If the condition variable is broadcast, all threads waiting on the condition variable are resumed.

The following are the condition variable APIs supported by the PX5 RTOS:

pthread_cond_broadcast
pthread_cond_destroy
pthread_cond_init
pthread_cond_signal
pthread_cond_timedwaid
pthread_cond_wait
pthread_condattr_destroy
pthread_condattr_getcontroladdr
pthread_condattr_getcontrolsize
pthread_condattr_getname
pthread_condattr_getpshared
pthread_condattr_init
pthread_condattr_setcontroladdr
pthread_condattr_setname
pthread_condattr_setpshared

Message Queues

The POSIX message queues provide inter-thread communication. Messages passed between threads via message queues are passed by value, meaning messages are copied into the message queue when sending. Conversely, messages are copied from the message queue when receiving. The size of a message sent has an upper bound defined when the message queue was created (opened). The actual size of the message sent can be any size less than or equal to the maximum size – including a size of zero. When a message is received, the actual size of the message is provided as part of the receive API.

Messages also have priorities, ranging from 0 (lowest priority) through *_SC_MQ_PRIOMAX* (highest priority, default 31). Messages are placed in the queue in priority order. Messages of the same priority are placed in the queue in FIFO order.

Threads can suspend on either trying to receive from an empty queue or trying to send to a queue that is full. Suspension is determined by a queue attribute. If the queue attribute *O_NONBLOCK* is present, no blocking is allowed on the message queue. This attribute must be set via the *mq_setattr* API call, i.e., it is not the default upon queue creation. Threads that do suspend on a message queue, do so in FIFO order.

The following are the message queue APIs supported by the PX5 RTOS:

mq_close
mq_getattr
mq_open
mq_receive
mq_send
mq_setattr
mq_timedreceive
mq_timedsend
px5_mq_extendattr_destroy
px5_mq_extendattr_getcontroladdr
px5_mq_extendattr_getcontrolsize
px5_mq_extendattr_getqueueaddr
px5_mq_extendattr_getqueuesize
px5_mq_extendattr_init
px5_mq_extendattr_setcontroladdr
px5_mq_extendattr_setqueueaddr

Mutexes

The POSIX pthreads mutex is a thread synchronization object that is typically used for mutual exclusion—most often to protect a shared data structure from concurrent access by multiple threads. If a thread attempts to lock a mutex that is already owned, the thread suspends on the mutex waiting for it be become available. When the owning thread unlocks the mutex, the highest priority thread waiting to lock the mutex is given the mutex and resumed. Both *priority inheritance* and *recursive* mutexes are supported, however, neither is the default.

When using mutexes, it's important to avoid deadlock situations (each thread waiting for a mutex owned by the other thread). Allowing each thread to only lock one mutex at a time will avoid deadlocks. If multiple mutexes must be locked by the same thread, deadlock can be avoided if the mutexes are locked in the exact same order by all threads. Finally, threads must not exist with a locked mutex. Doing so will leave the mutex permanently locked and all threads attempting to lock the mutex will wait forever.

The following are the mutex APIs supported by the PX5 RTOS:

pthread_mutex_destroy
pthread_mutex_init
pthread_mutex_lock
pthread_mutex_trylock
pthread_mutex_unlock
pthread_mutexattr_destroy
pthread_mutexattr_getcontroladdr
pthread_mutexattr_getcontrolsize
pthread_mutexattr_getname
pthread_mutexattr_getpshared
pthread_mutexattr_init
pthread_mutexattr_setcontroladdr
pthread_mutexattr_setname
pthread_mutexattr_setpshared

Semaphores

POSIX semaphores are synchronization objects not part of the pthreads API per-se but are commonly found with pthread implementations. A semaphore is effectively a counter. Non-zero values indicate availability, while a zero value indicates the semaphore is not available. Threads block when attempting to get a semaphore with a value of zero. When the semaphore is released by another thread, the highest priority thread waiting on the semaphore is resumed.

A semaphore can be used in a consumer-producer fashion as a way to synchronize thread execution. Semaphores can also be used for mutual exclusion. However, mutexes are generally a more robust object for mutual exclusion (semaphores don’t have the concept of ownership and also don’t provide nesting or priority inheritance).

The following are the semaphore APIs supported by the PX5 RTOS:

sem_destroy
sem_init
sem_post
sem_trywait
sem_wait
px5_sem_extend_init
px5_semattr_destroy
px5_semattr_getcontroladdr
px5_semattr_getcontrolsize

px5_semattr_getname
px5_semattr_init
px5_semattr_setcontroladdr
px5_semattr_setname

Signals

POSIX signals provide a mechanism to notify a thread of a system event. The notification can be done synchronously or asynchronously, as determined by the receiving thread's signal mask setting. If a thread has a signal masked, the signal processing for that specific signal is synchronous and accomplished via the various signal wait API calls. However, if the thread has a signal unmasked, the signal processing for that signal is asynchronous, i.e., it happens immediately when the signal is raised.

For asynchronous signal processing, the application must have previously registered a signal handler for the signal via the *sigaction* API. This signal handler will execute immediately in the context of the selected thread when the corresponding signal is raised.

Each signal is represented by a single bit position of a 32-bit word. Multiple signals can be masked or unmasked or waited for simultaneously. Raising a signal is done via the *pthread_kill* API using the signal's numeric value and the responsible thread's handle.

The following are the signal APIs supported by the PX5 RTOS:

pthread_kill
pthread_sigmask
sigaction
sigaddset
sigdelset
sigemptyset
sigfillset
sigismember
sigpending
sigtimedwait
sigwait
sigwaitinfo

Pthread Extensions

The POSIX pthread API provides a rich set of multithreading services, which the PX5 RTOS makes available to the application. However, deeply embedded, real-time applications often require additional services. For this reason, the PX5 RTOS provides optional extensions to pthreads, which are collectively called pthreads+. These extensions to pthreads are completely optional and are easily identifiable by having the "*px5_*" prefix to the API name.

Fastqueues

The PX5 RTOS provides Fastqueue thread communication objects, which are not part of the pthreads API but are quite useful in embedded real-time programming. Each Fastqueue is capable of holding one or more fixed-sized messages (message sizes must be evenly divisible by sizeof the u_long data type. Threads may suspend on queue full or queue empty requests. Suspended threads are resumed in FIFO order. ISRs are allowed to send messages via the *px5_pthread_fastqueue_trysend* API.

The following are the Fastqueue APIs supported by the PX5 RTOS:

px5_pthread_lags_clear
px5_pthread_fastqueue_create
px5_pthread_fastqueue_destroy
px5_pthread_fastqueue_send
px5_pthread_fastqueue_receive
px5_pthread_fastqueue_trysend
px5_pthread_fastqueue_tryreceive
px5_pthread_fastqueueattr_getcontroladdr
px5_pthread_fastqueueattr_getcontrolsize
px5_pthread_fastqueueattr_getname
px5_pthread_fastqueueattr_getqueueaddr
px5_pthread_fastqueueattr_getqueuesize
px5_pthread_fastqueueattr_init
px5_pthread_fastqeueuattr_setcontroladdr
px5_pthread_fastqueueattr_setname
px5_pthread_fastqueueattr_setqueueaddr

Event Flags

The PX5 RTOS provides thread synchronization event flags objects, which are not part of the pthreads API but are quite useful in embedded real-time programming. Each event flag is represented by a single bit in a 32-bit word (maximum of 32 events per event flags group). Threads may wait for any one of the events they specify. Alternatively, threads may wait on the reception of all the events they specific. When a thread (or ISR) sets an event, all threads that have their event request satisfied are resumed. Events that satisfy a suspended thread's request are automatically cleared.

The following are the event flags APIs supported by the PX5 RTOS:

px5_pthread_event_flags_clear
px5_pthread_event_flags_create
px5_pthread_event_flags_destroy
px5_pthread_event_flag_set
px5_pthread_event_flags_trywait
px5_pthread_event_flags_wait
px5_pthread_event_flagsattr_getcontroladdr
px5_pthread_event_flagsattr_getcontrolsize
px5_pthread_event_flagsattr_getname
px5_pthread_event_flagsattr_init
px5_pthread_event_flagsattr_setcontroladdr
px5_pthread_event_flagsattr_setname

Memory Pools

The PX5 RTOS provides dynamic, variable-lenth memory allocation via memory pools objects, which are not part of the pthreads API but is often necessary in embedded real-time programming. Memory is allocated on a first-fit basis, i.e., the first memory encountered in the search that satisfies the request is used. If not enough memory is available in the memory pool, the calling thread will suspend. When memory becomes available, all suspended threads with a potential of a successful allocation are resumed.

The following are the memory pool APIs supported by the PX5 RTOS:

px5_pthread_memorypool_allocate
px5_pthread_memorypool_create
px5_pthread_memorypool_destroy
px5_pthread_memorypool_free
px5_pthread_memorypool_tryallocate
px5_pthread_memorypoolattr_destroy

px5_pthread_memorypoolattr_getcontroladdr
px5_pthread_memorypoolattr_getcontrolsize
px5_pthread_memorypoolattr_getname
px5_pthread_memorypoolattr_init
px5_pthread_memorypoolattr_setcontroladdr
px5_pthread_memorypoolattr_setname

Ticktimer Services

The PX5 RTOS provides ticktimer services (usually, the tick interval is 10ms, but this is completely under application control). Ticktimer services include retrieval of the current number of timer ticks, thread sleeping for a specified number of timer ticks, and tick-based application one-shot as well as periodic timers.

The following are the ticktimer APIs supported by the PX5 RTOS:

px5_pthread_tick_sleep
px5_pthread_ticks_get
px5_pthread_ticktimer_create
px5_pthread_ticktimer_destroy
px5_pthread_ticktimer_start
px5_pthread_ticktimer_stop
px5_pthread_ticktimer_update
px5_pthread_ticktimerattr_destroy
px5_pthread_ticktimerattr_getcontroladdr
px5_pthread_ticktimerattr_getcontrolsize
px5_pthread_ticktimerattr_init
px5_ticktimerattr_setcontroladdr
px5_pthread_ticktimerattr_setcontrolsize
px5_pthread_ticktimerattr_setname

Chapter
7

Chapter 7: PX5 RTOS API definitions

This chapter provides the API description of the PX5 RTOS. Each API definition includes the API's C prototype, a description including the various real-time scenarios that can occur as a result of the API, where the API can be called, a definition of each parameter, return value(s), and a small C code example.

All APIs starting with "*px5_*" are PX5 RTOS pthreads+ extensions. These APIs are PX5-specific and are not generally available on non-PX5 platforms.

clock_getres

C Prototype:

```
#include <pthread.h>

int  clock_getres(clockid_t clock_id, struct timespec *resolution);
```

Description:

This service returns the current time resolution in *resolution*, which is effectively the underlying ticktimer periodic interrupt frequency.

API Parameters:

`clock_id`	This parameter must be CLOCK_REALTIME, since it is the only supported clock.
`resolution`	Destination for the clock resolution.

Return Codes:

PX5_SUCCESS (0)	Successful retrieval of clock resolution.
PX5_ERROR (-1)	Error attempting to get clock resolution. Please use *errno* to retrieve the exact error:

EINVAL	Invalid clock ID.
EFAULT	Invalid resolution pointer.

Real-time Scenarios:

Upon the successful execution of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

clock_gettime, clock_settime, time

Small Example:

```
#include <pthread.h>


Int                 status;
struct timespec     my_resolution;




    /* Pickup the current clock resolution.  */
    status =  clock_getres(CLOCK_REALTIME, &my_resolution);

    /* If status contains PX5_SUCCESS (0), the clock resolution is
       in "my_resolution". */
```

clock_gettime

C Prototype:

```
#include <pthread.h>

int  clock_gettime(clockid_t clock_id, struct timespec *current_time);
```

Description:

This service returns the current time in the destination specified by *current_time*.

API Parameters:

`clock_id`	This parameter must be CLOCK_REALTIME, since it is the only supported clock.
`current_time`	Destination for the current time.

Return Codes:

PX5_SUCCESS (0)	Successful get time.
PX5_ERROR (-1)	Error attempting to get current time. Please use *errno* to retrieve the exact error:

EINVAL	Invalid clock ID.
EFAULT	Invalid current time pointer.

Real-time Scenarios:

Upon the successful execution of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

clock_getres, clock_settime, time

Small Example:

```
#include <pthread.h>


Int                 status;
struct timespec     my_current_time;



    /* Pickup the current time.  */
    status =  clock_gettime(CLOCK_REALTIME, &my_current_time);

    /* If status contains PX5_SUCCESS (0), the current time is
       in "my_current_time". */
```

clock_settime

C Prototype:

```
#include <pthread.h>

int  clock_settime(clockid_t clock_id, struct timespec *new_time);
```

Description:

This service sets the current time to the value specified by *new_time*.

API Parameters:

`clock_id`	This parameter must be CLOCK_REALTIME, since it is the only supported clock.
`new_time`	User supplied time value the system is set to.

Return Codes:

PX5_SUCCESS (0)	Successful set time.
PX5_ERROR (-1)	Error attempting to set time. Please use *errno* to retrieve the exact error:

EINVAL	Invalid clock ID or invalid time specification.
EFAULT	Invalid new time pointer.

Real-time Scenarios:

Upon the successful execution of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

clock_getres, clock_gettime, time

Small Example:

```
#include <pthread.h>


Int                status;
struct timespec    my_new_time;



    /* Get the current time.  */
    clock_gettime(CLOCK_REALTIME, &my_new_time);

    /* Move one second forward.  */
    my_new_time.tv_sec++;

    /* Set the new time.  */
    status =  clock_settime(CLOCK_REALTIME, &my_new_time);

    /* If status contains PX5_SUCCESS (0), the new time is one second
       later. */
```

mq_close

C Prototype:

```
#include <mqueue.h>

int  mq_close(mqd_t message_queue);
```

Description:

This service closes and destroys the specified message queue. If there are any threads suspended on the message queue, an error is returned.

API Parameters:

`message_queue`	Specifies the queue to close/destroy.

Return Codes:

PX5_SUCCESS (0)	Successful message queue close.
PX5_ERROR (-1)	Error attempting to close the message queue. Please use *errno* to retrieve the exact error:

EBADF	Invalid memory queue handle.
EBUSY	Specified message queue has threads suspended on it.

Real-time Scenarios:

Upon the successful execution of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

mq_open, mq_receive, mq_send

Small Example:

```
#include <mqueue.h>


int                status;
mqd_t              my_queue_handle;



    /* Close the previously opened queue “my_queue_handle”.  */
    status =  mq_close(my_queue_handle);

    /* If status contains PX5_SUCCESS (0), the message queue
       “my_queue_handle” is closed. */
```

mq_getattr

C Prototype:

```
#include <mqueue.h>

int  mq_getattr(mqd_t message_queue, struct mq_attr* queue_attributes);
```

Description:

This service retrieves the current attributes of the specified message queue.

API Parameters:

`message_queue`	Specifies the queue to retrieve attributes from.
`queue_attributes`	Specifies the destination for the queue attributes information.

Return Codes:

PX5_SUCCESS (0)	Successful message queue attributes retrieval.
PX5_ERROR (-1)	Error attempting to get the queue attributes. Please use *errno* to retrieve the exact error:

EBADF	Invalid memory queue handle or invalid attributes destination pointer.

Real-time Scenarios:

Upon the successful execution of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

mq_open, mq_setattr

Small Example:

```
#include <mqueue.h>


int                 status;
struct mqd_t        my_queue_attributes;
mqd_t               my_queue_handle;



    /* Get the attributes of the previously opened queue
       "my_queue_handle" and return them in "my_queue_attributes".  */
    status =  mq_getattr(my_queue_handle, &my_queue_attributes);

    /* If status contains PX5_SUCCESS (0), "my_queue_attributes"
       contains the attributes of message queue "my_queue_handle". */
```

mq_open

C Prototype:

```
#include <fcntl.h>
#include <sys/stat.h>
#include <mqueue.h>

mqd_t  mq_open(const char * queue_name, int operation, mode_t mode,
                                 mq_attr * queue_attributes);
```

Description:

This service opens (creates) the message queue specified and returns the message queue handle, if successful.

API Parameters:

queue_name	Name of the queue to open/create.
operation	Specifies how the queue will operate. The supported options are: O_CREAT O_RDWR O_NONBLOCK Both O_CREAT and O_RDWR should be specified. The O_NONBLOCK option that disables threads from suspending on the queue is optional.
mode	Not currently used.
queue_attributes	Attributes that specify the dimensions of the message queue, as defined by these structure members: mq_maxmsg mq_msgsize Where *mq_maxmsg* defines the total number of messages the queue can hold. The maximum size of each message (in bytes) is defined by *mq_msgsize*. Note the size and

	priority of each message must be stored along with the message content. In addition, there is one pointer type required for each message. On most architectures, this amounts to 12 bytes of additional-per message overhead.

Return Codes:

queue handle	Positive value represents the successfully opened (created) queue handle.
PX5_ERROR (-1)	Error attempting to open/create the queue. Please use *errno* to retrieve the exact error:

ENOSPC	Not enough memory to create the specified queue.
EINVAL	Invalid operation or invalid attributes pointer.

Real-time Scenarios:

Upon the successful execution of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

mq_close, mq_getattr, mq_setattr, px5_mq_extend_open

Small Example:

```
#include <fcntl.h>
#include <sys/stat.h>
#include <mqueue.h>


int                 status;
struct mq_attr      my_queue_attributes;
mqd_t               my_queue_handle;


    /* Setup the attributes for 100 total messages, where each
       message is a maximum of 16 bytes.  */
    my_queue_attributes.mq_maxmsg =  100;
    my_queue_attributes.mq_msgsize = 16;

    /* Open (create) the queue “my_queue”. */
    my_queue_handle =  mq_open(“my queue”, (O_CREAT | O_RDWR), 0,
                                          &my_queue_attributes);

    /* If “my_queue_handle” is positive, the queue was successfully
       created. */
```

mq_receive

C Prototype:

```
#include <mqueue.h>

int  mq_receive(mqd_t message_queue, char *  message,
            size_t  message_size, unsigned int *  message_priority);
```

Description:

This service receives the highest priority message from the specified message queue. If the queue is empty and the O_NONBLOCK attribute is not set, the calling thread will suspend waiting for a message to arrive. If multiple threads are waiting on an empty queue, the first thread waiting is given the message.

This API is a cancellation point, meaning that if a cancellation is pending, it will be detected and executed by this API.

API Parameters:

`message_queue`	Message queue to get message from.
`message`	Pointer to the destination for the message.
`message_size`	Maximum size for the message. This must be equal to or greater than the maximum message size of the queue.
`message_priority`	Priority of the message received.

Return Codes:

message size	A non-negative value represents the actual size of the successfully received message (messages of zero size are allowed).
PX5_ERROR (-1)	Error attempting to receive a message from the queue. Please use *errno* to retrieve the exact error:

EBADF	Invalid message queue handle or invalid message destination pointer.
EMSGSIZE	Invalid message size.

EAGAIN	Message queue is empty and O_NONBLOCK was specified to disable thread suspension to wait for a message.

Real-time Scenarios:

Upon the successful execution of this service, the following real-time scenarios are possible:

NO PREEMPTION. There is no preemption possible with this service if a message was present in the message queue.

SUSPENSION. The calling thread is suspended until a message arrives in the queue.

PREEMPTION. If a higher-priority thread was waiting to place a message on the queue, it is resumed, and preemption will occur.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

mq_open, mq_send, mq_timedreceive

Small Example:

```
#include <mqueue.h>


ssize_t           received_size;
char              my_message[16];
unsigned int      my_priority;
mqd_t             my_queue_handle;


    /* Receive message from “my_queue”. */
    received_size =  mq_receive(my_queue_handle, &my_message[0],
                           sizeof(my_message), &my_priority);

    /* If “received_size” is positive, the message was successfully
       received. */
```

mq_send

C Prototype:

```
#include <mqueue.h>

int  mq_send(mqd_t message_queue, char *  message,
             size_t  message_size, unsigned int  message_priority);
```

Description:

This service send the message to the specified message queue. If the queue is full and the O_NONBLOCK attribute is not set, the calling thread will suspend waiting for room in the queue.

This API is a cancellation point, meaning that if a cancellation is pending, it will be detected and executed by this API.

API Parameters:

`message_queue`	Message queue to send message to.
`message`	Pointer to the source of the message.
`message_size`	Size of the message. This must be equal to or less than the maximum message size of the queue. Messages of zero size are allowed.
`message_priority`	Priority of the message to send.

Return Codes:

PX5_SUCCESS (0)	Successful send to the message queue.
PX5_ERROR (-1)	Error attempting to send a message to the queue. Please use *errno* to retrieve the exact error:

EBADF	Invalid message queue handle or invalid message pointer.
EMSGSIZE	Invalid message size.
EAGAIN	Message queue is full and O_NONBLOCK was specified to

disable thread suspension to wait for room in the queue.

Real-time Scenarios:

Upon the successful execution of this service, the following real-time scenarios are possible:

NO PREEMPTION. There is no preemption possible with this service if there is room in the message queue.

SUSPENSION. The calling thread is suspended if the specified queue is full.

PREEMPTION. If a higher-priority thread was waiting for a message on an empty queue, it is resumed, and preemption will occur.

Callable From:

This service is only callable from the thread context and from interrupt handlers (ISRs).

The user must not call this API from an interrupt handler if suspension is possible. An easy way to ensure the queue attribute O_NONBLOCK is in force. Otherwise, unpredictable behavior is possible.

See Also:

mq_open, mq_receive, mq_timedreceive, mq_timedsend

Small Example:

```
#include <mqueue.h>


int                 status;
char                my_message[16];
mqd_t               my_queue_handle;


    /* Build the message.  */
    my_message[0] =  “m”;
    my_message[1] =  “y”;
    my_message[2] =  “ “;
    my_message[3] =  “m”;
    my_message[4} =  “e”;
    my_message[5] =  “s”;
    my_message[6] =  “s”;
    my_message[7] =  “a“;
    my_message[8] =  “g”;
    my_message[9} =  “e”;

    /* Send a priority 0 message to “my_queue”. */
    status =  mq_send(my_queue_handle, &my_message[0], 10, 0);

    /* If status is PX5_SUCCESS (0), the message was successfully
       sent. */
```

mq_setattr

C Prototype:

```
#include <mqueue.h>

int  mq_setattr(mqd_t message_queue, struct mq_attr * new_attributes,
                           struct mq_attr * previous_attributes);
```

Description:

This service sets the attributes of the specified message queue. If the *previous_attributes* pointer is non-NULL, the attributes prior to this service are returned.

API Parameters:

`message_queue`	Specifies the queue to retrieve attributes from.
`new_attributes`	Specifies the new queue attributes to set, either O_NONBLOCK to disable thread suspension on the queue or zero to enable thread suspension.
`previous_attributes`	If non-NULL, specifies the destination for the prior queue attributes information.

Return Codes:

PX5_SUCCESS (0)	Successful message queue attributes set.
PX5_ERROR (-1)	Error attempting to set the queue attributes. Please use *errno* to retrieve the exact error:

EBADF	Invalid memory queue handle or invalid attributes destination pointer.
EINVAL	Invalid attribute specification – must be either O_NONBLOCK or zero.

Real-time Scenarios:

Upon the successful execution of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

mq_getattr, mq_open

Small Example:

```
#include <mqueue.h>


int                  status;
struct mq_attr       my_new_attributes;
struct mq_attr       my_previous_attributes;
mqd_t                my_queue_handle;



    /* Turn off thread suspension on "my_queue_handle" and return the
       previous attributes in "my_previous_attributes".  */
    my_new_attributes.mq_flags =  O_NONBLOCK;
    status =  mq_setattr(my_queue_handle, &my_new_attributes,
                                    &my_previous_attributes);

    /* If status contains PX5_SUCCESS (0), the queue won't allow thread
       suspension and "my_previous_attributes" contains the attributes
       of message queue "my_queue_handle" before this call. */
```

mq_timedreceive

C Prototype:

```
#include <pthread.h>
#include <mqueue.h>

int  mq_timedreceive(mqd_t message_queue, char *  message,
             size_t  message_size, unsigned int *  message_priority,
             const struct timespec absolute_timeout);
```

Description:

This service receives the highest priority message from the specified message queue. If the queue is empty and the O_NONBLOCK attribute is not set, the calling thread will suspend waiting for a message to arrive. If the timeout is exceeded before a message arrives, the thread is resumed with an error.

This API is a cancellation point, meaning that if a cancellation is pending, it will be detected and executed by this API.

API Parameters:

`message_queue`	Message queue to get message from.
`message`	Pointer to the destination for the message.
`message_size`	Maximum size for the message. This must be equal to or greater than the maximum message size of the queue.
`message_priority`	Priority of the message received.
`absolute_timeout`	Absolute time to wait for the message to arrive.

Return Codes:

message size	A non-negative value represents the actual size of the successfully received message (messages of zero size are allowed).
PX5_ERROR (-1)	Error attempting to receive a message from the queue. Please use *errno* to retrieve the exact error:

EBADF	Invalid message queue handle or invalid message destination pointer.
EMSGSIZE	Invalid message size.
EAGAIN	Message queue is empty and O_NONBLOCK was specified to disable thread suspension to wait for a message.
ETIMEDOUT	Timeout on thread suspension waiting for a message.

Real-time Scenarios:

Upon the successful execution of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. There is no preemption possible with this service if a message was present in the message queue.

S **SUSPENSION**. The calling thread is suspended until a message arrives in the queue.

P **PREEMPTION**. If a higher-priority thread was waiting to place a message on the queue, it is resumed, and preemption will occur.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

mq_open, mq_send, mq_receive, mq_timedreceive

Small Example:

```
#include <mqueue.h>


ssize_t            received_size;
char               my_message[16];
unsigned int       my_priority;
mqd_t              my_queue_handle;
struct timespec    my_absolute_timeout;


    /* Setup a timeout for one second in the future.  */
    clock_gettime(CLOCK_REALTIME, &my_absolute_time);
    my_absolute_time.tv_sec += 1;

    /* Receive message from “my_queue” but only wait for 1 second. */
    received_size =  mq_timedreceive(my_queue_handle, &my_message[0],
                           sizeof(my_message), &my_priority,
                           &my_absolute_timeout);

    /* If “received_size” is positive, the message was successfully
       received. */
```

mq_timedsend

C Prototype:

```
#include <mqueue.h>

int  mq_timedsend(mqd_t message_queue, char *  message,
             size_t  message_size, unsigned int  message_priority,
             const struct timespec absolute_timeout);
```

Description:

This service send the message to the specified message queue. If the queue is full and the O_NONBLOCK attribute is not set, the calling thread will suspend waiting for room in the queue. If the timeout is exceeded before room for the message is available in the queue, the thread is resumed with an error.

This API is a cancellation point, meaning that if a cancellation is pending, it will be detected and executed by this API.

API Parameters:

`message_queue`	Message queue to send message to.
`message`	Pointer to the source of the message.
`message_size`	Size of the message. This must be equal to or less than the maximum message size of the queue. Messages with zero size are allowed.
`message_priority`	Priority of the message to send.
`absolute_timeout`	Absolute time to wait for room for the message in the queue.

Return Codes:

PX5_SUCCESS (0)	Successful send to the message queue.
PX5_ERROR (-1)	Error attempting to send a message to the queue. Please use *errno* to retrieve the exact error:

EBADF	Invalid message queue handle or invalid message pointer.
EMSGSIZE	Invalid message size.
EAGAIN	Message queue is full and O_NONBLOCK was specified to disable thread suspension to wait for room in the queue.
ETIMEDOUT	Timeout on thread suspension waiting for room for the message in the queue.

Real-time Scenarios:

Upon the successful execution of this service, the following real-time scenarios are possible:

NO PREEMPTION. There is no preemption possible with this service if there is room in the message queue.

SUSPENSION. The calling thread is suspended if the specified queue is full.

PREEMPTION. If a higher-priority thread was waiting for a message on an empty queue, it is resumed, and preemption will occur.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

mq_open, mq_receive, mq_send, mq_timedreceive

Small Example:

```
#include <mqueue.h>


ssize_t           received_size;
char              my_message[16];
unsigned int      my_priority;
mqd_t             my_queue_handle;
struct timespec   my_absolute_timeout;


    /* Setup a timeout for one second in the future.  */
    clock_gettime(CLOCK_REALTIME, &my_absolute_time);
    my_absolute_time.tv_sec += 1;

    /* Send message to "my_queue" but only wait for 1 second. */
    status =  mq_timedsend(my_queue_handle, &my_message[0],
                    sizeof(my_message), my_priority,
                           &my_absolute_timeout);

    /* If "status" is PX5_SUCCESS, the message was successfully
       sent. */
```

nanosleep

C Prototype:

```
#include <pthread.h>

int  nanosleep(struct timespec *request_time,
                              struct timespec *remainng_time);
```

Description:

This service causes the calling thread to suspend for the amount of time specified in *request_time*. If an unmasked signal is sent to the thread while sleeping, the thread is resumed, and the amount of remaining time is returned in *remaining_time*.

ⓘ *Nanosleep requests are rounded up to the next time that is evenly divisible by the timer resolution.*

ⓘ *This API is a cancellation point, meaning that if a cancellation is pending, it will be detected and executed by this API.*

API Parameters:

`request_time`	The amount of time to sleep.
`remaining_time`	If non-NULL, the destination for the amount of remaining time to sleep if nanosleep was interrupted by a signal.

Return Codes:

PX5_SUCCESS (0)	Successful sleep.
PX5_ERROR (-1)	Error attempting to sleep. Please use *errno* to retrieve the exact error:

EINVAL	Invalid request pointer or invalid request time.

EINTR	Nanosleep was interrupted by a signal. The time left to sleep is returned in *remaining_time*.

Real-time Scenarios:

Upon the successful execution of this service, the following real-time scenarios are possible:

S **SUSPENSION**. The calling thread is suspended until the time specified has lapsed or until another thread sends a signal to this thread.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

px5_pthread_tick_sleep, sleep, usleep

Small Example:

```
#include <pthread.h>


int                  status;
struct timespec   my_sleep_time;
struct timespec   my_remaining_time;



    /* Sleep for 1 second. */
    my_sleep_time.tv_sec =   1;
    my_sleep_time.tv_nsec =  0;

    status =  nanosleep(&my_sleep_time, &my_remaining_time);

    /* If status contains PX5_SUCCESS (0), the calling thread slept for
       1 second. */
```

pthread_attr_destroy

C Prototype:

```
#include <pthread.h>

int  pthread_attr_destroy(pthread_attr_t *attributes);
```

Description:

This service destroys a previously initialized thread creation attributes structure. Once destroyed, the attributes structure can no longer be used.

