Platform Independence Layer API

Reference Guide

Control Plane-Platform Development Kit 2.11

March 2004

Contents

Revision History

Revision	Description	Date	Author
2.11	Updated for Release 2.11	March 2004	Amit Kaul
2.1	Updated for Release 2.1	December 2003	Dan Baumberger
2.0	Updated for Release 2.0	August 2003	Dan Baumberger

Part 1: Overview

1 Overview

In reusing software components for different applications, writing modules in high-level language is not sufficient for easy reuse. Differences in operating systems, platforms or compilers can make porting difficult. The porting effort can be greatly reduced by following guidelines to minimize compiler exceptions, and using an operating system-independent API. Section 2, Compiler Guidelines provides compiler guidelines, and Section 3, PIL API describes the Platform Independence Layer (PIL) API.

1.1 PIL

PIL provides an operating system independent API for many common operations. It also provides common debugging routines. PIL provides appropriately defined `trace` and `assert` macros (they are automatically disabled when compiled in release mode). With a specially built version, PIL keeps track of all the resources allocated through it and can report on the ones that were not freed.

PIL can provide other runtime support needs that an application may have. This may include implementing parts of the C or C++ run-time for environments that do not have native support. Other common utility code, such as heap management, can be included in the PIL, allowing all clients to benefit from the shared code.

The PIL API abstracts the common operating system services that applications and drivers use. By porting the PIL functionality to the new platform and rebuilding depending applications or drivers, much, if not all, of the code does not need to be ported, since it is not dependent on any single operating system but only on the PIL.

PIL services are divided into two categories: required and optional services. Required services are available in all implementations. Optional services may not be available on every implementation due to difficulty in emulation, or the services are not required for that implementation.

Required services include:

- Asynchronous Procedure Calls (APC)
- Critical sections
- Events
- Heap management
- Initialization and de-initialization
- Mailbox (messaging) management
- Memory management
- Semaphores
- Threading
- Timers
- PIL also provides utility services. These services include:
- Miscellaneous utility services
- ASSERT and DEBUG macros
- Allocated resource tracking

1.2 PIL Naming Conventions

All required PIL functions start with the prefix `PIL_`. This clearly distinguishes PIL calls from platform or application specific functions. Any function that starts with `PIL_` is guaranteed to be available on any platform for which a PIL has an implementation. Optional PIL services have the prefix `PILOS_` to distinguish that they may not be available in all implementations. When porting code with calls to `PILOS_` functions, the PIL implementation notes for the target implementation need to be consulted to see if optional services are available.

Platform-specific functions may be added to specific implementations of PIL if they follow a certain naming scheme. They cannot start with `PIL_` or `PILOS_`. They must start with the PIL implementation specific name such as `PIL32_` for Win32* or `PILVX_` for VxWorks*. If the service is needed in more than one implementation of PIL, it should be made a `PILOS_` optional service.

All PIL data structures start with the `PIL` prefix and follow the Pascal naming convention (for example, `PilSemaphore` or `PilCriticalSection`). PIL constants start with the prefix `PIL_` (for example, `PIL_OK` or `PIL_ERROR`).

1.3 Standard C Run-time

In an environment that provides a C compiler, it can be assumed that