The destruction of thread attributes doesn't have any effect on threads previously created with the attributes. The only effect is that this attributes structure cannot be used in the creation of any additional threads.

API Parameters:

`attributes`	Pointer to a previously initialized attributes structure.

Return Codes:

PX5_SUCCESS (0)	Successful attributes structure destruction.
EINVAL	Attributes structure pointer is invalid.

Real-time Scenarios:

Upon the successful execution of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

pthread_attr_*, pthread_create

Small Example:

```
#include <pthread.h>


Int                 status;
pthread_attr_t      my_thread_attributes;





    /* Destroy the previously initialized attributes.  */
    status =  pthread_attr_destroy(&my_thread_attributes);

    /* If status contains PX5_SUCCESS, the attributes
       have been destroyed. */
```

pthread_attr_getdetachstate

C Prototype:

```
#include <pthread.h>

int  pthread_attr_getdetachstate(pthread_attr_t *attributes,
                                 int *  detach_state);
```

Description:

This service returns the detach state stored in the thread attributes structure. Valid detach states are *PTHREAD_CREATE_JOINABLE* (default) and *PTHREAD_CREATE_DETACHED*.

API Parameters:

`attributes`	Pointer to a previously initialized attributes structure.
`detach_state`	Pointer to the destination of where to return the detach state setting in this attributes structure.

Return Codes:

PX5_SUCCESS (0)	Successful retrieval of the attributes detach state.
EINVAL	Attributes structure pointer or detach state destination pointer is invalid.

Real-time Scenarios:

Upon the successful execution of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

pthread_attr_*, pthread_create

Small Example:

```
#include <pthread.h>

pthread_attr_t    my_thread_attributes;
int               detach_state;
int               status;




    /* Get the detach state in the previously initialized
       attributes.  */
    status =  pthread_attr_getdetachstate(&my_thread_attributes,
                                          &detach_state);

    /* If status contains PX5_SUCCESS, the detach state value in this
       attribute structure is contained in "detach_state". */
```

pthread_attr_getstackaddr

C Prototype:

```
#include <pthread.h>

int  pthread_attr_getstackaddr(pthread_attr_t *attributes,
                                         void **  stack_address);
```

Description:

This service returns the stack address stored in the thread attributes structure. By default, this value is NULL unless specified by the application via a call to *pthread_attr_setstackaddr*.

API Parameters:

`attributes`	Pointer to a previously initialized attributes structure.
`stack_address`	Pointer to the destination of where to return the stack address of this attributes structure.

Return Codes:

PX5_SUCCESS (0)	Successful retrieval of the stack address.
EINVAL	Attributes structure pointer or stack address destination pointer is invalid.

Real-time Scenarios:

Upon the successful execution of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

pthread_attr_*, pthread_create

Small Example:

```
#include <pthread.h>

pthread_attr_t    my_thread_attributes;
void *            stack_address;
int               status;



    /* Get the stack address in the previously initialized
       attributes.  */
    status =  pthread_attr_getstackaddr(&my_thread_attributes,
                                        &stack_address);

    /* If status contains PX5_SUCCESS, the stack address value in this
       attribute structure is contained in "stack_address". */
```

pthread_attr_getstacksize

C Prototype:

```
#include <pthread.h>

int  pthread_attr_getstacksize(pthread_attr_t *attributes,
                                   size_t *     stack_size);
```

Description:

This service returns the current stack size stored in the thread attributes structure. By default, this value is *PX5_DEFAULT_STACK_SIZE* unless specified by the application via a call to *pthread_attr_setstacksize*.

API Parameters:

`attributes`	Pointer to a previously initialized attributes structure.
`stack_size`	Pointer to the destination of where to return the stack size of this attributes structure.

Return Codes:

PX5_SUCCESS (0)	Successful retrieval of the stack size.
EINVAL	Attributes structure pointer or stack size destination pointer is invalid.

Real-time Scenarios:

Upon the successful execution of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

pthread_attr_*, pthread_create

Small Example:

```
#include <pthread.h>

pthread_attr_t    my_thread_attributes;
size_t            stack_size;
int               status;




    /* Get the stack size in the previously initialized
       attributes.  */
    status =  pthread_attr_getstacksize(&my_thread_attributes,
                                         &stack_size);

    /* If status contains PX5_SUCCESS, the stack size value in this
       attribute structure is contained in "stack_size". */
```

pthread_attr_init

C Prototype:

```
#include <pthread.h>

int  pthread_attr_init(pthread_attr_t *attributes);
```

Description:

This service initializes the attributes structure with default thread creation values. These defaults are as follows:

Attribute	**Default Setting**
Detach State	Joinable – *PTHREAD_CREATE_JOINABLE*
Stack Address	NULL – Allocate dynamically
Stack Size	*PX5_DEFAULT_STACK_SIZE*
Priority	*PX5_DEFAULT_PRIORITY*
Thread Control Address	NULL – Allocate dynamically
Thread Control Size	Size of internal thread control structure
Time Slice	No time-slice (0)

Once attributes are initialized via this service, the default settings above can be overridden by *pthread_attr_set** API calls.

Note that the attributes are only relevant when supplied to the pthread_create API. After the thread is created, the attributes can be changed for the next thread without any effect on previously created threads.

API Parameters:

`attributes`	Pointer to a previously initialized attributes structure.

Return Codes:

PX5_SUCCESS (0)	Successful retrieval of the stack size.
EINVAL	Attributes structure pointer is invalid.

Real-time Scenarios:

Upon the successful execution of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

pthread_attr_*, pthread_create

Small Example:

```
#include <pthread.h>

pthread_attr_t    my_thread_attributes;
int               status;



    /* Initialize the thread creation attributes. */
    status =  pthread_attr_init(&my_thread_attributes);

    /* If status contains PX5_SUCCESS, the "my_thread_attributes"
       structure has been initialized with default values. */
```

pthread_attr_setdetachstate

C Prototype:

```
#include <pthread.h>

int  pthread_attr_setdetachstate(pthread_attr_t *attributes,
                                 int    new_detach_state);
```

Description:

This service sets the detach state specified by *new_detach_state* in the thread attributes structure. Valid detach states are *PTHREAD_CREATE_JOINABLE* and *PTHREAD_CREATE_DETACHED*.

Note that threads created with PTHREAD_CREATE_DETACHED cannot be joined or canceled. When they exit, all of their resources are released.

API Parameters:

`attributes`	Pointer to a previously initialized attributes structure.
`detach_state`	The new detach state setting for this attributes structure.

Return Codes:

PX5_SUCCESS (0)	Successful detach state set.
EINVAL	Attributes structure pointer or detach state specification.

Real-time Scenarios:

Upon the successful execution of this service, the following real-time scenarios are possible:

NO PREEMPTION. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

pthread_attr_*, pthread_create

Small Example:

```
#include <pthread.h>

pthread_attr_t    my_thread_attributes;
int               status;



    /* Set the detach state in the previously initialized
        attributes such that threads are created as detached.  */
    status =  pthread_attr_setdetachstate(&my_thread_attributes,
                                           PTHREAD_CREATE_DETACHED);

    /* If status contains PX5_SUCCESS, the detach state in this
       attributes structure is set to PTHREAD_CREATE_DETACHED. */
```

pthread_attr_setstackaddr

C Prototype:

```
#include <pthread.h>

int  pthread_attr_setstackaddr(pthread_attr_t *attributes,
                               void *    stack_address);
```

Description:

This service sets the stack address to the value specified by *stack_address*. The stack address must be at least *PTHREAD_STACK_MIN* number of bytes.

Note that each thread created must have its own unique stack memory. Hence, the stack address supplied here is only valid for one pthread_create call.

API Parameters:

`attributes`	Pointer to a previously initialized attributes structure.
`stack_address`	Stack address to use for the next thread creation.

Return Codes:

PX5_SUCCESS (0)	Successful setting of the stack address.
EINVAL	Attributes structure pointer or stack address pointer is invalid.

Real-time Scenarios:

Upon the successful execution of this service, the following real-time scenarios are possible:

NO PREEMPTION. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

pthread_attr_*, pthread_create

Small Example:

```
#include <pthread.h>

pthread_attr_t    my_thread_attributes;
int               status;



    /* Set the stack address to 0x30000 in the previously initialized
        attributes.  */
    status =  pthread_attr_setstackaddr(&my_thread_attributes,
                                           (void *) 0x30000);

    /* If status contains PX5_SUCCESS, the stack address in this
       attributes structure is set to 0x30000. */
```

pthread_attr_setstacksize

C Prototype:

```
#include <pthread.h>

int  pthread_attr_setstacksize(pthread_attr_t *attributes,
                                  size_t       stack_size);
```

Description:

This service sets the specified stack size in the thread attributes structure. The stack address must be at least *PTHREAD_STACK_MIN* number of bytes.

API Parameters:

`attributes`	Pointer to a previously initialized attributes structure.
`stack_size`	Stack size to set in the attributes structure.

Return Codes:

PX5_SUCCESS (0)	Successful setting of the stack size.
EINVAL	Attributes structure pointer or stack size is invalid.

Real-time Scenarios:

Upon the successful execution of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

pthread_attr_*, pthread_create

Small Example:

```
#include <pthread.h>

pthread_attr_t    my_thread_attributes;
int               status;



    /* Set the stack size in the previously initialized
       attributes.  */
    status =  pthread_attr_setstacksize(&my_thread_attributes,
                                         1024);

    /* If status contains PX5_SUCCESS, the stack size value of 1024
       is placed in this attributes structure. */
```

pthread_cancel

C Prototype:

```
#include <pthread.h>

int  pthread_cancel(pthread_t  thread_handle);
```

Description:

This service cancels the specified thread. If the specified thread has cancelation disabled or deferred, this service simply marks the thread for cancellation for at a later point determined by the thread itself. Otherwise, if the specified thread has asynchronous cancellation enabled, it is immediately canceled (terminated) by this service.

By default, threads are created with deferred cancellation enabled. This can be changed dynamically via the pthread_setcancelstate and pthread_setcanceltype APIs.

API Parameters:

`thread_handle`	Handle of thread to cancel.

Return Codes:

PX5_SUCCESS (0)	Successful thread cancellation.
ESRCH	Specified thread handle is invalid.

Real-time Scenarios:

Upon the successful execution of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. No preemption takes place as a result of this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

pthread_cleanup_pop, pthread_cleanup_push, pthread_create, pthread_setcancelstate, pthread_setcanceltype, pthread_testcancel

Small Example:

```
#include <pthread.h>


pthread_t  my_cancel_thread;
int        status;


    /* Cancel “my_cancel_thread”. */
    status =  pthread_cancel(my_cancel_thread);

    /* If “status” is PX5_SUCCESS (0), “my_cancel_thread” was
       canceled. */
```

pthread_cleanup_pop

C Prototype:

```
#include <pthread.h>

void  pthread_cleanup_pop(int execute);
```

Description:

This service pops the most recently pushed cleanup handler from the thread's cleanup handler stack. If the specified *execute* parameter is non-zero and the cleanup handler function pointer is non-NULL, the cleanup handler is executed before returning to the caller.

API Parameters:

`execute`	If non-zero and cleanup handler function pointer is non-NULL, the cleanup handler is executed.

Return Codes:

None

Real-time Scenarios:

Upon the successful execution of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. No preemption takes place as a result of this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

pthread_cancel, pthread_cleanup_push, pthread_create, pthread_setcancelstate, pthread_setcanceltype, pthread_testcancel

Small Example:

```
#include <pthread.h>


    /* Pop and execute the most recently pushed cleanup
       handler for the calling thread. */
    pthread_cleanup_pop(1);

    /* At this point the cleanup handler was popped and executed. */
```

pthread_cleanup_push

C Prototype:

```
#include <pthread.h>

void  pthread_cleanup_push(void (*cleanup_handler)(void *),
                                         void *argument);
```

Description:

This service pushes the specified *cleanup_handler* on the calling thread's cleanup handler stack.

Note the number of cleanup handlers a thread can push is determined by the PX5_MAXIMUM_CLEANUP_HANDLERS define that by default is 3. This can be changed via user configuration. If the limit is reached, the push request is silently discarded.

API Parameters:

`cleanup_handler`	Cleanup handler function pointer to push.
`argument`	Argument pointer to supply to the cleanup handler if it is called later.

Return Codes:

None

Real-time Scenarios:

Upon the successful execution of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. No preemption takes place as a result of this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

pthread_cancel, pthread_cleanup_pop, pthread_create, pthread_setcancelstate, pthread_setcanceltype, pthread_testcancel

Small Example:

```
#include <pthread.h>

    void my_cleanup_handler(void *agument)
    {
       /* Cleanup handler processing goes here!  */
    }

    /* Push "my_cleanup_handler" to the thread's cleanup handler
       stack. */
    pthread_cleanup_push(my_cleanup_handler, NULL);

    /* At this point, the cleanup handler was pushed. */
```

pthread_cond_broadcast

C Prototype:

```
#include <pthread.h>

int pthread_cond_broadcast(pthread_cond_t *  condition_var_handle);
```

Description:

This service resumes all threads currently waiting on this condition variable.

Note that the calling thread must own the mutex associated with this condition variable prior to making this call. Unpredictable behavior can result if the associated mutex is not owned by the caller of this service.

API Parameters:

`condition_var_handle`	Handle of the condition variable to broadcast to.

Return Codes:

PX5_SUCCESS (0)	Successful condition variable broadcast.
EINVAL	Invalid condition variable handle pointer.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. If there are no other threads waiting for the condition variable, no preemption takes place.

P **PREEMPTION**. If a higher-priority thread was waiting on the condition variable, when it is resumed, preemption will occur.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

pthread_cond_*, pthread_cond_init, pthread_cond_signal, pthread_cond_wait

Small Example:

```
#include <pthread.h>

/* Condition variable handle.  */
pthread_cond_t     my_condition_var_handle;
int                status;



    /* Broadcast to the condition variable
       "my_condition_var_handle". */
    status =  pthread_cond_broadcast(&my_condition_var_handle);

    /* If status is PX5_SUCCESS, the broadcast has resumed all waiting
       threads on this condition variable. */
```

pthread_cond_destroy

C Prototype:

```
#include <pthread.h>

int pthread_cond_destroy(pthread_cond_t *  condition_var_handle);
```

Description:

This service destroys the previously created condition variable specified by *condition_var_handle*. If the condition variable is still in use by another thread, an error is returned.

API Parameters:

`condition_var_handle`	Handle of the condition variable to destroy.

Return Codes:

PX5_SUCCESS (0)	Successful condition variable destroy.
EINVAL	Invalid condition variable handle.
EBUSY	Another thread is currently using the condition variable.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

pthread_cond_*, pthread_cond_init

Small Example:

```
#include <pthread.h>

/* Condition variable handle.  */
pthread_cond_t     my_condition_var_handle;
int                status;




    /* Destroy the condition variable referenced by
       "my_condition_var_handle". */
    status =  pthread_cond_destroy(&my_condition_var_handle);

    /* If status is PX5_SUCCESS, the condition variable was
       destroyed.  */
```

pthread_cond_init

C Prototype:

```
#include <pthread.h>

int pthread_cond_init(pthread_cond_t *  condition_var_handle,
                      pthread_condattr_t *  condition_var_attributes);
```

Description:

This service initializes (creates) a condition variable with the optional condition variable attributes. If successful, the condition variable handle is setup for further use by the application.

API Parameters:

`condition_var_handle`	Handle of the condition variable to initialize.
`condition_var_attributes`	Optional condition variable attributes. This value is NULL if no condition variable attributes are specified.

Return Codes:

PX5_SUCCESS (0)	Successful condition variable initialization.
EINVAL	Invalid condition variable handle pointer or condition variable attributes.
EBUSY	Condition variable is already created.
ENOMEM	Insufficient memory to create condition variable.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

pthread_condattr_*,pthread_cond_destroy

Small Example:

```
#include <pthread.h>

/* Condition variable handle.  */
pthread_cond_t     my_condition_var_handle;
int                status;




    /* Create the condition variable and setup
       "my_condition_var_handle". */
    status =  pthread_cond_init(&my_condition_var_handle, NULL);

    /* If status is PX5_SUCCESS, the condition variable was created and
       the condition variable handle is ready to use.  */
```

pthread_cond_signal

C Prototype:

```
#include <pthread.h>

int pthread_cond_signal(pthread_cond_t *  condition_var_handle);
```

Description:

This service resumes the highest priority thread currently waiting on this condition variable.

Note that the calling thread must own the mutex associated with this condition variable prior to making this call. Unpredictable behavior can result if the associated mutex is not owned by the caller of this service.

API Parameters:

`condition_var_handle`	Handle of the condition variable to signal.

Return Codes:

PX5_SUCCESS (0)	Successful condition variable signal.
EINVAL	Invalid condition variable handle pointer.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. If there are no other threads waiting for the condition variable, no preemption takes place.

P **PREEMPTION**. If a higher-priority thread was waiting on the condition variable, when it is resumed, preemption will occur.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

pthread_cond_*, pthread_cond_broadcast, pthread_cond_wait

Small Example:

```
#include <pthread.h>

/* Condition variable handle.  */
pthread_cond_t     my_condition_var_handle;
int                status;



    /* Signal the condition variable
       "my_condition_var_handle". */
    status =  pthread_cond_signal(&my_condition_var_handle);

    /* If status is PX5_SUCCESS, the highest priority waiting thread on
       this condition variable has been resumed. */
```

pthread_cond_timedwait

C Prototype:

```
#include <pthread.h>

int pthread_cond_timedwait(pthread_cond_t *  condition_var_handle,
                           pthread_mutex_t *  mutex_handle,
                           const struct timespec *absolute_time);
```

Description:

This services suspends on the condition variable specified by the *condition_var_handle* parameter. Before this service is called, the thread must have obtained the mutex specified by *mutex_handle*. Internally, PX5 releases the mutex atomically with the thread suspension. Once the calling thread is resumed via a signal or broadcast to the condition variable, the mutex is reobtained before this service returns to the caller.

This service waits until the absolute time specified by the *absolute_time* parameter.

Upon return, the calling thread will own the specified mutex – regardless of completion status. The one exception is the case where the mutex was not owned prior to calling this service.

This API is a cancellation point, meaning that if a cancellation is pending, it will be detected and executed by this API.

API Parameters:

`condition_var_handle`	Handle of the condition variable to suspend on.
`mutex_handle`	Handle of the mutex associated with this condition variable.
`absolute_time`	Absolute time to wait for. If the thread is still suspended on this condition variable when this time is reached, the service will timeout, and the thread will be resumed.

Return Codes:

PX5_SUCCESS (0)	Successful condition variable wait with timeout.
EINVAL	Invalid condition variable handle, mutex handle pointer, or absolute time. In addition, if the specified mutex is not owned or if a different mutex is associated with this condition variable, this error is returned.
ETIMEDOUT	Maximum wait time was reached, and this service timed out.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

SUSPENSION. The calling thread is suspended until another thread signals or broadcasts to the condition variable. The calling thread times out and returns with an error if absolute time is reached.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

pthread_cond_*, pthread_cond_signal, pthread_cond_wait

Small Example:

```
#include <pthread.h>

/* Condition variable and mutex handles.  */
pthread_cond_t      my_condition_var_handle;
pthread_mutex_t     my_mutex_handle;
timespec            max_time_to_wait;
int                 status;

    /* Get current time.  */
    clock_gettime(CLOCK_REALTIME, &max_time_to_wait);
```

```
/* Only wait for 2 seconds on the condition variable.  */
max_time_to_wait.tv_sec += 2;


/* Wait on this condition variable for maximum of 2 seconds. */
status =  pthread_cond_timedwait(&my_condition_var_handle,
                          &my_mutex_handle, &max_time_to_wait);

/* If status is PX5_SUCCESS, the condition variable was
   signed/broadcasted, and the mutex is again owned by the
   calling thread. */
```

pthread_cond_wait

C Prototype:

```
#include <pthread.h>

int pthread_cond_wait(pthread_cond_t *  condition_var_handle,
                      pthread_mutex_t *  mutex_handle);
```

Description:

This services suspends on the condition variable specified by the *condition_var_handle* parameter. Before this service is called, the thread must have obtained the mutex specified by *mutex_handle*. Internally, PX5 releases the mutex atomically with the thread suspension. Once the calling thread is resumed via a signal or broadcast to the condition variable, the mutex is reobtained before this service returns to the caller.

Upon return, the calling thread will again own the specified mutex – regardless of completion status. The one exception is the case where the mutex was not owned prior to calling this service.

This API is a cancellation point, meaning that if a cancellation is pending, it will be detected and executed by this API.

API Parameters:

`condition_var_handle`	Handle of the condition variable to suspend on.
`mutex_handle`	Handle of the mutex associated with this condition variable.

Return Codes:

PX5_SUCCESS (0)	Successful condition variable wait with timeout.
EINVAL	Invalid condition variable handle or mutex handle pointer. In addition, if the specified mutex is not owned or if a different mutex is

associated with this condition variable, this error is returned.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

S **SUSPENSION**. The calling thread is suspended until another thread signals or broadcasts to the condition variable.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

pthread_cond_*, pthread_cond_timedwait, pthread_cond_signal

Small Example:

```
#include <pthread.h>

/* Condition variable and mutex handles.  */
pthread_cond_t      my_condition_var_handle;
pthread_mutex_t     my_mutex_handle;
int                 status;


    /* Wait on this condition variable. */
    status =  pthread_cond_wait(&my_condition_var_handle,
                                 &my_mutex_handle);

    /* If status is PX5_SUCCESS, the condition variable was
       signed/broadcasted, and mutex is again owned by the
      calling thread. */
```

pthread_condattr_destroy

C Prototype:

```
#include <pthread.h>

int pthread_condattr_destroy(pthread_condattr_t *
                                         condition_var_attributes);
```

Description:

This service destroys the previously created condition variable attributes structure pointed to by *condition_var_attributes*. Once destroyed, the condition variable attributes structure cannot be used again unless it is recreated.

API Parameters:

`condition_var_attributes`	Pointer to the condition variable attributes to destroy.

Return Codes:

PX5_SUCCESS (0)	Successful condition variable attributes destroy.
EINVAL	Invalid condition variable attributes pointer.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

pthread_cond_*, pthread_cond_init

Small Example:

```
#include <pthread.h>

/* Condition variable attribute structure.  */
pthread_condattr_t     my_condition_var_attributes;
int                    status;




    /* Destroy the condition variable attributes referenced by
       "my_condition_var_attributes". */
    status =  pthread_condattr_destroy(&my_condition_var_attributes);

    /* If status is PX5_SUCCESS, the condition variable attributes
       structure was destroyed.  */
```

pthread_condattr_getpshared

C Prototype:

```
#include <pthread.h>

int pthread_condattr_getpshared(pthread_condattr_t *
         condition_var_attributes, int *  process_sharing_designation);
```

Description:

This service returns the current process sharing designation contained in the condition variable attribute structure. By default, the process sharing designation is *PTHREAD_PROCESS_PRIVATE*.

API Parameters:

`condition_var_attributes`	Pointer to the condition variable attributes.
`process_sharing_designation`	Pointer to the destination for the condition variable process sharing designation.

Return Codes:

PX5_SUCCESS (0)	Successful condition variable attributes process sharing designation retrieval.
EINVAL	Invalid condition variable attributes or process sharing designation pointer.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NO PREEMPTION. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

pthread_condattr_*, pthread_cond_init

Small Example:

```
#include <pthread.h>

/* Condition variable attribute structure.  */
pthread_condattr_t      my_condition_var_attributes;
int                     process_sharing_designation;
int                     status;



    /* Get the process sharing designation found in the condition
       variable attributes structure "my_condition_var_attributes". */
    status =  pthread_condattr_getpshared(&my_condition_var_attributes,
                                         &process_sharing_designation);

    /* If status is PX5_SUCCESS, the "process_sharing_designation"
        contains the current process sharing designation.  */
```

pthread_condattr_init

C Prototype:

```
#include <pthread.h>

int pthread_condattr_init(pthread_condattr_t *
                                        condition_var_attributes);
```

Description:

This service initializes the condition variable attributes structure with default condition variable creation values. These defaults are as follows:

Attribute	Default Setting
Process Sharing	Private – *PTHREAD_PROCESS_PRIVATE*

API Parameters:

`condition_var_attributes`	Pointer to the condition variable attributes structure to create.

Return Codes:

PX5_SUCCESS (0)	Successful condition variable attributes structure creation.
EINVAL	Invalid condition variable attributes pointer.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

pthread_cond_*, pthread_cond_init

Small Example:

```
#include <pthread.h>

/* Condition variable attribute structure.  */
pthread_condattr_t     my_condition_var_attributes;
int                    status;




    /* Create the condition variable attributes structure
      "my_condition_var_attributes". */
    status =  pthread_condattr_init(&my_condition_var_attributes);

    /* If status is PX5_SUCCESS, the "my_condition_var_attributes"
       structure is ready for use.  */
```

pthread_condattr_setpshared

C Prototype:

```
#include <pthread.h>

int pthread_condattr_setpshared(pthread_condattr_t *
         condition_var_attributes, int   process_sharing_designation);
```

Description:

This service sets the process sharing designation in the condition variable attribute structure to either *PTHREAD_PROCESS_PRIVATE* or *PTHREAD_PROCESS_SHARED.* By default, the process sharing designation is *PTHREAD_PROCESS_PRIVATE*.

API Parameters:

`condition_var_attributes`	Pointer to the condition variable attributes.
`process_sharing_designation`	Process sharing designation, either *PTHREAD_PROCESS_PRIVATE* or *PTHREAD_PROCESS_SHARED.*

Return Codes:

PX5_SUCCESS (0)	Successful condition variable attributes process sharing designation selection.
EINVAL	Invalid condition variable attributes structure or invalid process sharing designation.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NO PREEMPTION. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

pthread_cond_*, pthread_cond_init

Small Example:

```
#include <pthread.h>

/* Condition variable attribute structure.  */
pthread_condattr_t      my_condition_var_attributes;
int                     status;



    /* Set the process sharing designation in the condition variable
       attributes structure "my_condition_var_attributes". */
    status =  pthread_condattr_setpshared(&my_condition_var_attributes,
                                          PTHREAD_PROCESS_PRIVATE);

    /* If status is PX5_SUCCESS, the "my_condition_var_attributes"
        structure contains the PTHREAD_PROCESS_PRIVATE designation. */
```

pthread_create

C Prototype:

```
#include <pthread.h>

int pthread_create(pthread_t *  thread_handle,
                    const pthread_attr * attributes,
                    void *  (*start_routine)(void *),
                    void *arguments);
```

Description:

This service creates a new thread that starts execution at the caller’s *start_routine*, passing verbatim the supplied *arguments* parameter. Note that the optional *attributes* parameter can specify additional attributes for the thread, including explicitly setting its priority, stack location, stack size, etc. Please review the *pthread_attr_** APIs for more information on available thread creation attributes. The signal mask for the newly created thread is inherited from the calling thread.

If the thread creation is successful, the newly created thread's handle is returned in *thread_handle*. Otherwise, if the thread creation is unsuccessful, an error code is returned (as defined below).

By default, threads are created as joinable, meaning that another thread can wait for the thread to complete. All joinable threads should use pthread_exit when their processing is complete and either be joined or detached via pthread_join or pthread_detach. Failure to do so can leave the thread in a perpetual terminated state with all its resources allocated (internal thread control structure and stack).

By default, threads are created with deferred cancellation enabled. This can be changed dynamically via the pthread_setcancelstate and pthread_setcanceltype APIs.

API Parameters:

`thread_handle`	Pointer supplied by the caller to place the new thread handle upon successful creation.

`attributes`	Optional pointer to the new thread attributes structure, pthread_*attr*. Please see the *pthread_attr_** APIs for information on how to specify thread creation attributes.
`start_routine`	Specifies the new thread entry function. When the new thread is executed, this function will be called.
`Arguments`	Specifies the value supplied to the *start_routine* when called. Note this is only for use by the application.

Return Codes:

PX5_SUCCESS (0)	Successful new thread creation.
EINVAL	Invalid argument(s).
EAGAIN	Not enough resources to create the thread.

Real-time Scenarios:

Upon the successful execution of this service, the following real-time scenarios are possible:

NO PREEMPTION. If the newly created thread is the same or less priority than the calling thread, there is no preemption–this service simply returns immediately after creation of the new thread.

PREEMPTION. If the newly created thread is higher priority than the calling thread, the calling thread is *preempted,* and execution takes place in the newly created thread prior to returning from this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

pthread_attr_*, pthread_exit, pthread_detach, pthread_join, pthread_cancel

Small Example:

```
#include <pthread.h>

/* Define some memory for PX5.  */
unsigned long   memory_area[1024];

pthread_t       new_thread_handle;

unsigned long   new_thread_counter;
unsigned long   main_thread_counter;


/* Define new thread's staring function.  */
void *  new_thread_start(void *arguments)
{

    /* Loop forever incrementing a counter.  */
    while (1)
    {
        new_thread_counter++;
    }
}

int main(void)
{

int   status;


    /* Call the PX5 start function.  */
    status =  pthread_start(memory_area, sizeof(memory_area));

    /* Check completion status.  */
    if (status != PX5_SUCCESS)
    {
        printf(“Error starting PX5!\n”);
        exit(1);
    }

    /* Create the new thread.  */
    status =  pthread_create(&new_thread_handle, NULL,
                             new_thread_start, NULL);

    /* Check completion status.  */
    if (status != PX5_SUCCESS)
        printf("Error creating new thread!\n");

    /* In any case, simply loop in the main program, which is the
       initial thread.  */
    while(1)
    {

        /* Just increment the main thread's counter.  */
        main_thread_counter++;
    }
}
```

pthread_detach

C Prototype:

```
#include <pthread.h>

int  pthread_detach(pthread_t thread_handle);
```

Description:

This service places the specified thread in a detached state, such that it can't be joined or canceled in the future. It is equivalent to creating the thread with the *PTHREAD_CREATE_DETACHED* attribute. When the specified, detached thread exits or returns, all of its resources are immediately released.

A thread may call pthread_detach on itself.

API Parameters:

`thread_handle`	Handle of previously created thread.

Return Codes:

PX5_SUCCESS (0)	Successful detach of thread.
EINVAL	Specified thread was already detached or not in a joinable state.
ESRCH	Specified thread could not be found.

Real-time Scenarios:

Upon the successful execution of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

pthread_create, pthread_exit, pthread_join, pthread_cancel

Small Example:

```
#include <pthread.h>

pthread_t         my_thread_handle;
int               status;



    /* Set the detach state in the previously created thread.  */
    status =  pthread_detach(my_thread_handle);

    /* If status contains PX5_SUCCESS, the thread is now detached. */
```

pthread_equal

C Prototype:

```
#include <pthread.h>

int  pthread_equal(pthread_t first_thread, pthread_t second_thread);
```

Description:

This service compares two thread handles. If the handles are the same, a non-zero value is returned. If they are not the same, a value of zero is returned.

API Parameters:

`first_thread`	Handle of the first thread.
`second_thread`	Handle of the second thread.

Return Codes:

0	A value of zero is returned if the handles are not the same.
1	A value of 1 is returned if the handles are the same.

Real-time Scenarios:

Upon the successful execution of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

pthread_self

Small Example:

```
#include <pthread.h>

pthread_t    thread_to_exit;



void *  my_thread(void *  argument)
{

    /* Perform some processing…   */

    /* Determine if this is the thread designated to exit by
       the global thread handle "thread_to_exit". */
    if (pthread_equal(pthread_self(), thread_to_exit)
    {

        /* Yes, this is the thread that needs to exit.  */
        pthread_exit((void *) 7);
    }

}
```

pthread_exit

C Prototype:

```
#include <pthread.h>

void  pthread_exit(void *exit_value);
```

Description:

This service terminates the currenting executing thread. If this thread is detached, all of the thread's resources are released. Any cancellation cleanup handlers are popped and executed. The *exit_value* passed to this API will be sent to a thread that has joined this thread via the *pthread_join* API, assuming the value was requested in the join request (non-NULL destination value).

An implicit call to pthread_exit is made if the application returns from the thread's start routine.

If this function is called from the main thread (or the C main function returns), all multithreading stops, and execution is transferred to the C compiler's exit function.

API Parameters:

`exit_value`	Even though the type is "void *" this parameter is treated as a value and passed verbatim to a thread that has joined this thread. If no thread has joined, this parameter has no effect.

Return Codes:

None

Real-time Scenarios:

Upon the successful execution of this service, the following real-time scenarios are possible:

PREEMPTION. If a higher-priority thread has joined this thread, the exit processing will resume its execution, and preemption will occur.

SUSPENSION. Once the currently executing thread exit processing is complete, the thread enters a permanent terminated state of suspension.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

pthread_detach, pthread_join, pthread_cancel

Small Example:

```
#include <pthread.h>


void *  my_thread(void *  argument)
{

    /* Perform some processing…   */

    /* Processing is done, exit with a value of 7.  */
    pthread_exit((void *) 7);

    /* Note: processing will never get here since exit does not
       return.  */
}
```

pthread_join

C Prototype:

```
#include <pthread.h>

int  pthread_join(pthread_t thread_to_join, void ** value_destination);
```

Description:

This service suspends the currenting running thread until the specified thread completes its processing. When the specified thread exits, the value supplied in its exit call is returned in the destination specified by the caller–if a non-NULL exit value destination pointer is supplied.

All joinable threads that complete, should be joined or detached via pthread_join and pthread_detach, respectively.

Only one thread is allowed to join another thread. Any additional join requests will return an error.

This API is a cancellation point, meaning that if a cancellation is pending, it will be detected and executed by this API.

API Parameters:

`thread_to_join`	Handle of thread to join, i.e., thread completion to wait for.
`value_desitination`	Destination for the value from the specified thread's exit call.

Return Codes:

PX5_SUCCESS (0)	Successful thread join.
EINVAL	Thread specified is not joinable.
ESRCH	Thread specified is not a valid thread.

EDEADLK	Thread is attempting to join with itself, which would create a deadlock.

Real-time Scenarios:

Upon the successful execution of this service, the following real-time scenarios are possible:

S **SUSPENSION**. Once the currently executing thread is joined with the thread specified, it enters a suspended state. The highest priority ready thread will then start executing.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

pthread_exit, pthread_detach

Small Example:

```
#include <pthread.h>

pthread_t    my_thread2_handle;


void *  my_thread1(void *  argument)
{

void *  exit_value;


    /* Join the my_thread2. The API will suspend until my_thread_2
        exits. */
    pthread_join(my_thread2_handle, &my_thread_2_exit_value);

    /* When processing returns here, my_thread_2 has exited with a
       value of "7", which has been stored in "exit_value". */
}

void *  my_thread1(void *  argument)
{

    pthread_exit((void *) 7);
}
```

pthread_kill

C Prototype:

```
#include <signal.h>

int  pthread_kill(pthread_t thread, int signal);
```

Description:

This service raises a signal for the specified thread. If the signal is masked, it is added to the pending signals set. In addition, if the specified thread is suspended waiting for this signal, it is resumed. Otherwise, if the signal is not masked by the specified thread, the registered signal handler is invoked. If no signal handler has been registered, an error is returned.

API Parameters:

`thread`	Handle of thread to raise signal for.
`signal`	Signal number to raise, valid values range from 0 through 31, where signal 0 does not cause any actual signal processing – only error checking is performed.

Return Codes:

PX5_SUCCESS (0)	Successful signal raise.
EINVAL	Invalid signal number or signal was unmasked, but no signal handler was registered.
ESRCH	Thread specified is not a valid thread.

Real-time Scenarios:

Upon the successful execution of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. If the thread is not waiting for this signal, or there is no signal handler, or the signal handler does not unblock any higher-priority suspended thread, no preemption takes place.

P **PREEMPTION**. If a higher-priority thread is waiting for the signal, when it is delivered, the waiting thread is resumed, and preemption will occur.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

pthread_sigmask, sigaction, sigwait

Small Example:

```
#include <pthread.h>

pthread_t    my_thread2_handle;


void *  my_thread1(void *  argument)
{


    /* Raise SIGUSR1 signal for "my_thread2_hanlde". */
    pthread_kill(my_thread2_handle, SIGUSR1);

    /* If successful, the signal SIGUSR1 has been raised. */
}
```

pthread_mutex_destroy

C Prototype:

```
#include <pthread.h>

int pthread_mutex_destroy(pthread_mutex_t *  mutex_handle);
```

Description:

This service destroys the previously created mutex specified by *mutex_handle*. If the mutex is still owned by another thread, an error is returned.

API Parameters:

`mutex_handle`	Handle of the mutex to destroy.

Return Codes:

PX5_SUCCESS (0)	Successful mutex destroy.
EINVAL	Invalid mutex handle.
EBUSY	A thread currently owns the mutex.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NO PREEMPTION. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

pthread_mutex_*, pthread_mutex_create

Small Example:

```
#include <pthread.h>

/* Mutex handle.  */
pthread_mutex_t    my_mutex_handle;
int                status;



    /* Destroy the mutex referenced by “my_mutex_handle”. */
    status =  pthread_mutex_destroy(&my_mutex_handle);

    /* If status is PX5_SUCCESS, the mutex was destroyed.  */
```

pthread_mutex_init

C Prototype:

```
#include <pthread.h>

int pthread_mutex_init(pthread_mutex_t *  mutex_handle,
                       pthread_mutexattr_t *  mutex_attributes);
```

Description:

This service initializes (creates) a mutex with the optional mutex attributes. If successful, the mutex handle is setup for further use by the application.

API Parameters:

`mutex_handle`	Handle of the mutex to initialize.
`mutex_attributes`	Optional mutex attributes. This value is NULL if no mutex attributes are specified.

Return Codes:

PX5_SUCCESS (0)	Successful mutex initialization.
EINVAL	Invalid mutex handle pointer or mutex attributes.
EBUSY	Mutex is already created.
ENOMEM	Insufficient memory to create mutex.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

pthread_mutexattr_*, pthread_mutex_destroy

Small Example:

```
#include <pthread.h>

/* Mutex handle.  */
pthread_mutex_t    my_mutex_handle;
int                status;



    /* Create the mutex and setup “my_mutex_handle”. */
    status =  pthread_mutex_init(&my_mutex_handle, NULL);

    /* If status is PX5_SUCCESS, the mutex was created, and the mutex
       handle is ready to use.  */
```

pthread_mutex_lock

C Prototype:

```
#include <pthread.h>

int pthread_mutex_lock(pthread_mutex_t *  mutex_handle);
```

Description:

If the mutex is available, this service assigns ownership of the mutex to the calling thread. Otherwise, if the mutex is already owned by another thread, the calling thread suspends until the mutex is released by the other thread.

If a thread terminates while owning a mutex, all other threads waiting for the mutex will be indefinitely suspended.

API Parameters:

`mutex_handle`	Handle of the mutex to lock.

Return Codes:

PX5_SUCCESS (0)	Successful mutex lock.
EDEADLK	Thread already owns mutex (non-recursive mutex).
EINVAL	Invalid mutex handle pointer.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. If the mutex was available and ownership assigned to the calling thread, no preemption takes place.

S **SUSPENSION**. If the mutex is already owned by another thread, the calling thread is suspended until the other thread releases ownership via *pthread_mutex_unlock*.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

pthread_mutex_*, pthread_mutex_unlock

Small Example:

```
#include <pthread.h>

/* Mutex handle.  */
pthread_mutex_t    my_mutex_handle;
int                status;



    /* Lock the mutex "my_mutex_handle". */
    status =  pthread_mutex_lock(&my_mutex_handle);

    /* If status is PX5_SUCCESS, the mutex in now owned by the
       calling thread. Any other thread attempting to lock
       the mutex will suspend.  */
```

pthread_mutex_trylock

C Prototype:

```
#include <pthread.h>

int pthread_mutex_trylock(pthread_mutex_t *  mutex_handle);
```

Description:

If the mutex is available, this service assigns ownership of the mutex to the calling thread. Otherwise, if the mutex is already owned by another thread, this service immediately returns an error, i.e., there is no thread suspension like the *pthread_mutex_lock* service.

If a thread terminates while owning a mutex, all other threads waiting for the mutex will be indefinitely suspended.

API Parameters:

`mutex_handle`	Handle of the mutex to try to lock.

Return Codes:

PX5_SUCCESS (0)	Successful mutex lock.
EDEADLK	Thread already owns mutex (non-recursive mutex).
EBUSY	Mutex already owned by another thread.
EINVAL	Invalid mutex handle pointer.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NO PREEMPTION. If the mutex was available and ownership assigned to the calling thread, no preemption takes place.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

pthread_mutex_*, pthread_mutex_lock, pthread_mutex_unlock

Small Example:

```
#include <pthread.h>

/* Mutex handle.  */
pthread_mutex_t    my_mutex_handle;
int                status;



    /* Try to lock the mutex “my_mutex_handle”. */
    status =  pthread_mutex_trylock(&my_mutex_handle);

    /* If status is PX5_SUCCESS, the mutex is now owned by the
       calling thread. Any other thread attempting to lock
       the mutex will suspend.  */
```

pthread_mutex_unlock

C Prototype:

```
#include <pthread.h>

int pthread_mutex_unlock(pthread_mutex_t *  mutex_handle);
```

Description:

This service releases a previously owned mutex. If there are other threads suspended waiting for the mutex, the oldest suspended thread is given ownership and resumed.

API Parameters:

`mutex_handle`	Handle of the mutex to unlock.

Return Codes:

PX5_SUCCESS (0)	Successful mutex unlock.
EINVAL	Invalid mutex handle pointer.
EPERM	Calling thread does not own the mutex.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. If there are no other threads waiting for the mutex, no preemption takes place.

P **PREEMPTION**. If a higher-priority thread is waiting for the mutex, when it is given the mutex, the waiting thread is resumed, and preemption will occur.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

pthread_mutex_*, pthread_mutex_lock, pthread_mutex_trylock

Small Example:

```
#include <pthread.h>

/* Mutex handle.  */
pthread_mutex_t    my_mutex_handle;
int                status;



    /* Unlock the mutex "my_mutex_handle". */
    status =  pthread_mutex_unlock(&my_mutex_handle);

    /* If status is PX5_SUCCESS, the mutex is now unlocked, and no
       longer owned by the calling thread. */
```

pthread_mutexattr_destroy

C Prototype:

```
#include <pthread.h>

int pthread_mutexattr_destroy(pthread_mutexattr_t * mutex_attributes);
```

Description:

This service destroys the previously created mutex attributes structure pointed to by *mutex_attributes*. Once destroyed, the mutex attributes structure cannot be used again unless it is recreated.

API Parameters:

`mutex_attributes`	Pointer to the mutex attributes to destroy.

Return Codes:

PX5_SUCCESS (0)	Successful mutex attributes destroy.
EINVAL	Invalid mutex attributes pointer.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

pthread_mutex_*, pthread_mutexattr_*, pthread_mutexattr_init

Small Example:

```
#include <pthread.h>

/* Mutex attribute structure.  */
pthread_mutexattr_t    my_mutex_attributes;
int                    status;



    /* Destroy the mutex attributes referenced by
       "my_mutex_attributes". */
    status =  pthread_mutexattr_destroy(&my_mutex_attributes);

    /* If status is PX5_SUCCESS, the mutex attributes structure
       was destroyed.  */
```

pthread_mutexattr_getprotocol

C Prototype:

```
#include <pthread.h>

int pthread_mutexattr_getprotocol(pthread_mutexattr_t
                              *mutex_attributes, int *  protocol);
```

Description:

This service returns the previously supplied mutex protocol. By default, the process sharing designation is *PTHREAD_PRIO_NONE*.

API Parameters:

`mutex_attributes`	Pointer to the mutex attributes.
`protocol`	Pointer to the destination for the previously supplied mutex protocol.

Return Codes:

PX5_SUCCESS (0)	Successful mutex attributes mutex protocol retrieval.
EINVAL	Invalid mutex attributes or mutex protocol pointer.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

pthread_mutex_*, pthread_mutexattr_*, pthread_mutexattr_setprotocol

Small Example:

```
#include <pthread.h>

/* Mutex attribute structure.  */
pthread_mutexattr_t    my_mutex_attributes;
char *                 my_mutex_protocol;
int                    status;



    /* Get the mutex protocol in the mutex attributes structure
       "my_mutex_attributes". */
    status =  pthread_mutexattr_getprotocol(&my_mutex_attributes,
                                              &my_mutex_protocol);

    /* If status is PX5_SUCCESS, "my_mutex_protocol" contains the
        previously supplied protocol. */
```

pthread_mutexattr_getpshared

C Prototype:

```
#include <pthread.h>

int pthread_mutexattr_getpshared(pthread_mutexattr_t *mutex_attributes,
                                    int *  process_sharing_designation);
```

Description:

This service returns the current process sharing designation contained in the mutex attribute structure. By default, the process sharing designation is *PTHREAD_PROCESS_PRIVATE*.

API Parameters:

`mutex_attributes`	Pointer to the mutex attributes.
`process_sharing_designation`	Pointer to the destination for the mutex process sharing designation.

Return Codes:

PX5_SUCCESS (0)	Successful mutex attributes process sharing designation retrieval.
EINVAL	Invalid mutex attributes or process sharing designation pointer.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NO PREEMPTION. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

pthread_mutex_*, pthread_mutexattr_init

Small Example:

```
#include <pthread.h>

/* Mutex attribute structure.  */
pthread_mutexattr_t    my_mutex_attributes;
int                    process_sharing_designation;
int                    status;



    /* Get the process sharing designation found in the mutex
       attributes structure "my_mutex_attributes". */
    status =  pthread_mutexattr_getpshared(&my_mutex_attributes,
                                    &process_sharing_designation);

    /* If status is PX5_SUCCESS, "process_sharing_designation"
        contains the current process sharing designation.  */
```

pthread_mutexattr_gettype

C Prototype:

```
#include <pthread.h>

int pthread_mutexattr_gettype(pthread_mutexattr_t
                                        *mutex_attributes, int *  type);
```

Description:

This service returns the previously supplied mutex type. By default, the process sharing designation is *PTHREAD_MUTEX_ERRORCHECK*.

API Parameters:

`mutex_attributes`	Pointer to the mutex attributes.
`type`	Pointer to the destination for the previously supplied mutex type.

Return Codes:

PX5_SUCCESS (0)	Successful mutex attributes mutex type retrieval.
EINVAL	Invalid mutex attributes or mutex type pointer.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

pthread_mutex_*, pthread_mutexattr_*, pthread_mutexattr_settype

Small Example:

```
#include <pthread.h>

/* Mutex attribute structure.  */
pthread_mutexattr_t    my_mutex_attributes;
char *                 my_mutex_type;
int                    status;



    /* Get the mutex type in the mutex attributes structure
       “my_mutex_attributes”. */
    status =  pthread_mutexattr_gettype(&my_mutex_attributes,
                                                &my_mutex_type);

    /* If status is PX5_SUCCESS, “my_mutex_type” contains the
        previously supplied mutex type. */
```

pthread_mutexattr_init

C Prototype:

```
#include <pthread.h>

int pthread_mutexattr_init(pthread_mutexattr_t *mutex_attributes);
```

Description:

This service initializes the mutex attributes structure with default mutex creation values. These defaults are as follows:

Attribute	**Default Setting**
Process Sharing	Private – *PTHREAD_PROCESS_PRIVATE*

API Parameters:

`mutex_attributes`	Pointer to the mutex attributes structure to create.

Return Codes:

PX5_SUCCESS (0)	Successful mutex attributes structure creation.
EINVAL	Invalid mutex attributes pointer.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

pthread_mutex_*, pthread_mutexattr_destroy

Small Example:

```
#include <pthread.h>

/* Mutex attribute structure.  */
pthread_mutexattr_t    my_mutex_attributes;
int                    status;



    /* Create the mutex attributes structure “my_mutex_attributes”. */
    status =  pthread_mutexattr_init(&my_mutex_attributes);

    /* If status is PX5_SUCCESS, the “my_mutex_attributes” structure is
       ready for use.  */
```

pthread_mutexattr_setprotocol

C Prototype:

```
#include <pthread.h>

int pthread_mutexattr_setprotocol(pthread_mutexattr_t
                              *mutex_attributes, int protocol);
```

Description:

This service sets the specified protocol in the previously created mutex attributes structure. By default, the protocol is *PTHREAD_PRIO_NONE*.

API Parameters:

`mutex_attributes`	Pointer to the mutex attributes.
`protocol`	Specified mutex protocol, as follows: *PTHREAD_PRIO_NONE* *PTHREAD_PRIO_INHERIT*

Return Codes:

PX5_SUCCESS (0)	Successful mutex attributes mutex protocol set.
EINVAL	Invalid mutex attributes or mutex protocol. Note that *PTHREAD_PRIO_PROTECT* is not currently supported.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

pthread_mutex_*, pthread_mutexattr_*, pthread_mutexattr_getprotocol

Small Example:

```
#include <pthread.h>

/* Mutex attribute structure.  */
pthread_mutexattr_t    my_mutex_attributes;
char *                 my_mutex_protocol;
int                    status;



    /* Set the mutex protocol in the mutex attributes structure
       "my_mutex_attributes". */
    status =  pthread_mutexattr_setprotocol(&my_mutex_attributes,
                                          PTHREAD_PRIO_INHERIT);

    /* If status is PX5_SUCCESS, "my_mutex_protocol" specifies that
       priority inheritance is enabled when the mutex using these
       attributes is created.  */
```

pthread_mutexattr_setpshared

C Prototype:

```
#include <pthread.h>

int pthread_mutexattr_setpshared(pthread_mutexattr_t *mutex_attributes,
                                   int   process_sharing_designation);
```

Description:

This service sets the process sharing designation in the mutex attribute structure to either *PTHREAD_PROCESS_PRIVATE* or *PTHREAD_PROCESS_SHARED.* By default, the process sharing designation is *PTHREAD_PROCESS_PRIVATE*.

API Parameters:

`mutex_attributes`	Pointer to the mutex attributes.
`process_sharing_designation`	Process sharing designation, either *PTHREAD_PROCESS_PRIVATE* or *PTHREAD_PROCESS_SHARED.*

Return Codes:

PX5_SUCCESS (0)	Successful mutex attributes process sharing designation selection.
EINVAL	Invalid mutex attributes or invalid process sharing designation.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NO PREEMPTION. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

pthread_mutex_*, pthread_mutexattr_init

Small Example:

```
#include <pthread.h>

/* Mutex attribute structure.  */
pthread_mutexattr_t    my_mutex_attributes;
int                    status;



    /* Set the process sharing designation in the mutex
       attributes structure "my_mutex_attributes". */
    status =  pthread_mutexattr_setpshared(&my_mutex_attributes,
                                     PTHREAD_PROCESS_PRIVATE);

    /* If status is PX5_SUCCESS, the "my_mutex_attributes" structure
       contains the PTHREAD_PROCESS_PRIVATE designation. */
```

pthread_mutexattr_settype

C Prototype:

```
#include <pthread.h>

int pthread_mutexattr_settype(pthread_mutexattr_t
                     *  mutex_attributes, int type);
```

Description:

This service sets the specified mutex type in the previously created mutex attributes structure. By default, the type is *PTHREAD_MUTEX_ERRORCHECK*.

API Parameters:

`mutex_attributes`	Pointer to the mutex attributes.
`type`	Specified mutex type, as follows: *PTHREAD_MUTEX_NORMAL* *PTHREAD_MUTEX_ERRORCHECK* *PTHREAD_MUTEX_RECURSIVE*

Return Codes:

PX5_SUCCESS (0)	Successful mutex attributes mutex type set.
EINVAL	Invalid mutex attributes or mutex type.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

pthread_mutex_*, pthread_mutexattr_*, pthread_mutexattr_gettype

Small Example:

```
#include <pthread.h>

/* Mutex attribute structure.  */
pthread_mutexattr_t    my_mutex_attributes;
int                    status;




    /* Set the mutex type in the mutex attributes structure
       "my_mutex_attributes". */
    status =  pthread_mutexattr_settype(&my_mutex_attributes,
                                         PTHREAD_MUTEX_RECURSIVE);

    /* If status is PX5_SUCCESS, "my_mutex_protocol" specifies that
       nested (recursive) mutex locking is enabled when the mutex using
       these attributes is created.  */
```

pthread_self

C Prototype:

```
#include <pthread.h>

pthread_t pthread_self(void);
```

Description:

This service returns the thread handle of the currently running thread.

API Parameters:

`none`

Return Codes:

pthread_t	Handle of the currently running thread.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

pthread_equal

Small Example:

```
#include <pthread.h>

/* Thread handle.  */
pthread_t    my_thread_handle;



void *  my_thread(void *)
{

    /* Pickup the thread handle for this thread. */
    my_thread_handle =  pthread_self();

    /* my_thread_handle can now be used in other
       pthread API calls.  */
}
```

pthread_setcancelstate

C Prototype:

```
#include <pthread.h>

int  pthread_setcancelstate(int new_state, int *  old_state);
```

Description:

This service sets the calling thread's cancel state as specified by *new_state*. If *old_state* is non-NULL, the previous cancel state is stored in the specified destination.

By default, threads are created with the cancel state of PTHREAD_CANCEL_ENABLE.

API Parameters:

`new_state`	New cancelation state, which is one of the following: *PTHREAD_CANCEL_ENABLE* *PTHREAD_CANCEL_DISABLE*
`old_state`	If non-NULL, destination to store the previous cancel state of the calling thread.

Return Codes:

PX5_SUCCESS (0)	Successful change of calling thread's cancel state.
EINVAL	Invalid cancel state.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NO PREEMPTION. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

pthread_cancel, pthread_cleanup_pop, pthread_cleanup_push, pthread_create, pthread_setcanceltype, pthread_testcancel

Small Example:

```
#include <pthread.h>

int             status;


    /* Enable cancellation for the calling thread. */
    status =  pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);

    /* Cancellation is now available for the calling thread.  */
```

pthread_setcanceltype

C Prototype:

```
#include <pthread.h>

int  pthread_setcanceltype(int new_type, int old_type);
```

Description:

This service sets the calling thread's cancel type as specified by *new_type*. If *old_type* is non-NULL, the previous cancel type is stored in the specified destination.

Threads making PX5 RTOS API calls should only use deferred cancellation (PTHREAD_CANCEL_DEFERRED), since asynchronous cancellation could terminate thread processing inside of an API and leaving the system in an unknown state.

By default, threads are created with the cancel type of PTHREAD_CANCEL_DEFERRED.

API Parameters:

`new_type`	New cancelation type, which is one of the following: *PTHREAD_CANCEL_DEFERRED* *PTHREAD_CANCEL_ASYNCHRONOUS*
`old_type`	If non-NULL, destination to store the previous cancel type of the calling thread.

Return Codes:

PX5_SUCCESS (0)	Successful change of calling thread's cancel type.
EINVAL	Invalid cancel type.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NO PREEMPTION. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

pthread_cancel, pthread_cleanup_pop, pthread_cleanup_push, pthread_create, pthread_setcancelstate, pthread_testcancel

Small Example:

```
#include <pthread.h>


int             status;



    /* Enable asynchronous cancellation for the calling thread. */
    status =  pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS, NULL);

    /* Asynchronous cancellation is now enabled for calling thread. */
```

pthread_sigmask

C Prototype:

```
#include <signal.h>

int pthread_sigmask(int operation, sigset_t * signal_set,
                                   sigset_t *  previous_mask);
```

Description:

This service sets or clears signals in the currently executing thread's signal mask. The exact determination of how the signal mask is modified depends on the *operation* parameter as described in the parameter section below. Note that each thread inherits its initial signal mask from the thread that created it.

If a pending signal is unmasked and there is a corresponding signal handler, the signal handler is executed prior to returning from this call.

Signals must be masked in order to synchronously wait for them via one of the signal wait APIs.

API Parameters:

operation	Specifies the operation to perform on the calling thread's signal mask. Valid operations are:
	SIG_BLOCK The supplied signal set is used to block (set) signals in the thread's signal mask.
	SIG_UNBLOCK The supplied signal set is used to unblock (clear) signals in the thread's signal mask.
	SIG_SETMASK The supplied signal set is used to update the thread's signal mask verbatim.

`signal_set`	The signal set bitmap used to modify the thread's signal mask based on the specified operation.
`previous_mask`	The thread's previous signal mask.

Return Codes:

PX5_SUCCESS (0)	Successful signal mask update.
EINVAL	Invalid operation or signal set pointer.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. There is no preemption possible with this service.

P **PREEMPTION**. If a pending signal is unmasked and the corresponding signal handler resumes a higher priority thread, preemption will occur.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

pthread_kill, sigaction, sigwait

Small Example:

```
#include <pthread.h>

sigset_t   new_signal_mask;
```

```
void *  my_thread(void *)
{

    /* Set the signal mask to all ones – masking everything. */
    sigfillset(&new_signal_mask);

    /* Now mask all signals for this thread.  */
    pthread_sigmask(SIG_SETMASK, &new_signal_mask, NULL);


    /* All signals are now masked for this thread.  */
}
```

pthread_testcancel

C Prototype:

```
#include <pthread.h>

void  pthread_testcancel(void);
```

Description:

This service checks for a pending cancel request for the calling thread. If detected, the calling thread cancels itself via *pthread_exit*.

API Parameters:

None

Return Codes:

None

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NO PREEMPTION. If a cancel request is not pending for the calling thread, there is no preemption possible with this service.

SUSPENSION. If cancelation is pending, the calling thread is suspended due to termination.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

pthread_cancel, pthread_cleanup_pop, pthread_cleanup_push, pthread_create, pthread_setcancelstate, pthread_setcanceltype

Small Example:

```
#include <pthread.h>


    /* Test for pending cancelation for the calling thread. */
    pthread_testcancel();

    /* If we return, there was no pending cancelation request. */
```

px5_errno_get

C Prototype:

```
#include <px5_errno.h>

int  px5_errno_get(void);
```

Description:

This pthreads+ service retrieves the errno value of the currently executing thread.

API Parameters:

None

Return Codes:

errno

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. No preemption is possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

px5_errno_set

Small Example:

```
#include <px5_errno.h>


int   my_errno;


    /* Pickup the errno of the calling thread. */
    my_errno =  px5_errno_get();

    /* Upon return, the thread’s errno is stored in “my_errno”. */
```

px5_errno_set

C Prototype:

```
#include <px5_errno.h>

void  px5_errno_set(int new_error);
```

Description:

This pthreads+ service sets the errno value of the currently executing thread.

API Parameters:

`new_error`	New error value for the currently executing thread.

Return Codes:

`none`

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. No preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

px5_errno_get

Small Example:

```
#include <px5_errno.h>



    /* Set the errno to EINVAL of the calling thread. */
    px5_errno_set(EINVAL);

    /* Upon return, the thread's errno is EINVAL. */
```

px5_mq_extend_open

C Prototype:

```
#include <fcntl.h>
#include <sys/stat.h>
#include <mqueue.h>

mqd_t  px5_mq_extend_open(const char * queue_name, int operation,
                           mode_t mode, mq_attr * queue_attributes,
                                 mq_extendattr * extend_attributes);
```

Description:

This pthreads+ service opens (creates) the message queue specified (with optional extended attributes) and returns the message queue handle if successful.

API Parameters:

`queue_name`	Name of the queue to open/create.
`operation`	Specifies how the queue will operate. The supported options are: O_CREAT O_RDWR O_NONBLOCK Both O_CREAT and O_RDWR should be specified. The O_NONBLOCK option that disables threads from suspending on the queue is optional.
`mode`	Not currently used.
`queue_attributes`	Attributes that specify the dimensions of the message queue, as defined by these structure members: mq_maxmsg mq_msgsize Where *mq_maxmsg* defines the total number of messages the queue can hold. The maximum size of each message (in bytes) is

	defined by *mq_msgsize*. Note the size and priority of each message must be stored along with the message content. In addition, there is one pointer type required for each message. One most architecture, this amounts to 12 bytes of additional-per message overhead.
`extend_attributes`	Optional extended attributes that can specify memory for the queue control structure as well as the queue memory area.

Return Codes:

queue handle	Positive value represents the successfully opened (created) queue handle.
PX5_ERROR (-1)	Error attempting to open/create the queue. Please use *errno* to retrieve the exact error:

ENOSPC	Not enough memory to create the specified queue.
EINVAL	Invalid operation or invalid attributes pointer.

Real-time Scenarios:

Upon the successful execution of this service, the following real-time scenarios are possible:

NO PREEMPTION. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

mq_open, mq_close, mq_getattr, mq_setattr

Small Example:

```
#include <fcntl.h>
#include <sys/stat.h>
#include <mqueue.h>


int                 status;
struct mq_attr      my_queue_attributes;
mq_extendattr_t     my_extended_queue_attributes;
mqd_t               my_queue_handle;


    /* Setup the attributes for 100 total messages, where each
       message is a maximum of 16 bytes.  */
    my_queue_attributes.mq_maxmsg =  100;
    my_queue_attributes.mq_msgsize = 16;

    /* Open (create) the queue "my_queue". */
    my_queue_handle =  px5_mq_extend_open("my queue",
                           (O_CREAT | O_RDWR), 0,
                                 &my_queue_attributes,
                                 &my_extended_queue_attribute);

    /* If "my_queue_handle" is positive, the queue was successfully
       created. */
```

px5_mq_extendattr_destroy

C Prototype:

```
#include <mqueue.h>

int px5_mq_extendattr_destroy(mq_extendattr_t * queue_attributes);
```

Description:

This pthreads+ service destroys the previously created extended message queue attributes structure pointed to by *queue_attributes*. Once destroyed, the extended message queue attributes structure cannot be used again unless it is recreated.

API Parameters:

`queue_attributes`	Pointer to the extended message queue attributes to destroy.

Return Codes:

PX5_SUCCESS (0)	Successful extended message queue attributes destroy.
EINVAL	Invalid extended message queue attributes pointer.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

*px5_mq_extend_open, px5_mq_extendattr_init, px5_mq_extendattr_**

Small Example:

```
#include <mqueue.h>

/* Message attribute structure.  */
mq_extendattr_t        my_queue_attributes;
int                    status;




    /* Destroy the extended message queue attributes referenced by
       "my_queue_attributes". */
    status =  px5_mq_extendattr_destroy(&my_queue_attributes);

    /* If status is PX5_SUCCESS, the extended message queue attributes
       structure was destroyed.  */
```

px5_mq_extendattr_getcontroladdr

C Prototype:

```
#include <mqueue.h>

int px5_mq_extendattr_getcontroladdr(mq_extendattr_t* queue_attributes,
                                      void **  queue_control_address);
```

Description:

This pthreads+ service returns the previously supplied message queue control structure address.

API Parameters:

`queue_attributes`	Pointer to the extended message queue attributes.
`queue_control_address`	Pointer to the destination for the previously supplied message queue control address.

Return Codes:

PX5_SUCCESS (0)	Successful message queue control address retrieval.
EINVAL	Invalid extended message queue attributes or message queue control address designation pointer.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

*px5_mq_extend_open, px5_mq_extendattr_init, px5_mq_extendattr_**

Small Example:

```
#include <mqueue.h>

/* Message queue attribute structure.  */
mq_extendattr_t        my_queue_attributes;
void *                 my_queue_control_address;
int                    status;




    /* Get the message queue control structure address in the
       extended message queue attributes structure
       "my_queue_attributes". */
    status = px5_mq_extendattr_getcontroladdr(&my_queue_attributes,
                                              &my_queue_control_address);

    /* If status is PX5_SUCCESS, "my_queue_control_address"
        contains the address of the previously supplied message queue
        control memory.  */
```

px5_mq_extendattr_getcontrolsize

C Prototype:

```
#include <mqueue.h>

int px5_mq_extendattr_getcontrolsize(mq_extendattr_t* queue_attributes,
                                     size_t *  queue_control_size);
```

Description:

This pthreads+ service returns the size of the internal message queue control structure. The main purpose of this API is to inform the application how much memory is required for the *px5_mq_extendattr_setcontroladdr* API.

API Parameters:

`queue_attributes`	Pointer to the attributes.
`queue_control_size`	Pointer to the destination for the internal message queue control structure size.

Return Codes:

PX5_SUCCESS (0)	Successful retrieval of internal message queue control structure size.
EINVAL	Invalid extended message queue attributes or invalid destination for message queue control structure size.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

*px5_mq_extend_open,px5_mq_extendattr_init, px5_mq_extendattr_setcontroladdr, px5_mq_extendattr_**

Small Example:

```
#include <mqueue.h>

/* Message queue extended attribute structure.  */
mq_extendattr_t        my_queue_attributes;
size_t                 my_queue_control_size;
int                    status;



    /* Get the internal message queue control structure memory size. */
    status = px5_mq_extendattr_getcontrolsize(&my_queue_attributes,
                                 &my_queue_control_size);

    /* If status is PX5_SUCCESS, "my_queue_control_size"
       contains the size of the internal message queue control
       structure.  */
```

px5_mq_extendattr_getqueueaddr

C Prototype:

```
#include <mqueue.h>

int px5_mq_extendattr_getqueueaddr(mq_extendattr_t* queue_attributes,
                                   void **  queue_memory_address);
```

Description:

This pthreads+ service returns the previously supplied message queue memory address.

API Parameters:

`queue_attributes`	Pointer to the extended message queue attributes.
`queue_memory_address`	Pointer to the destination for the previously supplied message queue memory address.

Return Codes:

PX5_SUCCESS (0)	Successful message queue memory address retrieval.
EINVAL	Invalid extended message queue attributes or message queue memory address designation pointer.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

*px5_mq_extend_open,px5_mq_extendattr_init, px5_mq_extendattr_setqueueaddr, px5_mq_extendattr_**

Small Example:

```
#include <mqueue.h>

/* Message queue attribute structure.  */
mq_extendattr_t        my_queue_attributes;
void *                 my_queue_memory_address;
int                    status;



    /* Get the message queue memory address in the
       extended message queue attributes structure
       "my_queue_attributes". */
    status = px5_mq_extendattr_getqueueaddr(&my_queue_attributes,
                                             &my_queue_memory_address);

    /* If status is PX5_SUCCESS, "my_queue_memory_address"
        contains the address of the previously supplied message queue
        memory.  */
```

px5_mq_extendattr_getqueuesize

C Prototype:

```
#include <mqueue.h>

int px5_mq_extendattr_getqueuesize(mq_extendattr_t* queue_attributes,
                                   size_t *  queue_memory_size);
```

Description:

This pthreads+ service returns the size of the previously supplied message queue memory.

API Parameters:

`queue_attributes`	Pointer to the attributes.
`queue_memory_size`	Pointer to the destination for the previously supplied message queue memory size.

Return Codes:

PX5_SUCCESS (0)	Successful retrieval of message queue memory size.
EINVAL	Invalid extended message queue attributes or invalid destination for message queue memory size.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

*px5_mq_extend_open,px5_mq_extendattr_init, px5_mq_extendattr_setqueueaddr, px5_mq_extendattr_**

Small Example:

```
#include <mqueue.h>

/* Message queue extended attribute structure.  */
mq_extendattr_t        my_queue_attributes;
size_t                 my_queue_memory_size;
int                    status;



    /* Get the message queue memory size. */
    status = px5_mq_extendattr_getqueuesize(&my_queue_attributes,
                                 &my_queue_memory_size);

    /* If status is PX5_SUCCESS, "my_queue_memory_size"
       contains the size of the message queue memory.  */
```

px5_mq_extendattr_init

C Prototype:

```
#include <mqueue.h>

int px5_mq_extendattr_init(mq_extendattr_t * queue_attributes);
```

Description:

This pthreads+ service initializes the extended message queue attributes structure with default condition variable creation values. Note that the extended message queue attributes are used only by the *px5_mq_extend_open* API.

API Parameters:

`queue_attributes`	Pointer to the extended message queue attributes structure to create.

Return Codes:

PX5_SUCCESS (0)	Successful extended message queue attributes structure creation.
EINVAL	Invalid extended message queue attributes pointer.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

*px5_mq_extend_open, px5_mq_extendattr_destroy, px5_mq_extendattr_**

Small Example:

```
#include <mqueue.h>

/* Extended message queue attribute structure.  */
mq_extendattr_t        my_queue_attributes;
int                    status;




    /* Create the extended message queue attributes structure
      “my_queue_attributes”. */
    status =  px5_mq_extendattr_init(&my_queue_attributes);

    /* If status is PX5_SUCCESS, the “my_queue_attributes”
       structure is ready for use.  */
```

px5_mq_extendattr_setcontroladdr

C Prototype:

```
#include <mqueue.h>

int px5_mq_extendattr_setcontroladdr(mq_extendattr_t* queue_attributes,
                                     void *   queue_control_address,
                                     size_t   queue_memory_size);
```

Description:

This pthreads+ service provides a mechanism for the user to provide the memory for the internal PX5 RTOS message queue structure, as specified by the address contained in the *queue_control_address* parameter. This memory will subsequently be used for the next message queue created with this attribute structure. The size of the memory required for the internal message queue control structure can be found via a call to the *px5_mq_extendattr_getcontrol_size* service.

Note that each message queue created must have its own unique message queue control structure memory. Hence, the message queue control memory supplied here is only valid for one px5_mq_extend_open call.

API Parameters:

`queue_attributes`	Pointer to the message queue attributes.
`queue_control_address`	Pointer to the internal message queue control structure memory.
`queue_control_size`	Size of specified message queue control structure memory.

Return Codes:

PX5_SUCCESS (0)	Successful specification of message queue structure memory.
EINVAL	Invalid message queue attributes or invalid size of message queue control memory.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

*px5_mq_extend_open,px5_mq_extendattr_getcontrolsize, px5_mq_extendattr_**

Small Example:

```
#include <mqueue.h>

/* Message queue attribute structure.  */
mq_extendattr_t        my_queue_attributes;
int                    status;



    /* Set the message queue control structure memory address in the
       extended message queue attributes
       structure "my_queue_attributes". */
    status =  px5_mq_extendattr_setcontroladdr(&my_queue_attributes,
                                               0x80000, 100);
```

px5_mq_extendattr_setqueueaddr

C Prototype:

```
#include <mqueue.h>

int px5_mq_extendattr_setqueueaddr(mq_extendattr_t* queue_attributes,
                                   void *   queue_memory_address,
                                   size_t   queue_memory_size);
```

Description:

This pthreads+ service sets the internal message queue memory address to the address specified by *queue_memory_address*. This address will subsequently be used to supply the memory for the internal message queue memory on the next queue created with this attribute structure.

Note that each message queue created must have its own unique queue memory area. Hence, the message queue memory address supplied here is only valid for one px5_mq_extend_open call.

API Parameters:

`queue_attributes`	Pointer to the message queue attributes.
`queue_memory_address`	Pointer to the message queue memory address.
`queue_memory_size`	Size of specified message queue memory.

Return Codes:

PX5_SUCCESS (0)	Successful specification of message queue memory.
EINVAL	Invalid message queue attributes or invalid size of message queue memory.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

*px5_mq_extend_open,px5_mq_extendattr_init, px5_mq_extendattr_**

Small Example:

```
#include <mqueue.h>

/* Message queue attribute structure.  */
mq_extendattr_t         my_queue_attributes;
int                     status;



    /* Set the message queue memory address in the
       extended message queue attributes
       structure "my_queue_attributes". */
    status =  px5_mq_extendattr_setqueueaddr(&my_queue_attributes,
                                                 0x90000, 1024);
```

px5_pthread_attr_getcontroladdr

C Prototype:

```
#include <pthread.h>

int  px5_pthread_attr_getcontroladdr(pthread_attr_t *attributes,
                                      void **  thread_control_address);
```

Description:

This pthreads+ service returns the thread control structure address stored in the thread attributes structure. By default, this value is NULL unless specified by the application via a call to *px5_pthread_attr_setcontroladdr*.

API Parameters:

`attributes`	Pointer to a previously initialized attributes structure.
`thread_control_address`	Pointer to the destination of where to return the thread control address of this attributes structure.

Return Codes:

PX5_SUCCESS (0)	Successful retrieval of the thread control address.
EINVAL	Attributes structure pointer or thread control address destination pointer is invalid.

Real-time Scenarios:

Upon the successful execution of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

pthread_attr_*, pthread_attr_setcontroladdr

Small Example:

```
#include <pthread.h>

pthread_attr_t    my_thread_attributes;
void *            thread_control_address;
int               status;




/* Get the thread control address in the previously initialized
   attributes.  */
status =     px5_pthread_attr_getcontroladdr(&my_thread_attributes,
                                         &thread_control_address);

/* If status contains PX5_SUCCESS, the thread control address value in
   this attribute set is contained in "thread_control_address".  */
```

px5_pthread_attr_getcontrolsize

C Prototype:

```
#include <pthread.h>

int  px5_pthread_attr_getcontrolsize(pthread_attr_t *attributes,
                                     size_t *   thread_control_size);
```

Description:

This pthreads+ service returns the size of the internal thread control structure. The main purpose of this API is to inform the application how much memory is required for the *px5_pthread_attr_setcontroladdr* API.

API Parameters:

`attributes`	Pointer to a previously initialized attributes structure.
`thread_control_size`	Pointer to the destination of where to return the internal thread control structure size.

Return Codes:

PX5_SUCCESS (0)	Successful retrieval of the internal thread control size.
EINVAL	Attributes pointer or internal thread control size destination pointer is invalid.

Real-time Scenarios:

Upon the successful execution of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

pthread_attr_*, pthread_attr_getcontroladdr, pthread_attr_setcontroladdr

Small Example:

```
#include <pthread.h>

pthread_attr_t    my_thread_attributes;
size_t            thread_control_size;
int               status;




/* Get the internal thread control size.  */
status =  px5_pthread_attr_getcontrolsize(&my_thread_attributes,
                                          &thread_control_size);

/* If status contains PX5_SUCCESS, the internal thread control
   structure size is found in "thread_control_size". */
```

px5_pthread_attr_getname

C Prototype:

```
#include <pthread.h>

int  px5_pthread_attr_getname(pthread_attr_t *attributes,
                                           Char **     return_name);
```

Description:

This pthreads+ service returns the current thread name in the thread attributes structure.

API Parameters:

`attributes`	Pointer to a previously initialized attributes structure.
`return_name`	Pointer to the destination of where to return the name in this attributes structure.

Return Codes:

PX5_SUCCESS (0)	Successful retrieval of the thread name.
EINVAL	Attributes structure pointer or name destination pointer is invalid.

Real-time Scenarios:

Upon the successful execution of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

pthread_attr_*, pthread_attr_setname

Small Example:

```
#include <pthread.h>

pthread_attr_t    my_thread_attributes;
char *            my_name;
int               status;




    /* Get the thread name in the previously initialized
       attributes.  */
    status =  px5_pthread_attr_getname(&my_thread_attributes,
                                       &my_name);

    /* If status contains PX5_SUCCESS, the thread name in this
       attribute set is stored in "my_name". */
```

px5_pthread_attr_getpriority

C Prototype:

```
#include <pthread.h>

int  px5_pthread_attr_getpriority(pthread_attr_t *attributes,
                                        int *     priority);
```

Description:

This pthreads+ service returns the current thread priority in the thread attributes structure. By default, this value is *PX5_DEFAULT_PRIORITY* unless specified by the application via a call to *px5_pthread_attr_setpriority*.

API Parameters:

`attributes`	Pointer to a previously initialized attributes structure.
`priority`	Pointer to the destination of where to return the priority of this attributes structure.

Return Codes:

PX5_SUCCESS (0)	Successful retrieval of the thread priority.
EINVAL	Attributes structure pointer or priority destination pointer is invalid.

Real-time Scenarios:

Upon the successful execution of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

pthread_attr_*, px5_pthread_attr_setpriority

Small Example:

```
#include <pthread.h>

pthread_attr_t    my_thread_attributes;
int               priority;
int               status;



    /* Get the priority in the previously initialized
       attributes.  */
    status =  px5_pthread_attr_getpriority(&my_thread_attributes,
                                           &priority);

    /* If status contains PX5_SUCCESS, the priority in this
       attribute set is stored in "priority". */
```

px5_pthread_attr_gettimeslice

C Prototype:

```
#include <pthread.h>

int  px5_pthread_attr_gettimeslice(pthread_attr_t *attributes,
                                                  u_long *   thread_time_slice);
```

Description:

This pthreads+ service returns the current thread time-slice stored in the thread attributes structure. By default, this value is 0, which means time-slicing is disabled. The time-slice value for thread creation can be changed via the *px5_pthread_attr_settimeslice* service.

Each thread may have its own unique time-slice value through specific invocation of px5_pthread_attr_settimeslice.

API Parameters:

`attributes`	Pointer to a previously initialized attributes structure.
`thread_time_slice`	Pointer to the destination of where to return the thread time-slice value of this attributes structure.

Return Codes:

PX5_SUCCESS (0)	Successful retrieval of the thread per time-slice.
EINVAL	Attributes structure pointer or thread time-slice destination pointer is invalid.

Real-time Scenarios:

Upon the successful execution of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

pthread_attr_*, pthread_attr_settimeslice

Small Example:

```
#include <pthread.h>

pthread_attr_t    my_thread_attributes;
u_long            thread_time_slice;
int               status;




/* Get the thread time-slice in the previously initialized
   set of attributes.  */
status =  px5_pthread_attr_gettimeslice(&my_thread_attributes,
                                             &thread_time_slice);

/* If status contains PX5_SUCCESS, the thread time-slice value in this
   attribute set is stored in "thread_time_slice". */
```

px5_pthread_attr_setcontroladdr

C Prototype:

```
#include <pthread.h>

int  px5_pthread_attr_setcontroladdr(pthread_attr_t *attributes,
                                     void *   thread_control_address,
                                     size_t   thread_control_size);
```

Description:

This pthreads+ service provides a mechanism for the user to provide the memory for the internal PX5 RTOS thread control structure, as specified by the address contained in the *thread_control_address* parameter. This memory will subsequently be used for the next thread created with this attribute structure. The size of the memory required for the internal thread control structure can be found via a call to the *px5_pthread_attr_getcontrolsize* service.

Note that each thread created must have its own unique thread control structure memory. Hence, the thread control memory supplied here is only valid for one pthread_create call.

API Parameters:

`attributes`	Pointer to a previously initialized attributes structure.
`thread_control_address`	Thread control address to use for the next thread creation.
`thread_control_size`	Size of memory pointed to by the thread control address.

Return Codes:

PX5_SUCCESS (0)	Successful setting of the thread control address.
EINVAL	Attributes pointer or thread control address pointer or size is invalid.

Real-time Scenarios:

Upon the successful execution of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

pthread_attr_*, px5_pthread_attr_getcontroladdr, px5_pthread_attr_getcontrolsize

Small Example:

```
#include <pthread.h>

pthread_attr_t    my_thread_attributes;
int               status;




 /* Set the thread control address to the absolute address of
    0x40000 in the previously initialized attributes.  */
 status =  px5_pthread_attr_setcontroladdr(&my_thread_attributes,
                                              (void *) 0x40000, 300);

 /* If status contains PX5_SUCCESS, the thread control address in this
    attribute structure is 0x40000. */
```

px5_pthread_attr_setname

C Prototype:

```
#include <pthread.h>

int  px5_pthread_attr_setname(pthread_attr_t *attributes,
                                             char *   name);
```

Description:

This pthreads+ service stores the specified thread name (ASCII string) in the thread attributes structure.

API Parameters:

`attributes`	Pointer to a previously initialized attributes structure.
`name`	Thread name in ASCII string format.

Return Codes:

PX5_SUCCESS (0)	Successful setting of the thread name.
EINVAL	Attributes pointer or specified thread name pointer is invalid.

Real-time Scenarios:

Upon the successful execution of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

pthread_attr_*, pthread_attr_getname

Small Example:

```
#include <pthread.h>

pthread_attr_t    my_thread_attributes;
char              my_name[] =  “my thread name”;
int               status;



    /* Set the thread name to “my thread name” in the previously
       initialized attributes.  */
    status =  px5_pthread_attr_setname(&my_thread_attributes,
                                       my_name);

    /* If status contains PX5_SUCCESS, the attributes structure now
       has the name of “my thread name”. */
```

px5_pthread_attr_setpriority

C Prototype:

```
#include <pthread.h>

int  px5_pthread_attr_setpriority(pthread_attr_t *attributes,
                                            int    priority);
```

Description:

This pthreads+ service stores the specified priority in the thread attributes structure. The thread priority specified must be between 0 and (*PX5_MAXIMUM_PRIORITIES* – 1), where larger numbers represent higher priority.

API Parameters:

`attributes`	Pointer to a previously initialized attributes structure.
`priority`	Priority to store in the attributes structure.

Return Codes:

PX5_SUCCESS (0)	Successful setting of the priority.
EINVAL	Attributes pointer or specified priority is invalid.

Real-time Scenarios:

Upon the successful execution of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

pthread_attr_*, px5_pthread_attr_getpriority

Small Example:

```
#include <pthread.h>

pthread_attr_t    my_thread_attributes;
int               status;



    /* Set the priority to 31 in the previously initialized
        attributes.  */
    status =  px5_pthread_attr_setpriority(&my_thread_attributes, 31);

    /* If status contains PX5_SUCCESS, the priority in the
       attributes structure is now 31. */
```

px5_pthread_attr_settimeslice

C Prototype:

```
#include <pthread.h>

int  px5_pthread_attr_settimeslice(pthread_attr_t *attributes,
                                                 u_long   time_slice);
```

Description:

This pthreads+ service stores the specified time-slice in the thread attributes structure. A value of 0 disables time-slicing (default), while positive values represent the number of timer ticks the thread can execute before giving other threads of the same priority a chance to execute.

API Parameters:

`attributes`	Pointer to a previously initialized attributes structure.
`time_slice`	Time-slice to store in the attributes structure.

Return Codes:

PX5_SUCCESS (0)	Successful setting of the thread time-slice.
EINVAL	Attributes structure pointer is invalid.

Real-time Scenarios:

Upon the successful execution of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

pthread_attr_*, px5_pthread_attr_gettimeslice

Small Example:

```
#include <pthread.h>

pthread_attr_t    my_thread_attributes;
int               status;


    /* Set the thread time-slice to 2 in the previously initialized
        attributes structure.  */
    status =  px5_pthread_attr_settimeslice(&my_thread_attributes, 2);

    /* If status contains PX5_SUCCESS, the time-slice in the
       attributes structure is now 2. */
```

px5_pthread_condattr_getcontroladdr

C Prototype:

```
#include <pthread.h>

int px5_pthread_condattr_getcontroladdr(pthread_condattr_t
       *  condition_var_attributes,
             void **  condition_var_control_address);
```

Description:

This pthreads+ service returns the previously supplied condition variable control structure address.

API Parameters:

`condition_var_attributes`	Pointer to the condition variable attributes.
`Condition_var_control_address`	Pointer to the destination for the previously supplied condition variable control address.

Return Codes:

PX5_SUCCESS (0)	Successful condition variable attributes condition control address retrieval.
EINVAL	Invalid condition variable attributes or condition variable control address designation pointer.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

pthread_cond_*, pthread_condattr_*, px5_pthread_condattr_setcontroladdr, px5_pthread_condattr_getcontrolsize

Small Example:

```
#include <pthread.h>

/* Condition variable attribute structure.  */
pthread_condattr_t      my_cond_var_attributes;
void *                  my_cond_var_control_address;
int                     status;



  /* Get the condition variable control structure address in the
     condition variable attributes structure
     "my_cond_var_attributes". */
  status = px5_pthread_condattr_getcontroladdr(&my_cond_var_attributes,
                                     &my_cond_var_control_address);

    /* If status is PX5_SUCCESS, "my_cond_var_control_address"
       contains the address of the previously supplied condition
       variable control memory.  */
```

px5_pthread_condattr_getcontrolsize

C Prototype:

```
#include <pthread.h>

int px5_pthread_condattr_getcontrolsize(pthread_condattr_t
                    *  condition_var_attributes,
                    size_t *  condition_var_control_size);
```

Description:

This pthreads+ service returns the size of the internal condition variable control structure. The main purpose of this API is to inform the application how much memory is required for the *px5_pthread_condattr_setcontroladdr* API.

API Parameters:

`condition_var_attributes`	Pointer to the condition variable attributes.
`condition_var_control_size`	Pointer to the destination for the internal condition variable control structure size.

Return Codes:

PX5_SUCCESS (0)	Successful retrieval of internal condition variable control structure size.
EINVAL	Invalid condition variable attributes or invalid destination for condition variable control structure size.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NO PREEMPTION. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

pthread_cond_*, pthread_condattr_*, px5_pthread_condattr_setcontroladdr, px5_pthread_condattr_getcontroladdr

Small Example:

```
#include <pthread.h>

/* Condition variable attribute structure.  */
pthread_condattr_t     my_cond_var_attributes;
size_t                 my_cond_var_control_size;
int                    status;



    /* Get the condition variable control structure memory size. */
    status = px5_pthread_condattr_getcontrolsize(
             &my_cond_var_attributes,&my_cond_var_control_size);

    /* If status is PX5_SUCCESS, "my_cond_var_control_size"
       contains the size of the internal condition variable control
       structure.  */
```

px5_pthread_condattr_getname

C Prototype:

```
#include <pthread.h>

int px5_pthread_condattr_getname(pthread_condattr_t
                     *  condition_var_attributes,
                     char **  condition_var_name);
```

Description:

This pthreads+ service returns the previously supplied condition variable name.

API Parameters:

`condition_var_attributes`	Pointer to the condition variable attributes.
`Condition_var_name`	Pointer to the destination for the previously supplied condition variable name.

Return Codes:

PX5_SUCCESS (0)	Successful retrieval of last supplied condition variable name.
EINVAL	Invalid condition variable attributes or name destination pointer.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

pthread_cond_*, pthread_condattr_*, px5_pthread_condattr_setname

Small Example:

```
#include <pthread.h>

/* Condition variable attribute structure.  */
pthread_condattr_t     my_cond_var_attributes;
char *                 my_cond_var_name;
int                    status;



    /* Get the last supplied condition variable name. */
    status =  px5_pthread_condattr_getname(&my_cond_var_attributes,
                                           &my_cond_var_name);

    /* If status is PX5_SUCCESS, "my_cond_var_name" contains the
       name previously supplied. */
```

px5_pthread_condattr_setcontroladdr

C Prototype:

```
#include <pthread.h>

int px5_pthread_condattr_setcontroladdr(pthread_condattr_t
              *  condition_var_attributes,
              void *  condition_var_control_address,
              size_t  condition_var_control_size);
```

Description:

This pthreads+ service provides a mechanism for the user to provide the memory for the internal PX5 RTOS condition variable structure, as specified by the address contained in the *condition_var_control_address* parameter. This memory will subsequently be used for the next condition variable created with this attribute structure. The size of the memory required for the internal condition variable control structure can be found via a call to the *px5_pthread_condattr_getcontrolsize* service.

Note that each condition variable created must have its own unique condition variable control structure memory. Hence, the condition variable control memory supplied here is only valid for one pthread_cond_init call.

API Parameters:

`condition_var_attributes`	Pointer to the condition variable attributes.
`condition_var_control_address`	Pointer to the supplied condition variable control structure memory.
`condition_var_control_size`	Size of specified condition variable control structure memory.

Return Codes:

PX5_SUCCESS (0)	Successful specification of condition variable structure memory.
EINVAL	Invalid condition variable attributes or invalid size of condition variable control memory.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NO PREEMPTION. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

pthread_cond_*, pthread_condattr_*, pthread_condattr_getcontroladdr, pthread_condattr_getcontrolsize

Small Example:

```
#include <pthread.h>

/* Condition variable attribute structure.  */
pthread_condattr_t     my_cond_var_attributes;
int                    status;



    /* Set the condition variable control structure address in the
        condition variable attributes structure
       “my_cond_var_attributes”. */
    status =  pthread_condattr_setcontroladdr(&my_cond_var_attributes,
                                   0x60000, 60);

    /* If status is PX5_SUCCESS, the condition variable creation
       using these attributes will use address 0x60000 for the internal
       condition variable control structure. */
```

px5_pthread_condattr_setname

C Prototype:

```
#include <pthread.h>

int px5_pthread_condattr_setname(pthread_condattr_t
                    *  condition_var_attributes,
                    char *  cond_var_name);
```

Description:

This pthreads+ service sets the condition variable name in the specified attribute structure.

API Parameters:

`condition_var_attributes`	Pointer to the condition variable attributes.
`condition_var_name`	Pointer to the supplied condition variable name.

Return Codes:

PX5_SUCCESS (0)	Successful condition variable name set.
EINVAL	Invalid condition variable attributes.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

pthread_cond_*, pthread_condattr_*, px5_pthread_condattr_getname

Small Example:

```
#include <pthread.h>

/* Condition variable attribute structure.  */
pthread_condattr_t     my_cond_var_attributes;
int                    status;



    /* Set the condition variable name in the condition variable
       attributes structure "my_cond_var_attributes". */
    status =  px5_pthread_condattr_setname(&my_cond_var_attributes,
                                            "my_cond_var_name");

    /* If status is PX5_SUCCESS, "my_cond_var_name" is set in the
       condition variable attribute structure.  */
```

px5_pthread_event_flags_clear

C Prototype:

```
#include <pthread.h>

int px5_pthread_event_flags_clear(pthread_event_flags_t *
                                              event_flags_handle);
```

Description:

This pthreads+ service clears all events flags in the specified event flags group.

API Parameters:

`event_flags_handle`	Handle of the event flags group.

Return Codes:

PX5_SUCCESS (0)	Successful event flags clear.
EINVAL	Invalid event flags handle pointer or any/all events option.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. No preemption is possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

px5_pthread_event_flags_*, px5_pthread_event_flags_set

Small Example:

```
#include <pthread.h>

/* Event flags handle.  */
pthread_event_flags_t    my_event_flags_handle;
int                      status;



    /* Clear all event flags in the event flags group
       "my_event_flags_handle". */
    status =  px5_pthread_event_flags_clear(&my_event_flags_handle);

    /* If status is PX5_SUCCESS, all event flags are cleared.  */
```

px5_pthread_event_flags_create

C Prototype:

```
#include <pthread.h>

int px5_pthread_event_flags_create(pthread_event_flags_t *
             event_flags_handle,
             pthread_event_flagsattr_t *  event_flags_attributes);
```

Description:

This pthreads+ service initializes (creates) an event flags group with the optional event flags attributes. If successful, the event flags handle is setup for further use by the application.

API Parameters:

`event_flags_handle`	Handle of the event flags to create.
`event_flags_attributes`	Optional event flags attributes. This value is NULL if no event flags attributes are specified.

Return Codes:

PX5_SUCCESS (0)	Successful event flags initialization.
EINVAL	Invalid event flags handle pointer or event flags attributes.
EBUSY	Event flags is already created.
ENOMEM	Insufficient memory to create event flags.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

px5_pthread_event_flagsattr_*, px5_pthread_event_flags_destroy

Small Example:

```
#include <pthread.h>

/* Event flags handle.  */
pthread_event_flags_t    my_event_flags_handle;
int                      status;



    /* Create the event flags group, the handle of which is returned in
       "my_event_flags_handle". */
    status =  px5_pthread_event_flags_create(&my_event_flags_handle,
                                             NULL);

    /* If status is PX5_SUCCESS, the event flags group was created, and
        the event flags handle is ready to use.  */
```

px5_pthread_event_flags_destroy

C Prototype:

```
#include <pthread.h>

int px5_pthread_event_flags_destroy(pthread_event_flags_t *
                                            event_flags_handle);
```

Description:

This pthreads+ service destroys the previously created event flags group specified by *event_flags_handle*. If the event flags group has any threads waiting for events, an error is returned.

API Parameters:

`event_flags_handle`	Handle of the event flags to destroy.

Return Codes:

PX5_SUCCESS (0)	Successful event flags group destroy.
EINVAL	Invalid event flags handle.
EBUSY	A thread currently is suspended on the event flags.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

px5_pthread_event_flags_*, px5_pthread_event_flags_create

Small Example:

```
#include <pthread.h>

/* Event flags handle.  */
pthread_event_flags_t      my_event_flags_handle;
int                        status;




    /* Destroy the event flags referenced by
       "my_evemt_flags_handle". */
    status =  px5_pthread_event_flags_destroy(&my_event_flags_handle);

    /* If status is PX5_SUCCESS, the event flags group was destroyed.
*/
```

px5_pthread_event_flags_set

C Prototype:

```
#include <pthread.h>

int px5_pthread_event_flags_set(pthread_event_flags_t *
                     event_flags_handle, u_long  events_to_set);
```

Description:

This pthreads+ service sets all the event flags specified in *events_to_set* in the event flags group identified by *event_flags_handle*. All threads suspended on this event flags group that have their event flags request satisfied are resumed.

API Parameters:

`event_flags_handle`	Handle of the event flags group.
`events_to_set`	Bit map of event flags to set.

Return Codes:

PX5_SUCCESS (0)	Successful event flags set.
EINVAL	Invalid event flags group handle pointer.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NO PREEMPTION. If there are no other threads waiting for event flags, no preemption takes place.

PREEMPTION. If a higher-priority thread was waiting for event flags and their request is satisfied, those threads are resumed, and preemption will occur.

Callable From:

This service is callable from the thread context and from interrupt handlers (ISRs).

See Also:

px5_pthread_event_flags_*, px5_pthread_event_flags_wait

Small Example:

```
#include <pthread.h>

/* Event flags handle.  */
pthread_event_flags_t    my_event_flags_handle;
int                      status;




    /* Set event flags 7 and 3 (bits 7 and 3) in the event flags group
       "my_event_flags_handle". */
    status = px5_pthread_event_flags_set(&my_event_flags_handle, 0x88);

    /* If status is PX5_SUCCESS, event flags 7 and 3 are now set. */
```

px5_pthread_event_flags_trywait

C Prototype:

```
#include <pthread.h>

int px5_pthread_event_flags_trywait(pthread_event_flags_t *
                           event_flags_handle, u_long requested_events,
                    int all_or_any, u_long *  received_events);
```

Description:

This pthreads+ service attempts to retrieve the event flags specified by the *requested_events* parameter and in the manner specified by the *all_or_any* parameter from the specified event flags group. If the requested events are not available, this service returns an error.

API Parameters:

`event_flags_handle`	Handle of the event flags group.
`requested_events`	Requested event flags.
`all_or_any`	This parameter specifies if all the event flags are required (*PTHREAD_ALL_EVENTS*), or any of the events specified are required (*PTHREAD_ANY_EVENT*) to satisfy the request.
`received_events`	Optional destination of where to return the actual event flags that satisfied the request.

Return Codes:

PX5_SUCCESS (0)	Successful event flags retrieval.
EINVAL	Invalid event flags handle pointer or any/all events option.
EAGAIN	Requested event flags are not available.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NO PREEMPTION. No preemption is possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

px5_pthread_event_flags_*, px5_pthread_event_flags_wait

Small Example:

```
#include <pthread.h>

/* Event flags handle.  */
pthread_event_flags_t    my_event_flags_handle;
int                      status;



    /* Attempt to get either flag 3 or flag 7 in the event flags group
       “my_event_flags_handle”. */
    status =  px5_pthread_event_flags_trywait(&my_event_flags_handle,
                                  0x88, PTHREAD_ANY_EVENT, NULL);

    /* If status is PX5_SUCCESS, the either or both event flag 3 and 7
       were set.  */
```

px5_pthread_event_flags_wait

C Prototype:

```
#include <pthread.h>

int px5_pthread_event_flags_wait(pthread_event_flags_t *
                    event_flags_handle, u_long requested_events,
                    int all_or_any, u_long *  received_events);
```

Description:

This pthreads+ service retrieves the event flags specified by the *requested_events* parameter and in the manner specified by the *all_or_any* parameter from the specified event flags group. If the requested events are not available, this service suspends the calling thread.

This API is a cancellation point, meaning that if a cancellation is pending, it will be detected and executed by this API.

API Parameters:

`event_flags_handle`	Handle of the event flags group.
`requested_events`	Requested event flags.
`all_or_any`	This parameter specifies if all the event flags are required (*PTHREAD_ALL_EVENTS*), or any of the events specified are required (*PTHREAD_ANY_EVENT*) to satisfy the request.
`received_events`	Optional destination of where to return the actual event flags that satisfied the request.

Return Codes:

PX5_SUCCESS (0)	Successful event flags wait.
EINVAL	Invalid event flags handle pointer or any/all events option.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. If the event flags were available, no preemption takes place.

S **SUSPENSION**. If the event flags are not present, the calling thread is suspended until the event flags become available.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

px5_pthread_event_flags_*, px5_pthread_event_flags_set

Small Example:

```
#include <pthread.h>

/* Event flags handle.  */
pthread_event_flags_t    my_event_flags_handle;
int                      status;



    /* Get either flag 3 or flag 7 in the event flags group
       "my_event_flags_handle". */
    status =  px5_pthread_event_flags_wait(&my_event_flags_handle,
                                           0x88, PTHREAD_ANY_EVENT, NULL);

    /* If status is PX5_SUCCESS, the either or both event flag 3 and 7
       were set.  */
```

px5_pthread_event_flagsattr_destroy

C Prototype:

```
#include <pthread.h>

int px5_pthread_event_flagsattr_destroy(pthread_event_flagsattr_t *
                                         event_flags_attributes);
```

Description:

This pthreads+ service destroys the previously created event flags attributes structure pointed to by *event_flags_attributes*. Once destroyed, the event flags attributes structure cannot be used again unless it is recreated.

API Parameters:

`event_flags_attributes`	Pointer to the event flags attributes to destroy.

Return Codes:

PX5_SUCCESS (0)	Successful event flags attributes destroy.
EINVAL	Invalid event flags attributes pointer.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

px5_pthread_event_flags_*, px5_pthread_event_flagsattr_*, px5_pthread_event_flagsattr_init

Small Example:

```
#include <pthread.h>

/* Event flags attribute structure.  */
pthread_event_flagsattr_t    my_event_flags_attributes;
int                          status;


    /* Destroy the event flags attributes referenced by
       "my_event_flags_attributes". */
    status =  px5_pthread_event_flagsattr_destroy(
                                 &my_event_flags_attributes);

    /* If status is PX5_SUCCESS, the event flags attributes structure
       was destroyed.  */
```

px5_pthread_event_flagsattr_getcontroladdr

C Prototype:

```
#include <pthread.h>

int
px5_pthread_event_flagsattr_getcontroladdr(pthread_event_flagsattr_t
       * event_flags_attributes, void **  event_flags_control_address);
```

Description:

This pthreads+ service returns the previously supplied event flags control structure memory address.

API Parameters:

`event_flags_attributes`	Pointer to the event flags attributes.
`event_flags_control_address`	Pointer to the destination for the previously supplied event flags control address.

Return Codes:

PX5_SUCCESS (0)	Successful event flags attributes event flags control address retrieval.
EINVAL	Invalid event flags attributes, or event flags control address designation pointer.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

px5_pthread_event_flags_*, px5_pthread_event_flagsattr_*, px5_pthread_event_flagsattr_setcontroladdr, px5_pthread_event_flagsattr_getcontrolsize

Small Example:

```
#include <pthread.h>

/* Event flags attribute structure.  */
pthread_event_flagsattr_t  my_event_flags_attributes;
void *                     my_event_flags_control_address;
int                        status;




    /* Get the event flags control structure address in the
       attributes structure “my_event_flags_attributes”. */
    status = px5_pthread_event_flagsattr_getcontroladdr(
                    &my_event_flags_attributes,
                    &my_event_flags_control_address);

    /* If status is PX5_SUCCESS, “my_event_flags_control_address”
        contains the address of the previously supplied event flags
        control memory.  */
```

px5_pthread_event_flagsattr_getcontrolsize

C Prototype:

```
#include <pthread.h>

int
px5_pthread_event_flagsattr_getcontrolsize(pthread_event_flagsattr_t
       * event_flags_attributes, size_t *  event_flags_control_size);
```

Description:

This pthreads+ service returns the size of the internal event flags control structure. The main purpose of this API is to inform the application how much memory is required for the *px5_pthread_event_flagsattr_setcontroladdr* API.

API Parameters:

`event_flags_attributes`	Pointer to the event flags attributes.
`event_flags_control_size`	Pointer to the destination for the internal event flags control structure size.

Return Codes:

PX5_SUCCESS (0)	Successful retrieval of internal event flags control structure size.
EINVAL	Invalid event flags attributes or event flags control structure size destination pointer.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

px5_pthread_event_flags_*, px5_pthread_event_flagsattr_*, px5_pthread_event_flagsattr_setcontroladdr

Small Example:

```
#include <pthread.h>

/* Event flags attribute structure.  */
pthread_event_flagsattr_t  my_event_flags_attributes;
size_t                     my_event_flags_control_size;
int                        status;


    /* Get the event flags control structure memory size. */
    status = px5_pthread_event_flagsattr_getcontrolsize(
       &my_event_flags_attributes, &my_event_flags_control_size);

    /* If status is PX5_SUCCESS, "my_event_flags_control_size"
        contains the size of the internal event flags
        control structure.  */
```

px5_pthread_event_flagsattr_getname

C Prototype:

```
#include <pthread.h>

int px5_pthread_event_flagsattr_getname(pthread_event_flagsattr_t
             * event_flags_attributes, char **  event_flags_name);
```

Description:

This pthreads+ service returns the previously supplied event flags name.

API Parameters:

`event_flags_attributes`	Pointer to the event flags attributes.
`event_flags_name`	Pointer to the destination for the previous event flags name.

Return Codes:

PX5_SUCCESS (0)	Successful event flags attributes event flags name retrieval.
EINVAL	Invalid event flags attributes or event flags name pointer.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

*px5_pthread_event_flags_*, px5_pthread_event_flagsattr_**

Small Example:

```
#include <pthread.h>

/* Event flags attribute structure.  */
pthread_event_flagsattr_t  my_event_flags_attributes;
char *                     my_event_flags_name;
int                        status;



    /* Get the previous event flags name. */
    status =  px5_pthread_event_flagsattr_getname(
                   &my_event_flags_attributes, &my_event_flags_name);

    /* If status is PX5_SUCCESS, "my_event_flags_name" contains the
       name previously supplied. */
```

px5_pthread_event_flagsattr_init

C Prototype:

```
#include <pthread.h>

int px5_pthread_event_flagsattr_init(pthread_event_flagsattr_t *
                                     event_flags_attributes);
```

Description:

This pthreads+ service initializes the event flags attributes structure with default event flags creation values.

API Parameters:

`event_flags_attributes`	Pointer to the event flags attributes structure to create.

Return Codes:

PX5_SUCCESS (0)	Successful event flags attributes structure creation.
EINVAL	Invalid event flags attributes pointer.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

px5_pthread_event_flags_*, px5_pthread_event_flagsattr_*, px5_pthread_event_flagsattr_destroy

Small Example:

```
#include <pthread.h>

/* Event flags attribute structure.  */
pthread_event_flagsattr_t  my_event_flags_attributes;
int                        status;



    /* Create the event flags attributes structure
       "my_event_flags_attributes". */
    status =
       px5_pthread_event_flagsattr_init(&my_event_flags_attributes);

    /* If status is PX5_SUCCESS, the "my_event_flags_attributes"
       structure is ready for use.  */
```

px5_pthread_event_flagsattr_setcontroladdr

C Prototype:

```
#include <pthread.h>

int
px5_pthread_event_flagsattr_setcontroladdr(pthread_event_flagsattr_t
       * event_flags_attributes, void *  event_flags_control_address,
                                 size_t event_flags_control_size);
```

Description:

This pthreads+ service provides a mechanism for the user to provide the memory for the internal PX5 RTOS event flags structure, as specified by the address contained in the *event_flags_control_address* parameter. This memory will subsequently be used for the next event flags created with this attribute structure. The size of the memory required for the internal event flags control structure can be found via a call to the *px5_pthread_event_flagsattr_getcontrolsize* service.

Note that each event flags created must have its own unique event flags control structure memory. Hence, the event flags control address supplied here is only valid for one px5_pthread_event_flags_create call.

API Parameters:

`event_flags_attributes`	Pointer to the event flags attributes.
`event_flags_control_address`	Pointer to the supplied event flags control structure memory.
`event_flags_control_size`	Size of memory specified.

Return Codes:

PX5_SUCCESS (0)	Successful internal event flags control address set.

EINVAL	Invalid event flags attributes or memory size.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NO PREEMPTION. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

px5_pthread_event_flags_*, px5_pthread_event_flagsattr_*, px5_pthread_event_flagsattr_getcontrolsize

Small Example:

```
#include <pthread.h>

/* Event flags attribute structure.  */
pthread_event_flagsattr_t  my_event_flags_attributes;
int                        status;



    /* Provide the memory for the event flags control structure in the
       next event flags create call by placing it's address in the
       attributes structure "my_event_flags_attributes". */
    status =  px5_pthread_event_flagsattr_setcontroladdr(
              &my_event_flags_attributes, 0x70000, 60);

    /* If status is PX5_SUCCESS, the next event flags creation using
      these attributes will use memory at address 0x70000 for the
      internal event flags control structure. */
```

px5_pthread_event_flagsattr_setname

C Prototype:

```
#include <pthread.h>

int px5_pthread_event_flagsattr_setname(pthread_event_flagsattr_t
              * event_flags_attributes, char *  event_flags_name);
```

Description:

This pthreads+ service sets the event flags name in the specified attribute structure.

API Parameters:

`event_flags_attributes`	Pointer to the event flags attributes.
`event_flags_name`	Pointer to the supplied event flags name.

Return Codes:

PX5_SUCCESS (0)	Successful event flags attributes event flags name set.
EINVAL	Invalid event flags attributes.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

px5_pthread_event_flags_*, px5_pthread_event_flagsattr_*, px5_pthread_event_flagsattr_getname

Small Example:

```
#include <pthread.h>

/* Event flags attribute structure.  */
pthread_event_flagsattr_t  my_event_flags_attributes;
int                        status;


    /* Set the event flags name in the event flags attributes structure
       "my_event_flags_attributes". */
    status =  px5_pthread_event_flagsattr_setname(
                    &my_event_flags_attributes, "my_event_flags_name");

    /* If status is PX5_SUCCESS, "my_event_flags_name" is set in the
       event flags attribute structure.  */
```

px5_pthread_fastqueue_create

C Prototype:

```
#include <pthread.h>

int px5_pthread_fastqueue_create(pthread_fastqueue_t *fastqueue_handle,
             pthread_event_flagsattr_t * fast_queue_attributes,
             size_t message_size, int max_messages);
```

Description:

This pthreads+ service initializes (creates) a fastqueue with the optional fast queue attributes. If successful, the fastqueue handle is setup for further use by the application.

API Parameters:

`Fastqueue_handle`	Handle of the fastqueue to create.
`fastqueue_attributes`	Optional fastqueue attributes. This value is NULL if no fastqueue attributes are specified.
`message_size`	Fixed-size of queue message in bytes (must be evenly divisible by sizeof(u_long)).
`max_messages`	Total number of messages in the queue.

Return Codes:

PX5_SUCCESS (0)	Successful fastqueue creation.
EINVAL	Invalid fastqueue handle pointer or attributes.
EBUSY	Fastqueue is already created.
ENOMEM	Insufficient memory to create fastqueue.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NO PREEMPTION. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

px5_pthread_fastqueueattr_*, px5_pthread_fastqueue_destroy

Small Example:

```
#include <pthread.h>

/* Fastqueue handle.  */
pthread_fastqueue_t      my_fastqueue_handle;
int                      status;



    /* Create a fastqueue that can hold 100 4-byte messages, the handle
       of which is returned in "my_fastqueue_handle". */
    status =  px5_pthread_fastqueue_create(&my_fastqueue_handle, NULL,
                                              100, 4);

    /* If status is PX5_SUCCESS, the event flags group was created, and
        the event flags handle is ready to use.  */
```

px5_pthread_fastqueue_destroy

C Prototype:

```
#include <pthread.h>

int px5_pthread_fastqueue_destroy(pthread_fastqueue_t *
                                         fastqueue_handle);
```

Description:

This pthreads+ service destroys the previously created fastqueue specified by *fastqueue_handle*. If the fastqueue has any threads waiting for a messages, an error is returned.

API Parameters:

`fastqueue_handle`	Handle of the fastqueu to destroy.

Return Codes:

PX5_SUCCESS (0)	Successful fastqueue destroy.
EINVAL	Invalid fastqueue handle.
EBUSY	A thread currently is suspended on the fastqueue.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

px5_pthread_fastqueue_*, px5_pthread_fastqueue_create

Small Example:

```
#include <pthread.h>

/* Fastqueue handle.  */
pthread_fastqueue_t        my_fastqueue_handle;
int                        status;




    /* Destroy the fastqueue referenced by
       "my_fastqueue_handle". */
    status =  px5_pthread_fastqueue_destroy(&my_fastqueue_handle);

    /* If status is PX5_SUCCESS, the fastqueue was destroyed.  */
```

px5_pthread_fastqueue_receive

C Prototype:

```
#include <pthread.h>

int px5_pthread_fastqueue_receive(pthread_fastqueue_t *
      fastqueue_handle, u_long *  message, size_t message_size);
```

Description:

This pthreads+ service receives a message from the specified fastqueue. If the fastqueue is empty, this thread suspends until a message is sent to the fastqueue. If there is a thread waiting on to send to the fastqueue, the thread’s message is placed in the fastqueue and is resumed.

API Parameters:

`fastqueue_handle`	Handle of the fastqueue.
`message`	Pointer to the destination for the message.
`message_size`	Size of the message in bytes (must be evenly divisible by sizeof(u_long)).

Return Codes:

PX5_SUCCESS (0)	Successful fastqueue receive.
EINVAL	Invalid fastqueue handle pointer.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. If there are one or more messages in the fastqueue, no preemption takes place.

P **PREEMPTION**. If a higher-priority thread is waiting to send a message to the fastqueue, it is resumed and preemption will occur.

SUSPENSION. If the fastqueue is empty, the calling thread is suspended until a message is sent to the fastqueue.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

px5_pthread_fastqueue_*, px5_pthread_fastqueue_send

Small Example:

```
#include <pthread.h>

/* Fastqueue handle.  */
pthread_fastqueue_t      my_fastqueue_handle;
u_long                   my_message;
int                      status;


    /* Receive message from the fastqueue "my_fastqueue_handle". */
    status = px5_pthread_fastqueue_receive(&my_fastqueue_handle,
                                                 &my_message, 1);

    /* If status is PX5_SUCCESS, "my_message" contains the message
       received. */
```

px5_pthread_fastqueue_send

C Prototype:

```
#include <pthread.h>

int px5_pthread_fastqueue_send(pthread_fastqueue_t *
         fastqueue_handle, u_long *  message, size_t message_size);
```

Description:

This pthreads+ service sends the specified message to the specified fastqueue. If there is a thread waiting on the fastqueue, this message is delivered to the thread and it is resumed. If the fastqueue is full, this thread suspends until there is room in the fastqueue.

API Parameters:

`fastqueue_handle`	Handle of the fastqueue.
`message`	Pointer to the start of the message.
`message_size`	Size of the message in bytes (must be evenly divisible by sizeof(u_long)).

Return Codes:

PX5_SUCCESS (0)	Successful fastqueue send.
EINVAL	Invalid fastqueue handle pointer.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. If there are no other threads waiting for a message on the fastqueue, no preemption takes place.

PREEMPTION. If a higher-priority thread is the first waiting for a message, it is resumed and preemption will occur.

SUSPENSION. If the fastqueue is full, the calling thread is suspended until there is room in the fastqueue.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

px5_pthread_fastqueue_*, px5_pthread_fastqueue_receive

Small Example:

```
#include <pthread.h>

/* Fastqueue handle.  */
pthread_fastqueue_t      my_fastqueue_handle;
u_long                   my_message;
int                      status;


    /* Set the message to 0x12345678. */
    my_message =  0x12345678;

    /* Send the message to the fastqueue "my_fastqueue_handle". */
    status = px5_pthread_fastqueue_send(&my_fastqueue_handle,
                                                  &my_message, 1);

    /* If status is PX5_SUCCESS, "my_message" was sent. */
```

px5_pthread_fastqueue_tryreceive

C Prototype:

```
#include <pthread.h>

int px5_pthread_fastqueue_tryreceive(pthread_fastqueue_t *
      fastqueue_handle, u_long *  message, size_t message_size);
```

Description:

This pthreads+ service attempts to receive a message from the specified fastqueue. If the fastqueue is empty, this service returns an error.

API Parameters:

`fastqueue_handle`	Handle of the fastqueue.
`message`	Pointer to the destination for the message.
`message_size`	Size of the message in bytes (must be evenly divisible by sizeof(u_long)).

Return Codes:

PX5_SUCCESS (0)	Successful fastqueue receive.
EINVAL	Invalid fastqueue handle pointer.
EAGAIN	Message is not available (fastqueue is empty).

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. If there are one or more messages in the fastqueue, no preemption takes place.

P **PREEMPTION**. If a higher-priority thread is waiting to send a message to the fastqueue, it is resumed and preemption will occur.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

px5_pthread_fastqueue_*, px5_pthread_fastqueue_receive

Small Example:

```
#include <pthread.h>

/* Fastqueue handle.  */
pthread_fastqueue_t      my_fastqueue_handle;
u_long                   my_message;
int                      status;


    /* Try to receive message from the fastqueue
       “my_fastqueue_handle”. */
    status = px5_pthread_fastqueue_tryreceive(&my_fastqueue_handle,
                                                       &my_message, 1);

    /* If status is PX5_SUCCESS, “my_message” contains the message
        received. */
```

px5_pthread_fastqueue_trysend

C Prototype:

```
#include <pthread.h>

int px5_pthread_fastqueue_trysend(pthread_fastqueue_t *
               fastqueue_handle, u_long *  message, size_t
message_size);
```

Description:

This pthreads+ service tries to sends the specified message to the specified fastqueue. If there is a thread waiting on the fastqueue, this message is delivered to the thread and it is resumed. If the fastqueue is full, an error is returned.

API Parameters:

`fastqueue_handle`	Handle of the fastqueue.
`message`	Pointer to the start of the message.
`message_size`	Size of the message in bytes (must be evenly divisible by sizeof(u_long)).

Return Codes:

PX5_SUCCESS (0)	Successful fastqueue send.
EINVAL	Invalid fastqueue handle pointer.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. If there are no other threads waiting for a message on the fastqueue, no preemption takes place.

PREEMPTION. If a higher-priority thread is the first waiting for a message, it is resumed and preemption will occur.

Callable From:

This service is callable from the thread context and from interrupt handlers (ISRs).

See Also:

px5_pthread_fastqueue_*, px5_pthread_fastqueue_send

Small Example:

```
#include <pthread.h>

/* Fastqueue handle.  */
pthread_fastqueue_t      my_fastqueue_handle;
u_long                   my_message;
int                      status;


    /* Set the message to 0x12345678. */
    my_message =  0x12345678;

    /* Try to send the message to the fastqueue
       "my_fastqueue_handle". */
    status = px5_pthread_fastqueue_trysend(&my_fastqueue_handle,
                                                 &my_message, 1);

    /* If status is PX5_SUCCESS, "my_message" was sent. */
```

px5_pthread_fastqueueattr_destroy

C Prototype:

```
#include <pthread.h>

int px5_pthread_fastqueueattr_destroy(pthread_fastqueueattr_t *
                                      fastqueue_attributes);
```

Description:

This pthreads+ service destroys the previously created fastqueue attributes structure pointed to by *fastqueue_attributes*. Once destroyed, the fastqueue attributes structure cannot be used again unless it is recreated.

API Parameters:

`fastqueue_attributes`	Pointer to the fastqueue attributes to destroy.

Return Codes:

PX5_SUCCESS (0)	Successful fastqueue attributes destroy.
EINVAL	Invalid fastqueue attributes pointer.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

px5_pthread_fastqueue_*, px5_pthread_fastqueueattr_*, px5_pthread_fastqueueattr_init

Small Example:

```
#include <pthread.h>

/* Fastqueue attribute structure.  */
pthread_fastqueueattr_t      my_fastqueue_attributes;
int                          status;


    /* Destroy the fastqueue attributes referenced by
       "my_fastqueue_attributes". */
    status =  px5_pthread_fastqueueattr_destroy(
                                 &my_fastqueue_attributes);

    /* If status is PX5_SUCCESS, the fastqueue attributes structure
       was destroyed.  */
```

px5_pthread_fastqueueattr_getcontroladdr

C Prototype:

```
#include <pthread.h>

int px5_pthread_fastqueueattr_getcontroladdr(pthread_fastqueueattr_t
       * fastqueue_attributes, void **  fastqueue_control_address);
```

Description:

This pthreads+ service returns the previously supplied fastqueue control structure memory address.

API Parameters:

`fastqueue_attributes`	Pointer to the fastqueue attributes.
`fastqueue_control_address`	Pointer to the destination for the previously supplied fastqueue control address.

Return Codes:

PX5_SUCCESS (0)	Successful fastqueue attributes fastqueue control address retrieval.
EINVAL	Invalid fastqueue attributes, or fastqueue control address designation pointer.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

px5_pthread_fastqueue_*, px5_pthread_fastqueueattr_*, px5_pthread_fastqueueattr_setcontroladdr, px5_pthread_fastqueueattr_getcontrolsize

Small Example:

```
#include <pthread.h>

/* Fastqueue attribute structure.  */
pthread_fastqueueattr_t    my_fastqueue_attributes;
void *                     my_fastqueue_control_address;
int                        status;




    /* Get the fastqueue control structure address in the
       attributes structure "my_fastqueue_attributes". */
    status = px5_pthread_fastqueueattr_getcontroladdr(
                    &my_fastqueue_attributes,
                    &my_fastqueue_control_address);

    /* If status is PX5_SUCCESS, "my_fastqueue_control_address"
        contains the address of the previously supplied fastqueue
        control memory.  */
```

px5_pthread_fastqueueattr_getcontrolsize

C Prototype:

```
#include <pthread.h>

int px5_pthread_fastqueueattr_getcontrolsize(pthread_fastqueueattr_t
       * fastqueue_attributes, size_t *  fastqueue_control_size);
```

Description:

This pthreads+ service returns the size of the internal fastqueue control structure. The main purpose of this API is to inform the application how much memory is required for the *px5_pthread_fastqueueattr_setcontroladdr* API.

API Parameters:

`fastqueue_attributes`	Pointer to the fastqueue attributes.
`fastqueue_control_size`	Pointer to the destination for the internal fastqueue control structure size.

Return Codes:

PX5_SUCCESS (0)	Successful retrieval of internal fastqueue control structure size.
EINVAL	Invalid fastqueue attributes or fastqueue control structure size destination pointer.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NO PREEMPTION. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

px5_pthread_fastqueue_*, px5_pthread_fastqueueattr_*, px5_pthread_fastqueueattr_setcontroladdr

Small Example:

```
#include <pthread.h>

/* Fastqueue attribute structure.  */
pthread_fastqueueattr_t    my_fastqueue_attributes;
size_t                     my_fastqueue_control_size;
int                        status;


    /* Get the fastqueue control structure memory size. */
    status = px5_pthread_fastqueueattr_getcontrolsize(
       &my_fastqueue_attributes, &my_fastqueue_control_size);

    /* If status is PX5_SUCCESS, "my_fastqueue_control_size"
        contains the size of the internal fastqueue
        control structure.  */
```

px5_pthread_fastqueueattr_getname

C Prototype:

```
#include <pthread.h>

int px5_pthread_fastqueueattr_getname(pthread_fastqueueattr_t
              * fastqueue_attributes, char **  fastqueue_name);
```

Description:

This pthreads+ service returns the previously supplied fastqueue name.

API Parameters:

`fastqueue_attributes`	Pointer to the fastqueue attributes.
`fastqueue_name`	Pointer to the destination for the previous fastqueue name.

Return Codes:

PX5_SUCCESS (0)	Successful fastqueue attributes fastqueue name retrieval.
EINVAL	Invalid fastqueue attributes or fastqueue name pointer.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

*px5_pthread_fastqueue_*, px5_pthread_fastqueueattr_**

Small Example:

```
#include <pthread.h>

/* Fastqueue attribute structure.  */
pthread_fastqueueattr_t    my_fastqueue_attributes;
char *                     my_fastqueue_name;
int                        status;



    /* Get the previous fastqueue name. */
    status =  px5_pthread_fastqueueattr_getname(
                    &my_fastqueue_attributes, &my_fastqueue_name);

    /* If status is PX5_SUCCESS, "my_fastqueue_name" contains the
        name previously supplied. */
```

px5_pthread_fastqueueattr_getqueueaddr

C Prototype:

```
#include <pthead.h>

int px5_pthread_fastqueueattr_getqueueaddr(pthread_fastqueueattr_t*
                    fastqueue_attributes,
                    void **  fastqueue_memory_address);
```

Description:

This pthreads+ service returns the previously supplied fastqueue memory area address.

API Parameters:

`fastqueue_attributes`	Pointer to the fastqueue attributes.
`fastqueue_memory_address`	Pointer to the destination for the previously supplied fastqueue memory area address.

Return Codes:

PX5_SUCCESS (0)	Successful fastqueue memory area address retrieval.
EINVAL	Invalid fastqueue attributes or fastqueue memory area address designation pointer.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NO PREEMPTION. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

*px5_pthread_fastqueue_*, px5_pthread_fastqueueattr_**

Small Example:

```
#include <mqueue.h>

/* Fastqueue queue attribute structure.  */
pthread_fastqueueattr_t my_fastqueue_attributes;
void *                  my_fastqueue_memory_address;
int                     status;




    /* Get the fastqueue memory area address in the
       fastqueue attributes structure
       "my_fastqueue_attributes". */
    status = px5_pthread_fastqueueattr_getqueueaddr(
                                  &my_fastqueue_attributes,
                                  &my_queue_memory_address);

    /* If status is PX5_SUCCESS, "my_fastqueue_memory_address"
        contains the address of the previously supplied fastqueue
        memory.  */
```

px5_pthread_fastqueueattr_getqueuesize

C Prototype:

```
#include <pthread.h>

int px5_pthread_fastqueueattr_getqueuesize(pthread_fastqueueattr_t*
             fastqeueu_attributes, size_t *  fastqueue_memory_size);
```

Description:

This pthreads+ service returns the size of the previously supplied fastqueue memory area.

API Parameters:

`fastqueue_attributes`	Pointer to the attributes.
`fastqueue_memory_size`	Pointer to the destination for the previously supplied fastqueue memory area size.

Return Codes:

PX5_SUCCESS (0)	Successful retrieval of fastqueue memory area size.
EINVAL	Invalid extended fastqueue attributes or invalid destination for fastqueue memory area size.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NO PREEMPTION. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

*px5_pthread_fastqueue_*, px5_pthread_fastqueueattr_**

Small Example:

```
#include <pthread.h>

/* Fastqueue extended attribute structure.  */
pthread_fastqueueattr_t  my_fastqueue_attributes;
size_t                   my_fastqueue_memory_size;
int                      status;



    /* Get the fastqueue memory area size. */
    status =  px5_pthread_fastqueueattr_getqueuesize(
              &my_fastqueue_attributes, &my_fastqueue_memory_size);

    /* If status is PX5_SUCCESS, "my_fastqueue_memory_size"
       contains the size of the fastqueue memory area.  */
```

px5_pthread_fastqueueattr_init

C Prototype:

```
#include <pthread.h>

int px5_pthread_fastqueueattr_init(pthread_fastqueueattr_t *
                                    fastqueue_attributes);
```

Description:

This pthreads+ service initializes the fastqueue attributes structure with default fastqueue creation values.

API Parameters:

`fastqueue_attributes`	Pointer to the fastqueue attributes structure to create.

Return Codes:

PX5_SUCCESS (0)	Successful fastqueue attributes structure creation.
EINVAL	Invalid fastqueue attributes pointer.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

px5_pthread_fastqueue_*, px5_pthread_fastqueueattr_*, px5_pthread_fastqueueattr_destroy

Small Example:

```
#include <pthread.h>

/* Fastqueue attribute structure.  */
pthread_fastqueueattr_t    my_fastqueue_attributes;
int                        status;




    /* Create the fastqueue attributes structure
       "my_fastqueue_attributes". */
    status = px5_pthread_fastqueueattr_init(&my_fastqueue_attributes);

    /* If status is PX5_SUCCESS, the "my_fastqueue_attributes"
       structure is ready for use.  */
```

px5_pthread_fastqueueattr_setcontroladdr

C Prototype:

```
#include <pthread.h>

int px5_pthread_fastqueueattr_setcontroladdr(pthread_fastqueueattr_t
       * fastqueue_attributes, void *  fastqueue_control_address,
                                     size_t fastqueue_control_size);
```

Description:

This pthreads+ service provides a mechanism for the user to provide the memory for the internal PX5 RTOS fastqueue structure, as specified by the address contained in the *fastqueue_control_address* parameter. This memory will subsequently be used for the fastqueue created with this attribute structure. The size of the memory required for the internal fastqueue control structure can be found via a call to the *px5_pthread_fastqueueattr_getcontrolsize* service.

Note that each fastqueue created must have its own unique fastqueue control structure memory. Hence, the fastqueue control address supplied here is only valid for one px5_pthread_fastqueue_create call.

API Parameters:

`fastqueue_attributes`	Pointer to the fastqueue attributes.
`fastqueue_control_address`	Pointer to the supplied fastqueue control structure memory.
`fastqueue_control_size`	Size of memory specified.

Return Codes:

PX5_SUCCESS (0)	Successful internal fastqueue control address set.

EINVAL	Invalid fastqueue attributes or memory size.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

px5_pthread_fastqueue_*, px5_pthread_fastqueueattr_*, px5_pthread_fastqueueattr_getcontrolsize

Small Example:

```
#include <pthread.h>

/* Fastqueue attribute structure.  */
pthread_fastqueueattr_t    my_fastqueue_attributes;
int                        status;



    /* Provide the memory for the fastqueue control structure in the
        next fastqueue create call by placing it's address in the
        attributes structure "my_fastqueue_attributes". */
    status =  px5_pthread_fastqueueattr_setcontroladdr(
              &my_fastqueue_attributes, 0x90000, 60);

    /* If status is PX5_SUCCESS, the next fastqueue creation using
       these attributes will use memory at address 0x90000 for the
       internal fastqueue control structure. */
```

px5_pthread_fastqueueattr_setname

C Prototype:

```
#include <pthread.h>

int px5_pthread_fastqueueattr_setname(pthread_fastqueueattr_t
             * fastqueue_attributes, char *  fastqueue_name);
```

Description:

This pthreads+ service sets the fastqueue name in the specified attribute structure.

API Parameters:

`fastqueue_attributes`	Pointer to the fastqueue attributes.
`fastqueue_name`	Pointer to the supplied fastqueue name.

Return Codes:

PX5_SUCCESS (0)	Successful fastqueue attributes fastqueue name set.
EINVAL	Invalid fastqueue attributes.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

px5_pthread_fastqueue_*, px5_pthread_fastqueueattr_*, px5_pthread_fastqueueattr_getname

Small Example:

```
#include <pthread.h>

/* Fastqueue attribute structure.  */
pthread_fastqueueattr_t    my_fastqueue_attributes;
int                        status;


    /* Set the fastqueue name in the fastqueue attributes structure
       "my_fastqueue_attributes". */
    status =  px5_pthread_fastqueueattr_setname(
                    &my_fastqueue_attributes, "my_fastqueue_name");


    /* If status is PX5_SUCCESS, "my_fastqueue_name" is set in the
       fastqueue attribute structure.  */
```

px5_pthread_fastqueueattr_setqueueaddr

C Prototype:

```
#include <pthread.h>

int px5_pthread_fastqueueattr_setqueueaddr(pthread_fastqueueattr_t*
                                                fastqueue_attributes,
                                    void *  fastqueue_memory_address,
                                    size_t  fastqueue_memory_size);
```

Description:

This pthreads+ service sets the internal fastqueue message memory address to the address specified by *fastqueue_memory_address*. This address will subsequently be used to supply the memory for the message area on the next fastqueue created with this attribute structure.

Note that each fastqueue created must have its own unique fastqueue memory area. Hence, the memory address supplied here is only valid for one px5_pthread_fastqueue_create call.

API Parameters:

`fastqueue_attributes`	Pointer to the fastqueue attributes.
`fastqueue_memory_address`	Pointer to the fastqueue message memory area address.
`fastqueue_memory_size`	Size of specified fastqueue memory area.

Return Codes:

PX5_SUCCESS (0)	Successful specification of fastqueue memory.
EINVAL	Invalid fastqueue attributes or invalid size of fastqueue message memory area.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NO PREEMPTION. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

px5_pthread_fastqueue_*, px5_pthread_fastqueueattr_*,

Small Example:

```
#include <pthread.h>

/* Fastqueue attribute structure.  */
pthread_fastqueueattr_t  my_fastqueue_attributes;
int                      status;


    /* Set the fastqueue message memory area address in the
       fastqueue attributes structure "my_fastqueue_attributes". */
    status = px5_pthread_fastqueueattr_setqueueaddr(
                           &my_fastqueue_attributes, 0x90000, 1024);
```

px5_pthread_information_get

C Prototype:

```
#include <pthread.h>

int  px5_pthread_information_get(pthread_t thread_handle,
          char **name, int *  state, int *  priority,
          void ** stack_limit, void ** stack_pointer,
          u_long *  minimum_stack, pthread_t *  next_thread);
```

Description:

This pthreads+ service retrieves the specified information from the specified thread.

API Parameters:

`thread_handle`	Handle of thread to get information about.
`name`	If non-NULL, destination for the specified thread's name.
`state`	If non-NULL, destination for the specified thread's current state. A value of zero indicates the thread is currently ready for execution. All other values indicate the thread is blocked or has finished, or been canceled.
`priority`	If non-NULL, destination for the specified thread's priority.
`stack_limit`	If non-NULL, destination for the specified thread's stack limit.
`stack_pointer`	If non-NULL, destination for the specified thread's current stack pointer.
`minimum_stack`	If non-NULL, destination for the specified thread's minimum stack value.
`next_thread`	If non-NULL, destination for the specified next thread handle. This API can be called successively to examine all threads by using this parameter until the original thread handle is found.

Return Codes:

PX5_SUCCESS (0)	Successful information retrieved from the specified thread.
ESRCH	Invalid thread handle.

Real-time Scenarios:

Upon the successful execution of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

pthread_attr_*, pthread_create

Small Example:

```
#include <pthread.h>

pthread_t          my_thread_handle;
char *             name;
int                state;
int                priority;
void *             stack_limit;
void *             stack_pointer;
u_long             minimum_stack;
pthread_t          next_thread;
int                status;


    /* Get information about "my_thread_handle".  */
    status =  px5_pthread_information_get(&my_thread_handle,
              &name, &state, &priority, &stack_limit, &stack_pointer,
              &minimum_stack, &next_thread);

    /* If status contains PX5_SUCCESS, the information about
       "my_thread_handle" is available for use. */
```

px5_pthread_memory_manager_enable

C Prototype:

```
#include <pthread.h>

int  px5_pthread_memory_manager_enable(void);
```

Description:

This pthreads+ service enables full management of the remaining amount of memory supplied by the application when the PX5 RTOS is started via *px5_pthread_start*. The memory allocated with all subsequent system object creation and destruction will be fully managed.

This API creates a PX5 RTOS variable-length memory pool in order to manage the memory. This memory pool should not be released by the application.

API Parameters:

none

Return Codes:

PX5_SUCCESS (0)	Successful enablement of internal PX5 RTOS memory management.
EINVAL	Invalid (NULL) pointer supplied to the *px5_pthread_start* API.
ENOME	Not enough memory remaining.

Real-time Scenarios:

Upon the successful execution of this service, the following real-time scenarios are possible:

NO PREEMPTION. This service does not result in preemption.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

px5_pthread_memory_manager_get, px5_pthread_memory_manager_set, px5_pthread_start

Small Example:

```
#include <pthread.h>



/* Enable full management of PX5 RTOS memory.  */
status =  px5_pthread_memory_manager_enable();



/* If status is PX5_SUCCESS, all default memory
      allocation/deallocations are managed by the PX5 RTOS. */
```

px5_pthread_memory_manager_get

C Prototype:

```
#include <pthread.h>

int  px5_pthread_memory_manager_get(
      void *(** memory_allocate)(u_int type, u_long size),
      void  (** memory_release)(u_int type, void *memory_to_release));
```

Description:

This pthreads+ service retrieves the internal memory manager allocate and release function pointers. These function pointers can be used to restore the previous memory manager allocation and deallocation selection.

By default, the simple sequential memory allocation manager is setup when PX5 begins operation via px5_pthread_start. This memory manager does not support releasing memory. Hence, the memory_release function pointer is NULL.

API Parameters:

`memory_allocate`	Destination for current memory allocation function pointer.
`memory_release`	Destination for current memory release function pointer.

Return Codes:

PX5_SUCCESS (0)	Successful retrieval of memory manager function pointers.
EINVAL	Invalid (NULL) destination pointer(s) supplied.

Real-time Scenarios:

Upon the successful execution of this service, the following real-time scenarios are possible:

NO PREEMPTION. This service does not result in preemption.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

px5_pthread_memory_manager_set, px5_pthread_start

Small Example:

```
#include <pthread.h>

void *(* my_memory_allocate)(u_int type, u_long size);
void  (* my_memory_release)(u_int type, void *memory_to_release);
int   status;


/* Retrieve the current memory manager function pointers.  */
status =  px5_pthread_memory_manager_get(&my_memory_allocate,
                                              &my_memory_release);



/* If status is PX5_SUCCESS, the "my_memory_allocate" and
   "my_memory_release" contain the current default memory manager
   allocate and release function pointers. */
```

px5_pthread_memory_manager_set

C Prototype:

```
#include <pthread.h>

int  px5_pthread_memory_manager_set(
      void *(* memory_allocate)(u_int type, u_long size),
      void  (* memory_release)(u_int type, void *memory_to_release));
```

Description:

This pthreads+ service installs the specified memory manager allocate and release function pointers.

API Parameters:

`memory_allocate`	Memory manager's allocation function pointer.
`memory_release`	Memory manager's memory release function pointer.

Return Codes:

PX5_SUCCESS (0)	Successful setup of memory manager function pointers.
EINVAL	Invalid (NULL) allocation function pointer supplied.

Real-time Scenarios:

Upon the successful execution of this service, the following real-time scenarios are possible:

NO PREEMPTION. This service does not result in preemption.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

px5_pthread_memory_manager_get, px5_pthread_start

Small Example:

```
#include <pthread.h>

void *my_memory_allocate(u_int type, u_long size)
{

void *memory;

    /* Assuming thread-safe malloc, simply malloc the memory. If not,
       the malloc and free must be protected by mutex.  */
    memory =  malloc(size);

    /* Return the memory. */
    return(memory);
}

void  my_memory_release(u_int type, void *memory_to_release)
{

    /* Release the memory.  */
    free(memory);
}


/* Setup the new memory manager allocate and release memory
   routines.  */
status =  px5_pthread_memory_manager_set(my_memory_allocate,
                                               my_memory_release);



/* If status is PX5_SUCCESS, the "my_memory_allocate" and
   "my_memory_release" memory manager routines will be used for default
   internal memory allocation and deallocation.  */
```

px5_pthread_memorypool_allocate

C Prototype:

```
#include <pthread.h>

int px5_pthread_memorypool_allocate(pthread_memorypool_t *
                    memorypool_handle, void ** allocated_memory,
                    size_t request_size);
```

Description:

This pthreads+ service attempts to allocate memory from the specified memory pool. If there is not enough memory in the pool, the calling thread is suspended until the request can be satisfied.

The memory in the memory pool is managed as a linked-list. The number of fragments in this linked-list is unknown and can vary over time. Hence, the processing time for allocation is undeterministic and therefore this API should not be used in hard real-time situations.

This API is a cancellation point, meaning that if a cancellation is pending, it will be detected and executed by this API.

API Parameters:

`memorypool_handle`	Handle of the memory pool.
`allocated_memory`	Destination pointer to return the allocated memory.
`request_size`	Number of bytes requested.

Return Codes:

PX5_SUCCESS (0)	Successful memory allocation.
EINVAL	Invalid memory pool handle pointer, allocated memory destination pointer, or reqested size.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. If the requested memory is available, no preemption takes place.

S **SUSPENSION**. If the requested memory is not available, the calling thread is suspended until the memory becomes available.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

px5_pthread_memorypool_*, px5_pthread_memorypool_free

Small Example:

```
#include <pthread.h>

/* Memory pool handle and pointer for allocated memory.  */
pthread_memorypool_t     my_memorypool_handle;
void *                   allocated_memory;
int                      status;



    /* Attempt to allocate 100 bytes from "my_memorypool_handle." */
    status =  px5_pthread_memorypool_allocate(&my_memorypool_handle,
                    &allocated_memory, 100);

    /* If status is PX5_SUCCESS, a pointer to the allocated memory is
       in "allocated_memory."  */
```

px5_pthread_memorypool_create

C Prototype:

```
#include <pthread.h>

int px5_pthread_memorypool_create(pthread_memorypool_t *
       memorypool_handle,
       pthread_memorypoolattr_t *  memorypool_attributes,
        void *pool_start, size_t  pool_size);
```

Description:

This pthreads+ service creates a variable-length memory pool with the optional memory pool attributes. If successful, the memory pool handle is setup for further use by the application.

API Parameters:

`memorypool_handle`	Handle of the memory pool.
`memorypool_attributes`	Optional memory pool attributes. This value is NULL if no memory pool attributes are specified.
`pool_start`	Staring address of the memory area for the pool. This address must be at least 4-byte aligned.
`pool_size`	Numbe of byte in the memory area for the pool.

Return Codes:

PX5_SUCCESS (0)	Successful memory pool creation.
EINVAL	Invalid memor pool handle or pool start.
EBUSY	Memory pool is already created.
ENOMEM	Insufficient memory to the memory pool.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NO PREEMPTION. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

px5_pthread_memorypoolattr_*, px5_pthread_memorypool_destroy

Small Example:

```
#include <pthread.h>

/* Memory pool handle.  */
pthread_memorypool_t     my_memorypool_handle;
int                      status;


    /* Create a memory pool, the handle of which is returned in
       "my_memorypool_handle". */
    status =  px5_pthread_memorypool_create(&my_memorypool_handle,
                                     NULL, 0x40000, 8196);

    /* If status is PX5_SUCCESS, the memory pool was created at address
       0x40000 with a size of 8,196 bytes. */
```

px5_pthread_memorypool_destroy

C Prototype:

```
#include <pthread.h>

int px5_pthread_memorypool_destroy(pthread_memorypool_t *
                                         memorypool_handle);
```

Description:

This pthreads+ service destroys the previously created memory pool specified by *memorypool_handle*. If the memory pool has any suspended threads waiting for memory, an error is returned.

API Parameters:

`memorypool_handle`	Handle of the memory pool to destroy.

Return Codes:

PX5_SUCCESS (0)	Successful memory pool destruction.
EINVAL	Invalid memory pool handle.
EBUSY	A thread currently is suspended on the memory pool.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

px5_pthread_memorypool_*, px5_pthread_memorypool_create

Small Example:

```
#include <pthread.h>

/* Memory pool handle.  */
pthread_memorypool_t       my_memorypool_handle;
int                        status;




    /* Destroy the memory pool referenced by
       "my_memorypool_handle". */
    status =  px5_pthread_memorypool_destroy(&my_memorypool_handle);

    /* If status is PX5_SUCCESS, the memory pool was destroyed.  */
```

px5_pthread_memorypool_free

C Prototype:

```
#include <pthread.h>

int px5_pthread_memorypool_free(void * memory_to_release);
```

Description:

This pthreads+ service releases the previously allocated memory back to the memory pool it was allocated from. Any threads that are suspended on the memory pool that are waiting for this memory are resumend.

API Parameters:

`memory_to_release`	Pointer to previously allocated memory.

Return Codes:

PX5_SUCCESS (0)	Successful memory release.
EINVAL	Invalid memory pointer.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. If there are no other threads waiting for memory from the associated memory pool, no preemption takes place.

P **PREEMPTION**. If a higher-priority thread was waiting for this amount of memory, it is resumed, and preemption will occur.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

px5_pthread_memorypool_*, px5_pthread_memorypool_allocate

Small Example:

```
#include <pthread.h>

/* Allocated memory pointer.  */
void *                   my_memory_pointer;
int                      staus;


    /* Release the memory pointed to by “my_memory_pointer.” */
    status = px5_pthread_memorypool_free(my_memory_pointer);

    /* If status is PX5_SUCCESS, the memory was released. */
```

px5_pthread_memorypool_tryallocate

C Prototype:

```
#include <pthread.h>

int px5_pthread_memorypool_tryallocate(pthread_memorypool_t *
                    memorypool_handle, void ** allocated_memory,
                    size_t request_size);
```

Description:

This pthreads+ service attempts to allocate memory from the specified memory pool. If there is not enough memory in the pool, an error is returned.

The memory in the memory pool is managed as a linked-list. The number of fragments in this linked-list is unknown and can vary over time. Hence, the processing time for allocation is undeterministic and therefore this API should not be used in hard real-time situations.

API Parameters:

`memorypool_handle`	Handle of the memory pool.
`allocated_memory`	Destination pointer to return the allocated memory.
`request_size`	Number of bytes requested.

Return Codes:

PX5_SUCCESS (0)	Successful memory allocation.
EINVAL	Invalid memory pool handle pointer, allocated memory destination pointer, or reqested size.
ENOMEM	Not enough free memory to satisfy request.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NO PREEMPTION. No preemption takes place with this API.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

px5_pthread_memorypool_*, px5_pthread_memorypool_allocate

Small Example:

```
#include <pthread.h>

/* Memory pool handle and pointer for allocated memory.   */
pthread_memorypool_t       my_memorypool_handle;
void *                     allocated_memory;
int                        status;



    /* Attempt to allocate 100 bytes from "my_memorypool_handle." */
    status =  px5_pthread_memorypool_tryallocate(&my_memorypool_handle,
                    &allocated_memory, 100);

    /* If status is PX5_SUCCESS, a pointer to the allocated memory is
       in "allocated_memory."  */
```

px5_pthread_memorypoolattr_destroy

C Prototype:

```
#include <pthread.h>

int px5_pthread_memorypoolattr_destroy(pthread_memorypoolattr_t *
                                       memorypool_attributes);
```

Description:

This pthreads+ service destroys the previously created memory pool attributes structure pointed to by *memorypool_attributes*. Once destroyed, the memory pool attributes structure cannot be used again unless it is recreated.

API Parameters:

`memorypool_attributes`	Pointer to the memory pool attributes to destroy.

Return Codes:

PX5_SUCCESS (0)	Successful memory pool attributes destroy.
EINVAL	Invalid memory pool attributes pointer.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

px5_pthread_memorypool_*, px5_pthread_memorypoolattr_*, px5_pthread_memorypoolattr_init

Small Example:

```
#include <pthread.h>

/* Memory pool attribute structure.  */
pthread_memorypoolattr_t     my_memorypool_attributes;
int                          status;


    /* Destroy the memory pool attributes referenced by
       "my_memorypool_attributes". */
    status =  px5_pthread_memorypoolattr_destroy(
                                  &my_memorypool_attributes);

    /* If status is PX5_SUCCESS, the memory pool attributes structure
       was destroyed.  */
```

px5_pthread_memorypoolattr_getcontroladdr

C Prototype:

```
#include <pthread.h>

int px5_pthread_memorypoolattr_getcontroladdr(pthread_memorypoolattr_t
       * memorypool_attributes, void **  memorypool_control_address);
```

Description:

This pthreads+ service returns the previously supplied memory pool control structure memory address.

API Parameters:

`memorypool_attributes`	Pointer to the memory pool attributes.
`memorypool_control_address`	Pointer to the destination for the previously supplied memory pool control address.

Return Codes:

PX5_SUCCESS (0)	Successful memory pool attributes control address retrieval.
EINVAL	Invalid memory pool attributes, or memory pool control address designation pointer.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

px5_pthread_memorypool_*, px5_pthread_memorypoolattr_*, px5_pthread_memorypoolattr_setcontroladdr, px5_pthread_memorypoolattr_getcontrolsize

Small Example:

```
#include <pthread.h>

/* Memroy pool attribute structure.  */
pthread_memorypoolattr_t   my_memorypool_attributes;
void *                     my_memorypool_control_address;
int                        status;



    /* Get the memory pool control structure address in the
       attributes structure "my_memorypool_attributes". */
    status = px5_pthread_memorypoolattr_getcontroladdr(
                    &my_memorypool_attributes,
                    &my_memorypool_control_address);

    /* If status is PX5_SUCCESS, "my_memorypool_control_address"
        contains the address of the previously supplied event flags
        control memory.  */
```

px5_pthread_memorypoolattr_getcontrolsize

C Prototype:

```
#include <pthread.h>

int px5_pthread_memorypoolattr_getcontrolsize(pthread_memorypoolattr_t
       * memorypool_attributes, size_t *  memorypool_control_size);
```

Description:

This pthreads+ service returns the size of the internal memory pool control structure. The main purpose of this API is to inform the application how much memory is required for the *px5_pthread_memorypoolattr_setcontroladdr* API.

API Parameters:

`memorypool_attributes`	Pointer to the memory pool attributes.
`memorypool_control_size`	Pointer to the destination for the internal memory pool control structure size.

Return Codes:

PX5_SUCCESS (0)	Successful retrieval of internal memory pool control structure size.
EINVAL	Invalid memory pool attributes or memory pool control structure size destination pointer.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

px5_pthread_memorypool_*, px5_pthread_memorypoolattr_*, px5_pthread_memorypoolattr_setcontroladdr

Small Example:

```
#include <pthread.h>

/* Memory pool attribute structure.  */
pthread_memorypoolattr_t   my_memorypool_attributes;
size_t                     my_memorypool_control_size;
int                        status;


    /* Get the memory pool control structure memory size. */
    status = px5_pthread_memorypoolattr_getcontrolsize(
       &my_memorypool_attributes, &my_memorypool_control_size);

    /* If status is PX5_SUCCESS, "my_memorypool_control_size"
        contains the size of the internal memory pool
        control structure.  */
```

px5_pthread_memorypoolattr_getname

C Prototype:

```
#include <pthread.h>

int px5_pthread_memorypoolattr_getname(pthread_memorypoolattr_t
              * memorypool_attributes, char **  memorypool_name);
```

Description:

This pthreads+ service returns the previously supplied memory pool name.

API Parameters:

`memorypool_attributes`	Pointer to the memory pool attributes.
`memorypool_name`	Pointer to the destination for the previous memory pool name.

Return Codes:

PX5_SUCCESS (0)	Successful memory pool attributes name retrieval.
EINVAL	Invalid memory pool attributes or memory pool name pointer.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

*px5_pthread_memorypool_*, px5_pthread_memorypoolattr_**

Small Example:

```
#include <pthread.h>

/* Memory pool attribute structure.  */
pthread_memorypoolattr_t   my_memorypool_attributes;
char *                     my_memorypool_name;
int                        status;




    /* Get the previous memory pool name. */
    status =  px5_pthread_memorypoolattr_getname(
                    &my_memorypool_attributes, &my_memorypool_name);

    /* If status is PX5_SUCCESS, "my_memorypool_name" contains the
       name previously supplied. */
```

px5_pthread_memorypoolattr_init

C Prototype:

```
#include <pthread.h>

int px5_pthread_memorypoolattr_init(pthread_memorypoolattr_t *
                                    memorypool_attributes);
```

Description:

This pthreads+ service initializes the memory pool attributes structure with default memory pool creation values.

API Parameters:

`memorypool_attributes`	Pointer to the memory pool attributes structure to create.

Return Codes:

PX5_SUCCESS (0)	Successful memory pool attributes structure creation.
EINVAL	Invalid memory pool attributes pointer.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

px5_pthread_memorypool_*, px5_pthread_memorypoolattr_*, px5_pthread_memorypoolattr_destroy

Small Example:

```
#include <pthread.h>

/* Memory pool attribute structure.  */
pthread_memorypoolattr_t   my_memorypool_attributes;
int                        status;



    /* Create the memory pool attributes structure
       "my_memorypool_attributes". */
    status =
       px5_pthread_memorypoolattr_init(&my_memorypool_attributes);

    /* If status is PX5_SUCCESS, the "my_memorypool_attributes"
       structure is ready for use.  */
```

px5_pthread_memorypoolattr_setcontroladdr

C Prototype:

```
#include <pthread.h>

int px5_pthread_memorypoolattr_setcontroladdr(pthread_memorypoolattr_t
       * memorypool_attributes, void *  memorypool_control_address,
                                  size_t memorypool_control_size);
```

Description:

This pthreads+ service provides a mechanism for the user to provide the memory for the internal PX5 RTOS memory pool structure, as specified by the address contained in the *memorypool_control_address* parameter. This memory will subsequently be used for the next memory pool created with this attribute structure. The size of the memory required for the internal memory pool control structure can be found via a call to the *px5_pthread_memorypoolattr_getcontrolsize* service.

Note that each memory pool created must have its own unique control structure memory. Hence, the memory pool control address supplied here is only valid for one px5_pthread_memorypool_create call.

API Parameters:

`memorypool_attributes`	Pointer to the memory pool attributes.
`memorypool_control_address`	Pointer to the supplied memory pool control structure memory.
`memorypool_control_size`	Size of memory specified.

Return Codes:

PX5_SUCCESS (0)	Successful internal memory pool control address set.

EINVAL	Invalid memory pool attributes or memory size.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NO PREEMPTION. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

px5_pthread_memorypool_*, px5_pthread_memorypoolattr_*, px5_pthread_memorypoolattr_getcontrolsize

Small Example:

```
#include <pthread.h>

/* Memory pool attribute structure.  */
pthread_memorypoolattr_t   my_memorypool_attributes;
int                        status;


    /* Provide the memory for the memory pool control structure in the
        next memory pool create call by placing it’s address in the
        attributes structure “my_memorypool_attributes”. */
    status =  pthread_memorypoolattr_setcontroladdr(
              &my_memorypool_attributes, 0x80000, 150);

    /* If status is PX5_SUCCESS, the next memory pool creation using
       these attributes will use memory at address 0x80000 for the
       internal memory pool control structure. */
```

px5_pthread_memorypoolattr_setname

C Prototype:

```
#include <pthread.h>

int px5_pthread_memorypoolattr_setname(pthread_memorypoolattr_t
              * memorypool_attributes, char *  memorypool_name);
```

Description:

This pthreads+ service sets the memory pool name in the specified attribute structure.

API Parameters:

`memorypool_attributes`	Pointer to the memory pool attributes.
`memorypool_name`	Pointer to the supplied memory pool name.

Return Codes:

PX5_SUCCESS (0)	Successful memory pool attributes name set.
EINVAL	Invalid memory pool attributes.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

px5_pthread_memorypool_*, px5_pthread_memorypoolattr_*, px5_pthread_memorypoolattr_getname

Small Example:

```
#include <pthread.h>

/* Memory pool attribute structure.  */
pthread_memorypoolattr_t   my_memorypool_attributes;
int                        status;


    /* Set the memory pool name in the memory pool attributes structure
       "my_memorypool_attributes". */
    status =  px5_pthread_memorypoolattr_setname(
                    &my_memorypool_attributes, "my_memorypool_name");

    /* If status is PX5_SUCCESS, "my_memorypool_name" is set in the
       memory pool attribute structure.  */
```

px5_pthread_mutexattr_getcontroladdr

C Prototype:

```
#include <pthread.h>

int px5_pthread_mutexattr_getcontroladdr(pthread_mutexattr_t
              *mutex_attributes, void **  mutex_control_address);
```

Description:

This pthreads+ service returns the previously supplied mutex control structure address.

API Parameters:

`mutex_attributes`	Pointer to the mutex attributes.
`mutex_control_address`	Pointer to the destination for the previously supplied mutex control address.

Return Codes:

PX5_SUCCESS (0)	Successful mutex attributes mutex control address retrieval.
EINVAL	Invalid mutex attributes or mutex control address designation pointer.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

pthread_mutex_*, pthread_mutexattr_*, px5_pthread_mutexattr_setcontroladdr, px5_pthread_mutexattr_getcontrolsize

Small Example:

```
#include <pthread.h>

/* Mutex attribute structure.  */
pthread_mutexattr_t    my_mutex_attributes;
void *                 my_mutex_control_address;
int                    status;




    /* Get the mutex control structure address in the mutex
       attributes structure "my_mutex_attributes". */
    status = px5_pthread_mutexattr_getcontroladdr(&my_mutex_attributes,
                                  &my_mutex_control_address);

    /* If status is PX5_SUCCESS, "my_mutex_control_address"
        contains the address of the previously supplied mutex
        control memory.  */
```

px5_pthread_mutexattr_getcontrolsize

C Prototype:

```
#include <pthread.h>

int px5_pthread_mutexattr_getcontrolsize(pthread_mutexattr_t
              *mutex_attributes, size_t *  mutex_control_size);
```

Description:

This pthreads+ service returns the size of the internal mutex control structure. The main purpose of this API is to inform the application how much memory is required for the *px5_pthread_mutexattr_setcontroladdr* API.

API Parameters:

`mutex_attributes`	Pointer to the mutex attributes.
`mutex_control_size`	Pointer to the destination for the internal mutex control structure size.

Return Codes:

PX5_SUCCESS (0)	Successful retrieval of internal mutex control structure size.
EINVAL	Invalid mutex attributes or mutex control structure size destination pointer.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

pthread_mutex_*, pthread_mutexattr_*,
px5_pthread_mutexattr_setcontroladdr,
px5_pthread_mutexattr_getcontroladdr

Small Example:

```
#include <pthread.h>

/* Mutex attribute structure.  */
pthread_mutexattr_t    my_mutex_attributes;
size_t                 my_mutex_control_size;
int                    status;




    /* Get the mutex control structure memory size. */
    status = px5_pthread_mutexattr_getcontrolsize(&my_mutex_attributes,
                                   &my_mutex_control_size);

    /* If status is PX5_SUCCESS, "my_mutex_control_size"
       contains the size of the internal mutex
       control structure.  */
```

px5_pthread_mutexattr_getname

C Prototype:

```
#include <pthread.h>

int px5_pthread_mutexattr_getname(pthread_mutexattr_t
                                  *mutex_attributes, char **  mutex_name);
```

Description:

This pthreads+ service returns the previously supplied mutex name.

API Parameters:

`mutex_attributes`	Pointer to the mutex attributes.
`mutex_name`	Pointer to the destination for the previous mutex name.

Return Codes:

PX5_SUCCESS (0)	Successful mutex attributes mutex name retrieval.
EINVAL	Invalid mutex attributes or mutex name pointer.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

pthread_mutex_*, pthread_mutexattr_*, px5_pthread_mutexattr_setname

Small Example:

```
#include <pthread.h>

/* Mutex attribute structure.  */
pthread_mutexattr_t    my_mutex_attributes;
char *                 my_mutex_name;
int                    status;



    /* Get the previous mutex name. */
    status =  px5_pthread_mutexattr_getname(&my_mutex_attributes,
                                             &my_mutex_name);

    /* If status is PX5_SUCCESS, "my_mutex_name" contains the
       name previously supplied. */
```

px5_pthread_mutexattr_setcontroladdr

C Prototype:

```
#include <pthread.h>

int px5_pthread_mutexattr_setcontroladdr(pthread_mutexattr_t
                    *mutex_attributes, void *  mutex_control_address,
                                 size_t mutex_control_size);
```

Description:

This pthreads+ service provides a mechanism for the user to provide the memory for the internal PX5 RTOS mutex structure, as specified by the address contained in the *mutex_control_address* parameter. This memory will subsequently be used for the next mutex created with this attribute structure. The size of the memory required for the internal mutex control structure can be found via a call to the *px5_pthread_mutexattr_getcontrolsize* service.

Note that each mutex created must have its own unique mutex control structure memory. Hence, the mutex control memory supplied here is only valid for one pthread_mutex_init call.

API Parameters:

`mutex_attributes`	Pointer to the mutex attributes.
`mutex_control_address`	Pointer to the supplied mutex control structure memory.
`mutex_control_size`	Size of specified mutex control structure memory.

Return Codes:

PX5_SUCCESS (0)	Successful internal mutex control address set.

EINVAL	Invalid mutex attributes or internal mutex control memory size.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NO PREEMPTION. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

pthread_mutex_*, pthread_mutexattr_*, pthread_mutexattr_getcontroladdr, pthread_mutexattr_getcontrolsize

Small Example:

```
#include <pthread.h>

/* Mutex attribute structure.  */
pthread_mutexattr_t    my_mutex_attributes;
int                    status;



    /* Set the mutex control structure address in the mutex
       attributes structure "my_mutex_attributes". */
    status =  pthread_mutexattr_setcontroladdr(&my_mutex_attributes,
                                          0x50000, 80);

    /* If status is PX5_SUCCESS, the mutex creation using these
       attributes will use address 0x50000 for the internal mutex
       control structure. */
```

px5_pthread_mutexattr_setname

C Prototype:

```
#include <pthread.h>

int px5_pthread_mutexattr_setname(pthread_mutexattr_t
                                  *mutex_attributes, char *  mutex_name);
```

Description:

This pthreads+ service sets the mutex name in the specified attribute structure.

API Parameters:

`mutex_attributes`	Pointer to the mutex attributes.
`mutex_name`	Pointer to the supplied mutex name.

Return Codes:

PX5_SUCCESS (0)	Successful mutex attributes mutex name set.
EINVAL	Invalid mutex attributes.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

pthread_mutex_*, pthread_mutexattr_*, px5_pthread_mutexattr_getname

Small Example:

```
#include <pthread.h>

/* Mutex attribute structure.  */
pthread_mutexattr_t    my_mutex_attributes;
int                    status;




    /* Set the mutex name in the mutex attributes structure
       "my_mutex_attributes". */
    status =  px5_pthread_mutexattr_setname(&my_mutex_attributes,
                                            "my_mutex_name");

    /* If status is PX5_SUCCESS, "my_mutex_name" is set in the
       mutex attribute structure.  */
```

px5_pthread_resume

C Prototype:

```
#include <pthread.h>

int  px5_pthread_resume(pthread_t thread_handle);
```

Description:

This pthread+ service resumes a previously suspended thread. If the specified thread is not suspended, an error is returned.

API Parameters:

`thread_handle`	Handle of previously suspended thread.

Return Codes:

PX5_SUCCESS (0)	Successful thread resumption.
EINVAL	Thread handle is not valid.
EBUSY	Specified thread is not suspended.

Real-time Scenarios:

Upon the successful execution of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. If the thread resumed is not higher priority, there is no preemption.

P **PREEMPTION**. If a higher-priority thread is resumed, preemption will occur.

Callable From:

This service is callable from the thread context and from interrupt handlers (ISRs).

See Also:

px5_pthread_suspend

Small Example:

```
#include <pthread.h>

pthread_t           my_thread_handle;
int                 status;



    /* Resume the previously suspended thread.  */
    status =  px5_pthread_resume(my_thread_handle);

    /* If status contains PX5_SUCCESS, the thread is now resumed. */
```

px5_pthread_start

C Prototype:

```
#include <pthread.h>

int px5_pthread_start(u_long run_time_id, void *  memory_start,
                                          u_long  memory_size);
```

Description:

This pthreads+ service starts the PX5 RTOS. It should be called early in C *main*. Once called, all internal data structures are initialized and prepared for operation. The initialization processing also includes verification of the binding layer – the processor/compiler specific layer of PX5. If the start process is successful, the C *main* function is upscaled into the first thread in the system.

The optional *memory_start* and *memory_size* parameters provide the PX5 RTOS with memory to sequentially allocate from. For simple applications without object deletion, this memory allocation scheme is sufficient.

API Parameters:

`run_time_id`	This is a user-supplied run time identification of this particular execution instance. Ideally, the value supplied here is generated from a True Random Number Generator (TRNG) such that it is unique for each execution of an application via PX5. Note that this value is an integral component to the Pointer/Data Verification (PDV) of PX5.
`memory_start`	Optional pointer to the memory provided by the application for internal memory allocation by PX5. This pointer should be aligned to at least the word size of the underlying processor architecture. If the *px5_user_memory_manager_override* service is called by the application, the *memory_start* pointer may be NULL.

`memory_size`	Optional memory size supplied by the application.

Return Codes:

PX5_SUCCESS (0)	Successful PX5 start–C *main* is now a thread!
EMVSERR	System error–invalid processor binding.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. Even though C *main* is upscaled into the first thread, there is no actual preemption possible with this service.

Callable From:

This service is only callable once from the C *main* entry function.

See Also:

pthread_*, pthread_create

Small Example:

```
#include <pthread.h>

/* Define some memory for PX5.  */
unsigned long   memory_area[1024];

unsigned long   main_thread_counter;


int main(void)
{

int   status;


    /* Call the PX5 start function.  */
    status =  pthread_start(memory_area, sizeof(memory_area));
    /* Check completion status.  */
    if (status != PX5_SUCCESS)
    {
        printf(“Error starting PX5!\n”);
        exit(1);
    }

    /* When we return, "main" is now the first PX5 thread.
       Simply loop incrementing the counter.  */
    while(1)
    {

        /* Increment the main thread's counter.  */
        main_thread_counter++;
    }
}
```

px5_pthread_suspend

C Prototype:

```
#include <pthread.h>

int  px5_pthread_suspend(pthread_t thread_handle);
```

Description:

This pthread+ service suspends the specified thread. If the specified thread is already suspended, an error is returned.

Care should be taken to avoid suspending threads in critical execution paths, such as holding needed system resources or in process of making PX5 RTOS API calls. In general, this API should be used by the executing thread or on free-running threads.

API Parameters:

`thread_handle`	Handle of thread to suspend.

Return Codes:

PX5_SUCCESS (0)	Successful thread suspension.
EINVAL	Thread handle is not valid.
EBUSY	Specified thread is already suspended.

Real-time Scenarios:

Upon the successful execution of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. If the thread suspended in not the executing threasd, there is no preemption.

S **SUSPENSION**. If the currently executing thread calls this service, is is suspended until another thread calls *px5_pthread_resume*.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

px5_pthread_resume

Small Example:

```
#include <pthread.h>

int              status;



    /* Suspend the the currentl executing thread.  */
    status =  px5_pthread_suspend(pthread_self());

    /* If status contains PX5_SUCCESS, the current thread was
       suspended. */
```

px5_pthread_tick_sleep

C Prototype:

```
#include <pthread.h>

int px5_pthread_tick_sleep(tick_t ticks_to_sleep);
```

Description:

This pthreads+ service suspends the calling thread for the number of timer ticks specified by ticks_to_sleep. If a value of 0 ticks is supplied, this service returns immediately, i.e. the calling thread is not suspended.

This API is a cancellation point, meaning that if a cancellation is pending, it will be detected and executed by this API.

API Parameters:

`ticks_to_sleep`	This specifies the number of ticks the calling thread will be suspended for. If a value of zero is supplied, this service returns immediately.

Return Codes:

PX5_SUCCESS (0)	Successful thread tick sleep.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. If the number of ticks to sleep is zero, no preemption takes place.

S **SUSPENSION**. If the number of ticks to sleep is non-zero, the calling thread is suspended for that amount of ticks.

PREEMPTION. After the number of ticks has expired, the calling thread is resumed and will preempt the currently running thread if it is higher priority.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

sleep, nanosleep, usleep

Small Example:

```
#include <pthread.h>

unsigned long   my_example_thread_counter;


/* Define example thread.  */
void *  my_example_thread_start(void *arguments)
{

    /* Loop forever incrementing a counter and sleeping for 100
       ticks. */
    while (1)
    {
        /* Increment my example thread's counter. */
        my_example_thread_counter++;

        /* Sleep for 100 ticks, which is 1 second @10ms tick rate.  */
        px5_pthread_tick_sleep(100);
    }
}
```

px5_pthread_ticks_get

C Prototype:

```
#include <pthread.h>

tick_t px5_pthread_ticks_get(void);
```

Description:

This pthreads+ service returns the internal tick count, which represents the number of timer interrupts.

API Parameters:

`none`

Return Codes:

current tick count

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. No preemption takes place as a result of this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

px5_pthread_ticks_set

Small Example:

```
#include <pthread.h>

unsigned long   current_tick_count;



/* Pickup the current tick count.  */
current_tick_count =  px5_pthread_ticks_get();
```

px5_pthread_ticktimer_create

C Prototype:

```
#include <pthread.h>

int px5_pthread_ticktimer_create(pthread_ticktimer_t *ticktimer_handle,
       pthread_ticktimerattr_t *  attributes,
       void (*expiration_routine)(pthread_ticktimer_t *, void *),
       void *argument, tick_t initial_ticks, tick_t reload_ticks);
```

Description:

This pthreads+ service creates a ticktimer that calls the specified expiration routine after the specified number of initial ticks. If there is a non-zero value supplied for reload, this ticktimer will operate on a periodic basis.

The ticktimer is created in a stopped state. To start the ticktimer, please call the px5_pthread_ticktimer_start API after successful creation.

API Parameters:

`ticktimer_handle`	Upon successful completion, returned handle of the ticktimer.
`attributes`	Optional attributes for the ticktimer creation.
`expiration_routine`	Application routine to call when the ticktimer expires.
`argument`	Argument pointer that is passed verbatim when the application expiration routine is called.
`initial_ticks`	Number of initial ticks before expiration. This value must be non-zero.
`reload_ticks`	Number of ticks for periodic expriation. If this value is zero, this ticktimer is a one-shot ticktimer.

Return Codes:

PX5_SUCCESS (0)	Successful ticktimer creation.
EINVAL	Invalid ticktimer handle pointer, expiration routine, or initial ticks value.
ENOMEM	Insufficient memory to create ticktimer.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

px5_ticktimer_*, px5_ticktimer_destroy

Small Example:

```
#include <pthread.h>

/* Ticktimer handle.  */
pthread_ticktimer_t    my_ticktimer_handle;
int                    status;

/* Create a ticktimer that first expires after 10 ticks and then
   every 5 ticks thereafter. Upon each expiration, the
   "my_expiration_routine" will be called.  */
status =  px5_pthread_ticktimer_create(&my_ticktimer_handle,
                              my_expiration_routine, NULL, 10, 5);

/* If status is PX5_SUCCESS, the ticktimer was created and is ready
   to be started via px5_pthread_ticktimer_start.  */
```

px5_pthread_ticktimer_destroy

C Prototype:

```
#include <pthread.h>

int px5_pthread_ticktimer_destroy(pthread_ticktimer_t
                                              *ticktimer_handle);
```

Description:

This pthreads+ service destroys a previously created, but stopped ticktimer.

API Parameters:

`ticktimer_handle`	Handle of the ticktimer to destroy.

Return Codes:

PX5_SUCCESS (0)	Successful ticktimer destroy.
EINVAL	Invalid ticktimer handle.
EBUSY	Ticktimer is not stopped, i.e., still active.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

px5_ticktimer_*, px5_ticktimer_create

Small Example:

```
#include <pthread.h>

/* Ticktimer handle.  */
pthread_ticktimer_t    my_ticktimer_handle;
int                    status;

/* Destroy the previously created ticktimer “my_ticktimer_handle”.  */
status =  px5_pthread_ticktimer_destroy(&my_ticktimer_handle);

/* If status is PX5_SUCCESS, the ticktimer was destroyed.  */
```

px5_pthread_ticktimer_start

C Prototype:

```
#include <pthread.h>

int px5_pthread_ticktimer_start(pthread_ticktimer_t *ticktimer_handle);
```

Description:

This pthreads+ service starts a previously created ticktimer.

API Parameters:

`ticktimer_handle`	Handle of the ticktimer to start.

Return Codes:

PX5_SUCCESS (0)	Successful ticktimer start.
EINVAL	Invalid ticktimer handle.
EBUSY	Ticktimer was already started.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

px5_ticktimer_*, px5_ticktimer_stop

Small Example:

```
#include <pthread.h>

/* Ticktimer handle.  */
pthread_ticktimer_t    my_ticktimer_handle;
int                    status;

/* Start the previously created ticktimer “my_ticktimer_handle”.  */
status =  px5_pthread_ticktimer_start(&my_ticktimer_handle);

/* If status is PX5_SUCCESS, the ticktimer was started.  */
```

px5_pthread_ticktimer_stop

C Prototype:

```
#include <pthread.h>

int px5_pthread_ticktimer_stop(pthread_ticktimer_t *ticktimer_handle);
```

Description:

This pthreads+ service stops a previously created ticktimer. When stopped, the remaining number of ticks before expiration is saved. If the ticktimer is re-started, it is restarted with the previous remaining number of ticks.

API Parameters:

`ticktimer_handle`	Handle of the ticktimer to stop.

Return Codes:

PX5_SUCCESS (0)	Successful ticktimer stop.
EINVAL	Invalid ticktimer handle, or the ticktimer was already stopped.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

px5_ticktimer_*, px5_ticktimer_start

Small Example:

```
#include <pthread.h>

/* Ticktimer handle.  */
pthread_ticktimer_t    my_ticktimer_handle;
int                    status;

/* Stop the previously created ticktimer "my_ticktimer_handle".  */
status =  px5_pthread_ticktimer_stop(&my_ticktimer_handle);

/* If status is PX5_SUCCESS, the ticktimer was stopped.  */
```

px5_pthread_ticktimer_update

C Prototype:

```
#include <pthread.h>

int px5_pthread_ticktimer_update(pthread_ticktimer_t *ticktimer_handle,
                                 tick_t initial_ticks, tick_t reload_ticks);
```

Description:

This pthreads+ service updates the initial ticks and reload ticks of a previously created but stopped ticktimer

API Parameters:

`ticktimer_handle`	Handle of ticktimer to update.
`initial_ticks`	Updated number of initial ticks before expiration. This value must be non-zero.
`reload_ticks`	Updated number of ticks for periodic expiration. If this value is zero, this ticktimer is a one-shot ticktimer.

Return Codes:

PX5_SUCCESS (0)	Successful ticktimer update.
EINVAL	Invalid ticktimer handle pointer or new initial ticks value.
EBUSY	Ticktimer is running. It must be stopped to use this service.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

px5_ticktimer_*, px5_ticktimer_stop

Small Example:

```
#include <pthread.h>

/* Ticktimer handle.  */
pthread_ticktimer_t    my_ticktimer_handle;
int                    status


/* Update a ticktimer to first expires after 20 ticks and then every 2
   ticks thereafter. */
status =  px5_pthread_ticktimer_update(&my_ticktimer_handle, 20, 2);

/* If status is PX5_SUCCESS, the ticktimer was updated with the new
   initial and reload values.  */
```

px5_pthread_ticktimerattr_destroy

C Prototype:

```
#include <pthread.h>

int px5_pthread_ticktimerattr_destroy(pthread_ticktimerattr_t *
                                      ticktimer_attributes);
```

Description:

This pthreads+ service destroys the previously created ticktimer attributes structure pointed to by *ticktimer_attributes*. Once destroyed, the ticktimer attributes structure cannot be used again unless it is recreated.

API Parameters:

`ticktimer_attributes`	Pointer to the ticktimer attributes to destroy.

Return Codes:

PX5_SUCCESS (0)	Successful ticktimer attributes destroy.
EINVAL	Invalid ticktimer attributes pointer.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

px5_pthread_ticktimer_*, px5_pthread_ticktimerattr_*, px5_pthread_ticktimerattr_init

Small Example:

```
#include <pthread.h>

/* Ticktimer attribute structure.  */
pthread_ticktimerattr_t    my_ticktimer_attributes;
int                        status;


    /* Destroy the ticktimer attributes referenced by
       "my_ticktimer_attributes". */
    status =  px5_pthread_ticktimerattr_destroy(
                                  &my_ticktimer_attributes);

    /* If status is PX5_SUCCESS, the ticktimer attributes structure
       was destroyed.  */
```

px5_pthread_ticktimerattr_getcontroladdr

C Prototype:

```
#include <pthread.h>

int px5_pthread_ticktimerattr_getcontroladdr(pthread_ticktimerattr_t
       * ticktimer_attributes, void **  ticktimer_control_address);
```

Description:

This pthreads+ service returns the previously supplied ticktimer control structure memory address.

API Parameters:

`ticktimer_attributes`	Pointer to the ticktimer attributes.
`ticktimer_control_address`	Pointer to the destination for the previously supplied ticktimer control address.

Return Codes:

PX5_SUCCESS (0)	Successful ticktimer attributes ticktimer control address retrieval.
EINVAL	Invalid ticktimer attributes or ticktimer control address designation pointer.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

px5_pthread_ticktimer_*, px5_pthread_ticktimerattr_*, px5_pthread_ticktimerattr_setcontroladdr, px5_pthread_ticktimerattr_getcontrolsize

Small Example:

```
#include <pthread.h>

/* Ticktimer attribute structure.  */
pthread_ticktimerattr_t    my_ticktimer_attributes;
void *                     my_ticktimer_control_address;
int                        status;



    /* Get the ticktimer control structure address in the ticktimer
       attributes structure "my_ticktimer_attributes". */
    status = px5_pthread_ticktimerattr_getcontroladdr(
                    &my_ticktimer_attributes,
                    &my_ticktimer_control_address);

    /* If status is PX5_SUCCESS, "my_ticktimer_control_address"
        contains the address of the previously supplied ticktimer
        control memory.  */
```

px5_pthread_ticktimerattr_getcontrolsize

C Prototype:

```
#include <pthread.h>

int px5_pthread_ticktimerattr_getcontrolsize(pthread_ticktimerattr_t
       *ticktimer_attributes, size_t *  ticktimer_control_size);
```

Description:

This pthreads+ service returns the size of the internal ticktimer control structure. The main purpose of this API is to inform the application how much memory is required for the *px5_pthread_ticktimerattr_setcontroladdr* API.

API Parameters:

`ticktimer_attributes`	Pointer to the ticktimer attributes.
`ticktimer_control_size`	Pointer to the destination for the internal ticktimer control structure size.

Return Codes:

PX5_SUCCESS (0)	Successful retrieval of internal ticktimer control structure size.
EINVAL	Invalid ticktimer attributes or ticktimer control structure size destination pointer.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NO PREEMPTION. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

px5_pthread_ticktimer_*, px5_pthread_ticktimerattr_*, px5_pthread_ticktimerattr_setcontroladdr

Small Example:

```
#include <pthread.h>

/* Ticktimer attribute structure.  */
pthread_ticktimerattr_t    my_ticktimer_attributes;
size_t                     my_ticktimer_control_size;
int                        status;


    /* Get the ticktimer control structure memory size. */
    status = px5_pthread_ticktimerattr_getcontrolsize(
       &my_ticktimer_attributes, &my_ticktimer_control_size);

    /* If status is PX5_SUCCESS, "my_ticktimer_control_size"
        contains the size of the internal ticktimer
        control structure.  */
```

px5_pthread_ticktimerattr_getname

C Prototype:

```
#include <pthread.h>

int px5_pthread_ticktimerattr_getname(pthread_ticktimerattr_t
              *ticktimer_attributes, char **  ticktimer_name);
```

Description:

This pthreads+ service returns the previously supplied ticktimer name.

API Parameters:

`ticktimer_attributes`	Pointer to the ticktimer attributes.
`ticktimer_name`	Pointer to the destination for the previous ticktimer name.

Return Codes:

PX5_SUCCESS (0)	Successful ticktimer attributes ticktimer name retrieval.
EINVAL	Invalid ticktimer attributes or ticktimer name pointer.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

*px5_pthread_ticktimer_*, px5_pthread_ticktimerattr_**

Small Example:

```
#include <pthread.h>

/* Ticktimer attribute structure.  */
pthread_ticktimerattr_t    my_ticktimer_attributes;
char *                     my_ticktimer_name;
int                        status;




    /* Get the previous ticktimer name. */
    status =  px5_pthread_ticktimerattr_getname(
                    &my_ticktimer_attributes, &my_ticktimer_name);

    /* If status is PX5_SUCCESS, "my_ticktimer_name" contains the
       name previously supplied. */
```

px5_pthread_ticktimerattr_init

C Prototype:

```
#include <pthread.h>

int px5_pthread_ticktimerattr_init(pthread_ticktimerattr_t *
                                   ticktimer_attributes);
```

Description:

This pthreads+ service initializes the ticktimer attributes structure with default ticktimer creation values.

API Parameters:

`ticktimer_attributes`	Pointer to the ticktimer attributes structure to create.

Return Codes:

PX5_SUCCESS (0)	Successful ticktimer attributes structure creation.
EINVAL	Invalid ticktimer attributes pointer.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

px5_pthread_ticktimer_*, px5_pthread_ticktimerattr_*, px5_pthread_ticktimerattr_destroy

Small Example:

```
#include <pthread.h>

/* Ticktimer attribute structure.  */
pthread_ticktimerattr_t    my_ticktimer_attributes;
int                        status;




    /* Create the ticktimer attributes structure
       "my_ticktimer_attributes". */
    status =  px5_pthread_ticktimerattr_init(&my_ticktimer_attributes);

    /* If status is PX5_SUCCESS, the "my_ticktimer_attributes"
       structure is ready for use.  */
```

px5_pthread_ticktimerattr_setcontroladdr

C Prototype:

```
#include <pthread.h>

int px5_pthread_ticktimerattr_setcontroladdr(pthread_ticktimerattr_t
        * ticktimer_attributes, void *  ticktimer_control_address,
                                       size_t ticktimer_control_size);
```

Description:

This pthreads+ service provides a mechanism for the user to provide the memory for the internal PX5 RTOS ticktimer structure, as specified by the address contained in the *ticktimer_control_address* parameter. This memory will subsequently be used for the next ticktimer created with this attribute structure. The size of the memory required for the internal ticktimer control structure can be found via a call to the *px5_pthread_ticktimerattr_getcontrolsize* service.

Note that each ticktimer created must have its own unique ticktimer control structure memory. Hence, the ticktimer control address supplied here is only valid for one px5_pthread_ticktimer_create call.

API Parameters:

`ticktimer_attributes`	Pointer to the ticktimer attributes.
`ticktimer_control_address`	Pointer to the supplied ticktimer control structure memory.
`ticktimer_control_size`	Size of memory specified.

Return Codes:

PX5_SUCCESS (0)	Successful internal ticktimer control address set.
EINVAL	Invalid ticktimer attributes or memory size.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NO PREEMPTION. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

px5_pthread_ticktimer_*, px5_pthread_ticktimerattr_*, px5_pthread_ticktimerattr_getcontrolsize

Small Example:

```
#include <pthread.h>

/* Ticktimer attribute structure.  */
pthread_ticktimerattr_t    my_ticktimer_attributes;
int                        status;



    /* Provide the memory for the ticktimer control structure in the
       next ticktimer create call by placing it's address in the
       attributes structure "my_ticktimer_attributes". */
    status =  pthread_ticktimerattr_setcontroladdr(
             &my_ticktimer_attributes, 0x60000, 80);

    /* If status is PX5_SUCCESS, the next ticktimer creation using
      these attributes will use memory at address 0x60000 for the
      internal ticktimer control structure. */
```

px5_pthread_ticktimerattr_setname

C Prototype:

```
#include <pthread.h>

int px5_pthread_ticktimerattr_setname(pthread_ticktimerattr_t
                    * ticktimer_attributes, char *  ticktimer_name);
```

Description:

This pthreads+ service sets the ticktimer name in the specified attribute structure.

API Parameters:

`ticktimer_attributes`	Pointer to the ticktimer attributes.
`ticktimer_name`	Pointer to the supplied ticktimer name.

Return Codes:

PX5_SUCCESS (0)	Successful ticktimer attributes ticktimer name set.
EINVAL	Invalid ticktimer attributes.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NO PREEMPTION. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

px5_pthread_ticktimer_*, px5_pthread_ticktimerattr_*, px5_pthread_ticktimerattr_getname

Small Example:

```
#include <pthread.h>

/* Ticktimer attribute structure.  */
pthread_ticktimerattr_t    my_ticktimer_attributes;
int                        status;


    /* Set the ticktimer name in the ticktimer attributes structure
       "my_ticktimer_attributes". */
    status =  px5_pthread_ticktimerattr_setname(
                    &my_ticktimer_attributes, "my_ticktimer_name");

    /* If status is PX5_SUCCESS, "my_ticktimer_name" is set in the
       ticktimer attribute structure.  */
```

px5_sem_extend_init

C Prototype:

```
#include <semaphore.h>

int px5_sem_extend_init(sem_t *  semaphore_handle, int pshared,
             unsigned int value, semattr_t *  semaphore_attributes);
```

Description:

This pthreads+ service extension initializes (creates) a semaphore with the specified initial value and with optional PX5 semaphore attributes. If successful, the semaphore handle is available for use by the application. This service is an alternative to the *sem_init* API, i.e., all semaphore APIs can be used on the semaphore created with this extended API.

API Parameters:

`semaphore_handle`	Handle of the semaphore to setup.
`pshared`	Process sharing selection - not used by the PX5 RTOS.
`value`	Initial value of the semaphore.
`semaphore_attributes`	Optional semaphore attributes.

Return Codes:

PX5_SUCCESS (0)	Successful semaphore initialization.
PX5_ERROR (-1)	Error attempting to initialize the semaphore. Please use *errno* to retrieve the exact error:

EINVAL	Invalid semaphore handle or value exceeds SEM_VALUE_MAX.
ENOSPC	Insufficient memory to create semaphore.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

sem, sem_destroy, sem_init, px5_semattr_**

Small Example:

```
#include <semaphore.h>

/* Semaphore handle.  */
sem_t               my_semaphore_handle;

/* Semaphore attributes.  */
semattr_t           my_semaphore_attributes;


int                 status;



    /* Create the semaphore and setup "my_semaphore_handle" using
       the already initialized semaphore attributes
       "my_semaphore_attributes". */
    status =  px5_sem_extend_init(&my_semaphore_handle, 0, 1,
                                   &my_semaphore_attributes);

    /* If status is PX5_SUCCESS, the semaphore was created with a value
       of 1 and is ready to use!  */
```

px5_semattr_destroy

C Prototype:

```
#include <semaphore.h>

int px5_semattr_destroy(semttr_t * semaphore_attributes);
```

Description:

This pthreads+ service destroys the previously created semaphore attributes structure pointed to by *semaphore_attributes*. Once destroyed, the semaphore attributes structure cannot be used again unless it is recreated.

API Parameters:

`semaphore_attributes`	Pointer to the semaphore attributes to destroy.

Return Codes:

PX5_SUCCESS (0)	Successful semaphore attributes destroy.
EINVAL	Invalid semaphore attributes pointer.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

*px5_sem_extend_init, px5_semattr_init, px5_semattr_**

Small Example:

```
#include <semaphore.h>

/* Semaphore attribute structure.  */
semattr_t              my_semaphore_attributes;
int                    status;




    /* Destroy the semaphore attributes referenced by
       "my_semaphore_attributes". */
    status =  px5_semattr_destroy(&my_semaphore_attributes);

    /* If status is PX5_SUCCESS, the semaphore attributes
       structure was destroyed.  */
```

px5_semattr_getcontroladdr

C Prototype:

```
#include <semaphore.h>

int px5_semattr_getcontroladdr(semattr_t *  semaphore_attributes,
              void **  semaphore_control_address);
```

Description:

This pthreads+ service returns the previously supplied semaphore control structure address.

API Parameters:

`semaphore_attributes`	Pointer to the semaphore attributes.
`semaphore_control_address`	Pointer to the destination for the previously supplied semaphore control address.

Return Codes:

PX5_SUCCESS (0)	Successful semaphore attributes control address retrieval.
EINVAL	Invalid semaphore attributes or semaphore control address designation pointer.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

*px5_sem_extend_init, px5_semattr_init, px5_semattr_setcontroladdr, px5_semattr_**

Small Example:

```
#include <semaphore.h>

/* Semaphore attribute structure.  */
semattr_t               my_semaphore_attributes;
void *                  my_semaphore_control_address;
int                     status;



    /* Get the semaphore control structure address in the
       semaphore attributes structure "my_semaphore_attributes". */
    status = px5_semattr_getcontroladdr(&my_semaphore_attributes,
                                        &my_semaphore_control_address);

    /* If status is PX5_SUCCESS, "my_semaphore_control_address"
        contains the address of the previously supplied semaphore
        control memory.  */
```

px5_semattr_getcontrolsize

C Prototype:

```
#include <semaphore.h>

int px5_semattr_getcontrolsize(semattr_t *  semaphore_attributes,
                               size_t *  semaphore_control_size);
```

Description:

This pthreads+ service returns the size of the internal semaphore control structure. The main purpose of this API is to inform the application how much memory is required for the *px5_semattr_setcontroladdr* API.

API Parameters:

`semaphore_attributes`	Pointer to the attributes.
`semaphore_control_size`	Pointer to the destination for the internal semaphore control structure size.

Return Codes:

PX5_SUCCESS (0)	Successful retrieval of internal semaphore control structure size.
EINVAL	Invalid semaphore attributes or invalid destination for semaphore control structure size.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

*px5_sem_extend_init, px5_semattr_init, px5_semattr_setcontroladdr, px5_semattr_**

Small Example:

```
#include <semaphore.h>

/* Semaphore variable attribute structure.  */
semattr_t             my_semaphore_attributes;
size_t                my_semaphore_control_size;
int                   status;



    /* Get the internal semaphore control structure memory size. */
    status = px5_semattr_getcontrolsize(
                                 &my_semaphore_attributes,
                                 &my_semaphore_control_size);

    /* If status is PX5_SUCCESS, "my_semaphore_control_size"
       contains the size of the internal semaphore control
       structure.  */
```

px5_semattr_getname

C Prototype:

```
#include <semaphore.h>

int px5_semattr_getname(semattr_t *  semaphore_attributes,
                        char **  semaphore_name);
```

Description:

This pthreads+ service returns the previously supplied semaphore name.

API Parameters:

`semaphore_attributes`	Pointer to the semaphore attributes.
`semaphore_name`	Pointer to the destination for the previously supplied semaphore name.

Return Codes:

PX5_SUCCESS (0)	Successful retrieval of last supplied semaphore name.
EINVAL	Invalid semaphore attributes or name destination pointer.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

*px5_sem_extend_init, px5_semattr_init, px5_semattr_setname, px5_semattr_**

Small Example:

```
#include <semaphore.h>

/* Semaphore attribute structure.  */
semttr_t               my_semaphore_attributes;
char *                 my_semaphore_name;
int                    status;



    /* Get the last supplied semaphore name. */
    status =  px5_semattr_getname(&my_semaphore_attributes,
                                           &my_semaphore_name);

    /* If status is PX5_SUCCESS, "my_semaphore_name" contains the
       name previously supplied. */
```

px5_semattr_init

C Prototype:

```
#include <semaphore.h>

int px5_semattr_init(semattr_t * semaphore_attributes);
```

Description:

This pthreads+ service initializes the semaphore attributes structure with default semaphore variable creation values. Note that semaphore attributes are used only by the *px5_sem_extend_init* API.

API Parameters:

`semaphore_attributes`	Pointer to the semaphore attributes structure to create.

Return Codes:

PX5_SUCCESS (0)	Successful semaphore attributes structure creation.
EINVAL	Invalid semaphore attributes pointer.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

*px5_sem_extend_init, px5_semattr_destroy, px5_semattr_**

Small Example:

```
#include <semaphore.h>

/* Semaphore attribute structure.  */
semattr_t               my_semaphore_attributes;
int                     status;




    /* Create the semaphore attributes structure
      "my_semaphore_attributes". */
    status =  px5_semattr_init(&my_semaphore_attributes);

    /* If status is PX5_SUCCESS, the "my_semaphore_attributes"
       structure is ready for use.  */
```

px5_semattr_setcontroladdr

C Prototype:

```
#include <semaphore.h>

int px5_semattr_setcontroladdr(semattr_t *  semaphore_attributes,
                               void *  semaphore_control_address,
                               size_t  semaphore_control_size);
```

Description:

This pthreads+ service provides a mechanism for the user to provide the memory for the internal PX5 RTOS semaphore structure, as specified by the address contained in the *semaphore_control_address* parameter. This memory will subsequently be used for the next semaphore created with this attribute structure. The size of the memory required for the internal semaphore control structure can be found via a call to the *px5_semattr_getcontrolsize* service.

Note that each semaphore created must have its own unique semaphore control structure memory. Hence, the semaphore control memory supplied here is only valid for one px5_sem_extend_init call.

API Parameters:

`semaphore_attributes`	Pointer to the semaphore attributes.
`semaphore_control_address`	Pointer to the internal semaphore control structure memory.
`semaphore_control_size`	Size of specified semaphore control structure memory.

Return Codes:

PX5_SUCCESS (0)	Successful specification of semaphore structure memory.
EINVAL	Invalid semaphore attributes or invalid size of semaphore control memory.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NO PREEMPTION. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

*px5_sem_extend_init, px5_semattr_init, px5_semattr_getcontroladdr, px5_semattr_**

Small Example:

```
#include <semaphore.h>

/* Semaphore attribute structure.  */
semattr_t               my_semaphore_attributes;
int                     status;



    /* Set the semaphore control structure memory address in the
       semaphore attributes structure "my_semaphore_attributes". */
    status =  px5_semattr_setcontroladdr(&my_semaphore_attributes,
                                          0x70000, 60);
```

px5_semattr_setname

C Prototype:

```
#include <semaphore.h>

int px5_semattr_setname(semattr_t *  semaphore_attributes,
                        char *  semaphore_name);
```

Description:

This pthreads+ service sets the semaphore name in the specified attribute structure.

API Parameters:

`semaphore_attributes`	Pointer to the semaphore attributes.
`semaphore_name`	Pointer to the supplied semaphore name.

Return Codes:

PX5_SUCCESS (0)	Successful semaphore name set.
EINVAL	Invalid semaphore attributes.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

*px5_sem_extend_init, px5_semattr_init, px5_semattr_getname, px5_semattr_**

Small Example:

```
#include <semaphore.h>

/* Semaphore attribute structure.  */
semattr_t               my_semaphore_attributes;
int                     status;



    /* Set the semaphore name in the semaphore attributes
       structure "my_semaphore_attributes". */
    status =  px5_semattr_setname(&my_semaphore_attributes,
                                                "my_semaphore_name");

    /* If status is PX5_SUCCESS, "my_semaphore_name" is set in the
       semaphore attribute structure.  */
```

sched_yield

C Prototype:

```
#include <sched.h>

int sched_yield(void);
```

Description:

This service relinquishes control to the next thread of the same priority currently ready for execution. If there is no other thread of the same priority ready for execution, this service simply returns a successful status.

API Parameters:

```
none
```

Return Codes:

PX5_SUCCESS (0)	Successful thread yield.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. If no other threads of the same priority are ready for execution, this service simply returns without any preemption.

P **PREEMPTION**. If there are other threads read for execution at the same priority level, the other thread(s) will execute before this service returns.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

px5_ticktimer_sleep, sleep, nanosleep

Small Example:

```
#include <sched.h>


unsigned long   my_first_thread_counter;
unsigned long   my_second_thread_counter;


/* Define first thread.  */
void *  my_first_thread_start(void *arguments)
{

    /* Loop forever incrementing a counter and yielding
       the processor to thread 1.  */
    while (1)
    {
        /* Increment my first thread’s counter. */
        my_first_thread_counter++;

        /* Yield to my second thread.  */
        sched_yield();

        /* Once sched_yield returns, my second thread has executed.  */
    }
}


/* Define my second thread.  */
void *  my_second_thread_start(void *arguments)
{

    /* Loop forever incrementing a counter and yielding
       the processor to my first thread.  */
    while (1)
    {
        /* Increment my second thread’s counter. */
        my_second_thread_counter++;
```

sem_destroy

C Prototype:

```
#include <semaphore.h>

int sem_destroy(sem_t *  semaphore_handle);
```

Description:

This service destroys the previously created semaphore specified by *semaphore_handle*. If there are one or more threads still suspended on this semaphore, an error is returned.

API Parameters:

`semaphore_handle`	Handle of the semaphore to destroy.

Return Codes:

PX5_SUCCESS (0)	Successful semaphore destroy.
PX5_ERROR (-1)	Error attempting to destroy semaphore. Please use *errno* to retrieve the exact error:

EINVAL	Invalid semaphore handle.
EBUSY	One or more threads are currently suspended on this semaphore.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

sem, sem_init*

Small Example:

```
#include <semaphore.h>

/* Semaphore handle.  */
sem_t               my_semaphore_handle;
int                 status;



    /* Destroy the semaphore referenced by "my_semaphore_handle". */
    status =  sem_destroy(&my_semaphore_handle);

    /* If status is PX5_SUCCESS, the semaphore was destroyed.  */
```

sem_init

C Prototype:

```
#include <semaphore.h>

int sem_init(sem_t *  semaphore_handle, int pshared,
                                               unsigned int value);
```

Description:

This service initializes (creates) a semaphore with the specified initial value. If successful, the semaphore handle is available for use by the application.

API Parameters:

`semaphore_handle`	Handle of the semaphore to setup.
`pshared`	Process sharing selection - not used by the PX5 RTOS.
`value`	Initial value of the semaphore.

Return Codes:

PX5_SUCCESS (0)	Successful semaphore initialization.
PX5_ERROR (-1)	Error attempting to initialize the semaphore. Please use *errno* to retrieve the exact error:

EINVAL	Invalid semaphore handle or value exceeds SEM_VALUE_MAX.
ENOSPC	Insufficient memory to create semaphore.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NO PREEMPTION. There is no preemption possible with this service.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

sem, sem_destroy, px5_sem_extend_init, px5_semattr_**

Small Example:

```
#include <semaphore.h>

/* Semaphore handle.  */
sem_t               my_semaphore_handle;
int                 status;



    /* Create the semaphore and setup "my_semaphore_handle". */
    status =  sem_init(&my_semaphore_handle, 0, 1);

    /* If status is PX5_SUCCESS, the semaphore was created with a value
       of 1 and is ready to use!  */
```

sem_post

C Prototype:

```
#include <semaphore.h>

int sem_post(sem_t *  semaphore_handle);
```

Description:

This service posts to the specified semaphore. If one or more threads are waiting, the first thread waiting is resumed. Otherwise, if no threads are waiting, the internal semaphore count is incremented by 1.

API Parameters:

`semaphore_handle`	Handle of the semaphore to post.

Return Codes:

PX5_SUCCESS (0)	Successful semaphore post.
PX5_ERROR (-1)	Error attempting to post to the semaphore. Please use *errno* to retrieve the exact error:

EINVAL	Invalid semaphore handle or value exceeds SEM_VALUE_MAX.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. If there are no other threads waiting for the semaphore, no preemption takes place.

P **PREEMPTION**. If a higher-priority thread was waiting for the semaphore, when it is given the semaphore, the waiting thread is resumed, and preemption will occur.

Callable From:

This service is callable from the thread context and from interrupt handlers (ISRs).

See Also:

sem, sem_wait, sem_trywait*

Small Example:

```
#include <semaphore.h>

/* Semaphore handle.  */
sem_t              my_semaphore_handle;
int                status;



    /* Post to the semaphore "my_semaphore_handle". */
    status =  sem_post(&my_semaphore_handle);

    /* If status is PX5_SUCCESS, the post was made to the semaphore. */
```

sem_trywait

C Prototype:

```
#include <semaphore.h>

int sem_trywait(sem_t *  semaphore_handle);
```

Description:

If the semaphore is available (count greater than zero), this service decrements the count and returns success. Otherwise, an error is returned.

API Parameters:

`semaphore_handle`	Handle of the semaphore to try to get.

Return Codes:

PX5_SUCCESS (0)	Successful semaphore get.
PX5_ERROR (-1)	Error attempting to get the semaphore. Please use *errno* to retrieve the exact error:

EINVAL	Invalid semaphore handle.
EAGAIN	Semaphore is not available.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. If the semaphore was available and ownership assigned to the calling thread, no preemption takes place.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

sem, sem_wait*

Small Example:

```
#include <semaphore.h>

/* Semaphore handle.  */
sem_t               my_semaphore_handle;
int                 status;



    /* Try to get the semaphore "my_semaphore_handle". */
    status =  sem_trywait(&my_semaphore_handle);

    /* If status is PX5_SUCCESS, the semaphore was retrieved. */
```

sem_wait

C Prototype:

```
#include <semaphore.h>

int sem_wait(sem_t *  semaphore_handle);
```

Description:

If the semaphore is available (count greater than zero), this service decrements the semaphore count by one and returns success to the caller. Otherwise, if the semaphore is zero, the calling thread suspends until the semaphore is available.

This API is a cancellation point, meaning that if a cancellation is pending, it will be detected and executed by this API.

API Parameters:

`semaphore_handle`	Handle of the semaphore to get.

Return Codes:

PX5_SUCCESS (0)	Successful semaphore get.
PX5_ERROR (-1)	Error attempting to get the semaphore. Please use *errno* to retrieve the exact error:

EINVAL	Invalid semaphore handle.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NO PREEMPTION. If the semaphore is available, no preemption takes place.

SUSPENSION. If the semaphore is not available (count is zero), the calling thread is suspended until the semaphore becomes available via *sem_post*.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

sem, sem_trywait, sem_post*

Small Example:

```
#include <semaphore.h>

/* Semaphore handle.  */
sem_t                  my_semaphore_handle;
int                    status;


    /* Get the semaphore "my_semaphore_handle". */
    status =  sem_wait(&my_semaphore_handle);

    /* If status is PX5_SUCCESS, the semaphore was retrieved.  */
```

sigaction

C Prototype:

```
#include <signal.h>

int sigaction(int signal_number, struct sigaction * new_handler,
                                 struct sigaction * previous_handler);
```

Description:

This service sets up the signal handler for the specified signal number. If the previous handler pointer is non-NULL, the information for the previous signal handler is returned as well.

Signal handlers are only invoked if the corresponding signal is unmasked by the thread receiving the raised signal. A signal handler should be setup via sigaction before any signal is enabled.

API Parameters:

`signal_number`	Signal number to set up handler for.
`new_handler`	Pointer to signal action structure that contains signal information, including the handler. This pointer is NULL if handler is being removed.
`previous_handler`	Optional pointer to structure to store the previous handler information. If the previous handler information is not wanted, this value is NULL.

Return Codes:

PX5_SUCCESS (0)	Successful signal handler setup.
PX5_ERROR (-1)	Error attempting to set up signal handler. Please use *errno* to retrieve the exact error:

EINVAL	Invalid signal number or invalid new signal handler.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NO PREEMPTION. This service only sets up the signal handler, so no preemption is possible.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

sig, pthread_kill, pthread_sigmask, sigwait*

Small Example:

```
#include <signal.h>

struct sigaction   my_signal_handler_request;
struct sigaction   previous_signal_handler;
int                my_signal_handler_count;


    /* My signal handler.  */
    void my_signal_handler(int signal)
    {

       /* Signal will be equal to SIGUSR1  */

       /* Increment signal count.  */
       my_signal_handler_count++;

    }

    /* Later in the thread processing…  */

    /* Setup the signal handler for SIGUSR1.  */
    my_signal_handler_request.signal_handler =  my_signal_handler;
    status =  sigaction(SIGUSR1, &my_signal_handler_request,
                                 &prevous_signal_handler);
```

```
/* If status is PX5_SUCCESS (0), “my_signal_handler” is set up for
   SIGUSR1. Any pthread_kill request to a thread with SIGUSR1
   enabled will result in a call to “my_signal_handler”.  */
```

sigaddset

C Prototype:

```
#include <signal.h>

int sigaddset(sigset_t *  signal_set, int signal_number);
```

Description:

This service adds the specified *signal_number* to the set of signals.

API Parameters:

`signal_set`	Bit map of signals (signal set).
`signal_number`	Number of signal to add to the signal set.

Return Codes:

PX5_SUCCESS (0)	Successful signal add to set.
PX5_ERROR (-1)	Error attempting to add signal to set . Please use *errno* to retrieve the exact error:

EINVAL	Invalid signal number or invalid signal set pointer.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. This service only adds the signal number in the signal set, so no preemption is possible.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

sig, sigdelset, sigemptyset, sigfillset*

Small Example:

```
#include <signal.h>

sigset_t           my_signal_set;



    /* Add the SIGUSR1 signal to "my_signal_set".  */
    status =  sigaddset(&my_signal_set, SIGUSR1);

    /* If status is PX5_SUCCESS (0), SIGUSR1 has been added to the signal
       set in "my_signal_set".  */
```

sigdelset

C Prototype:

```
#include <signal.h>

int sigdelset(sigset_t *  signal_set, int signal_number);
```

Description:

This service deletes the specified *signal_number* from the set of signals.

API Parameters:

`signal_set`	Bit map of signals (signal set).
`signal_number`	Number of signal to remove from the signal set.

Return Codes:

PX5_SUCCESS (0)	Successful signal removal from set.
PX5_ERROR (-1)	Error attempting to remove signal from set . Please use *errno* to retrieve the exact error:

EINVAL	Invalid signal number or invalid signal set pointer.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. This service only deletes the signal number in the signal set, so no preemption is possible.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

sig, sigaddset, sigemptyset, sigfillset*

Small Example:

```
#include <signal.h>

sigset_t           my_signal_set;



    /* Remove the SIGUSR1 signal from "my_signal_set".  */
    status =  sigdelset(&my_signal_set, SIGUSR1);

    /* If status is PX5_SUCCESS, SIGUSR1 has been removed from the signal
       set "my_signal_set".  */
```

sigemptyset

C Prototype:

```
#include <signal.h>

int sigemptyset(sigset_t *  signal_set);
```

Description:

This service removes all signals from the specified set of signals.

API Parameters:

`signal_set`	Bit map of signals (signal set).

Return Codes:

PX5_SUCCESS (0)	Successful signal removal of all signals from set.
PX5_ERROR (-1)	Error attempting to remove all signals from set . Please use *errno* to retrieve the exact error:

EINVAL	Invalid signal set pointer.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. This service only removes all signals from the signal set, so no preemption is possible.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

sig, sigdelset, sigaddset, sigfillset*

Small Example:

```
#include <signal.h>

sigset_t           my_signal_set;



    /* Remove all signals from "my_signal_set".  */
    status =  sigemptyset(&my_signal_set);

    /* If status is PX5_SUCCESS (0), all signals have been removed from
       the signal set "my_signal_set".  */
```

sigfillset

C Prototype:

```
#include <signal.h>

int sigfillset(sigset_t *  signal_set);
```

Description:

This service sets all signals in the specified set of signals.

API Parameters:

`signal_set`	Bit map of signals (signal set).

Return Codes:

PX5_SUCCESS (0)	Successful signal setting of all signals in set.
PX5_ERROR (-1)	Error attempting to set all signals in set . Please use *errno* to retrieve the exact error:

EINVAL	Invalid signal set pointer.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. This service only sets signals in the signal set, so no preemption is possible.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

sig, sigdelset, sigemptyset, sigaddset*

Small Example:

```
#include <signal.h>

sigset_t           my_signal_set;



    /* Set all signals in "my_signal_set".  */
    status =  sigfillset(&my_signal_set);

    /* If status is PX5_SUCCESS (0), all signals have been set in
    "my_signal_set".  */
```

sigismember

C Prototype:

```
#include <signal.h>

int sigismember(sigset_t *  signal_set, int  signal_number);
```

Description:

This service determines if *signal_number* is part of the specified signal set.

API Parameters:

`signal_set`	Bit map of signals (signal set).
`signal_number`	Signal number to check for in the set.

Return Codes:

1	Signal is part of the set
0	Signal is not part of the set.
PX5_ERROR (-1)	Error attempting to set all signals in set . Please use *errno* to retrieve the exact error:

EINVAL	Invalid signal set pointer or invalid signal number.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. This service only checks for a signal in the signal set, so no preemption is possible.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

sig, sigaddset, sigdelset, sigemptyset, sigfillset, sigpending*

Small Example:

```
#include <signal.h>

sigset_t           my_signal_set;



    /* Check if SIGUSR1 is in the "my_signal_set" signal set.  */
    status =  sigismember(&my_signal_set, SIGUSR1);

    /* If status is 0, SIGUSR1 is not in the "my_signal_set" signal
       set.  */
```

sigpending

C Prototype:

```
#include <signal.h>

int sigpending(sigset_t *  pending_signals);
```

Description:

This service retrieves the pending signals for the calling thread.

API Parameters:

`pending_signals`	Pointer to destination for the pending signals of the calling thread.

Return Codes:

PX5_SUCCESS (0)	Successful pending signal retrieval.
PX5_ERROR (-1)	Error attempting to get the pending signals. Please use *errno* to retrieve the exact error:

EINVAL	Invalid pending signal pointer.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. This service only retrieves the pending signals of the calling thread, so no preemption is possible.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

sig, sigaddset, sigdelset, sigemptyset, sigfillset*

Small Example:

```
#include <signal.h>

sigset_t           my_pending_signals;



    /* Retrieve the pending signals for the calling thread. */
    status =  sigpending(&my_pending_signal);

    /* If status is PX5_SUCCESS (0), the pending signals for the calling
       thread can be found in "my_pending_signals".  */
```

sigtimedwait

C Prototype:

```
#include <signal.h>

int sigtimedwait(const sigset_t * signals, siginfo_t * signal_info,
                                                  const struct timespec *  timeout);
```

Description:

This service suspends the calling thread until a signal in the specified signal set is raised. If the signal is already pending, this service returns immediately with the corresponding signal number. If none of the signals in the set are pending, this service suspends the calling thread until a signal in the set is raised or until the specified timeout occurs.

All signals specified in the signal set must be masked in order to synchronously wait for them.

API Parameters:

`signals`	Bit map of signals (signal set) to wait for.
`signal_info`	Optional pointer to structure containing signal information.
`timeout`	Maximum time this service will wait before returning.

See Also:

sig, sigwait, sigwaitinfo*

Return Codes:

Positive Number	Signal raised.
PX5_ERROR (-1)	Error attempting to wait for signal. Please use *errno* to retrieve the exact error:

EINVAL	Invalid signal set or timeout value.
EAGAIN	Timeout occurred before signal was raised.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. A signal is already pending in the specified signal set, and this service returns immediately. No preemption takes place.

S **SUSPENSION**. If no signal in the specified signal set is pending, this service suspends the calling thread until a signal is raised via *pthread_kill* or the timeout occurs.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

sig, sigwait, sigwaitinfo, pthread_kill*

Small Example:

```
#include <signal.h>

int                 signal_received;
struct timespec     my_timeout;
sigset_t            my_signals;
```

```
/* Setup the signal set to specify SIGUSR1 signal.  */
sigemptyset(&my_signals);
sigaddset(&my_signals, SIGUSR1);

/* Setup timeout for 1 second. */
my_timeout.tv_sec =   1;
my_timeout.tv_nsec =  0;

/* Wait for SIGUSR1 signals. */
signal_received =  sigtimedwait(&my_signal, NULL, &my_timeout);

/* If signal_received is SIGUSR1 the service was successful.  */
```

sigwait

C Prototype:

```
#include <signal.h>

int sigwait(const sigset_t * signals, int *  signal_number);
```

Description:

This service suspends the calling thread until a signal in the specified signal set is raised. If the signal is already pending, this service returns immediately with the corresponding signal number. If none of the signals in the specified set are pending, this service suspends the calling thread until a signal in the set is raised.

All signals specified in the signal set must be masked in order to synchronously wait for them.

This API is a cancellation point, meaning that if a cancellation is pending, it will be detected and executed by this API.

API Parameters:

`signals`	Bit map of signals (signal set) to wait for.
`signal_number`	Pointer to return the signal within the specified set that was raised.

Return Codes:

PX5_SUCCESS (0)	Successful completion.
EINVAL	Error, invalid signal set, or signal number return pointer.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NO PREEMPTION. A signal is already pending in the specified signal set, and this service returns immediately. No preemption takes place.

SUSPENSION. If no signal in the specified signal set is pending, this service suspends the calling thread until a signal is raised via *pthread_kill*.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

sig, sigtimedwait, sigwaitinfo, pthread_kill*

Small Example:

```
#include <signal.h>

int                 signal_raised;
int                 status;
sigset_t            my_signals;



    /* Setup the signal set to specify SIGUSR1 signal.  */
    sigemptyset(&my_signals);
    sigaddset(&my_signals, SIGUSR1);

    /* Wait for SIGUSR1 signals. */
    status =  sigwait(&my_signal, &my_signal_number);

    /* If status is PX5_SUCCESS (0), my_signal_number contains
       SIGUSR1.  */
```

sigwaitinfo

C Prototype:

```
#include <semaphore.h>

int sigwaitinfo(const sigset_t * signals, siginfo_t * signal_info);
```

Description:

This service suspends the calling thread until a signal in the specified signal set is raised. If the signal is already pending, this service returns immediately with the corresponding signal number (and additional signal information in *signal_info*). If none of the signals in the specified set are pending, this service suspends the calling thread until a signal in the set is raised.

All signals specified in the signal set must be masked in order to synchronously wait for them.

This API is a cancellation point, meaning that if a cancellation is pending, it will be detected and executed by this API.

API Parameters:

`signals`	Bit map of signals (signal set) to wait for.
`signal_info`	Optional pointer to structure containing signal information.

Return Codes:

Positive Number	Signal raised.
PX5_ERROR (-1)	Error attempting to wait for signal. Please use *errno* to retrieve the exact error:

EINVAL	Invalid signal set.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. A signal is already pending in the specified signal set, and this service returns immediately. No preemption takes place.

S **SUSPENSION**. If no signal in the specified signal set is pending, this service suspends the calling thread until a signal is raised via *pthread_kill*.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

sig, sigwait, sigtimedwait, pthread_kill*

Small Example:

```
#include <signal.h>

int                 signal_received;
siginfo_t           my_signal_info;
sigset_t            my_signals;


    /* Setup the signal set to specify SIGUSR1 signal.  */
    sigemptyset(&my_signals);
    sigaddset(&my_signals, SIGUSR1);

    /* Wait for SIGUSR1 signals. */
    signal_received =  sigwaitinfo(&my_signal, &my_signal_info);

    /* If signal_received is SIGUSR1 the service was successful and
       my_signal_info.si_signo also contains SIGUSR1.  */
```

sleep

C Prototype:

```
#include <unistd.h>

unsigned int  sleep(unsigned int  seconds);
```

Description:

This service causes the calling thread to suspend for the number of seconds specified in *seconds*. If an unmasked signal is sent to the thread while sleeping, the thread is resumed, and the amount of remaining seconds is returned.

Sleep requests are rounded up to the next second.

This API is a cancellation point, meaning that if a cancellation is pending, it will be detected and executed by this API.

API Parameters:

`seconds`	The number of seconds to sleep.

Return Codes:

0	Successful sleep – no remaining seconds.
Positive number	Sleep was interrupted by a signal sent to this thread. This value represents the number of seconds left to sleep.

Real-time Scenarios:

Upon the successful execution of this service, the following real-time scenarios are possible:

SUSPENSION. The calling thread is suspended until the time specified has lapsed or until another thread sends a signal to this thread.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

nanosleep

Small Example:

```
#include "unistd.h"


unsigned int      remaining_seconds;


    /* Sleep for 5 second. */
    remaining_seconds =  sleep(5);

    /* If "remaining_seconds" is 0, the calling thread slept for
       5 seconds. */
```

time

C Prototype:

```
#include <pthread.h>

time_t time(time_t *  return_seconds);
```

Description:

This service returns the current number of seconds. If a non-null value for *return_seconds* is provide, the current number of seconds is also placed in the destination pointed to by *return_seconds*.

API Parameters:

`return_seconds`	If non-null, pointer to the destination of where to store the current seconds (in addition to current seconds returned by the function).

Return Value:

current seconds	Current seconds.

Real-time Scenarios:

Upon the successful completion of this service, the following real-time scenarios are possible:

NP **NO PREEMPTION**. No preemption takes place as a result of this service.,

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

pthread_ticks_get

Small Example:

```
#include <pthread.h>

/* Variable to store the current seconds.  */
time_t             my_current_seconds;

    /* Get the current seconds. */
    my_current_seconds =  time(NULL);

    /* At this point, my_current_seconds contains the current
       seconds.  */
```

usleep

C Prototype:

```
#include <unistd.h>

int  usleep(useconds_t microseconds);
```

Description:

This service causes the calling thread to suspend for the amount of time specified in *microseconds*. If an unmasked signal is sent to the thread while sleeping, the thread is resumed, and an error is returned.

usleep requests are rounded up to the next timer tick that is evenly divisible by the timer resolution.

This API is a cancellation point, meaning that if a cancellation is pending, it will be detected and executed by this API.

API Parameters:

`microsecond`	The amount of time in microseconds to sleep.

Return Codes:

PX5_SUCCESS (0)	Successful usleep.
PX5_ERROR (-1)	Error attempting to usleep. Please use *errno* to retrieve the exact error:

EINTR	usleep was interrupted by a signal.

Real-time Scenarios:

Upon the successful execution of this service, the following real-time scenarios are possible:

S **SUSPENSION**. The calling thread is suspended until the time specified has lapsed or until another thread sends a signal to this thread.

Callable From:

This service is only callable from the thread context, i.e., it may not be called from an interrupt handler.

See Also:

nanosleep, px5_pthread_tick_sleep, sleep

Small Example:

```
#include <unistd.h>


int                 status;


    /* Sleep for 1 second. */
    status =  usleep(1000000);

    /* If status contains PX5_SUCCESS (0), the calling thread slept for
       1 second. */
```

Index

PX5

Enhance • Simplify • Unite

11440 West Bernardo Court • Suite 300
San Diego, CA 92127, USA

Phone: +1 (858) 753-1715
Email: info@px5rtos.com
Website: px5rtos.com

© PX5 • All Rights Reserved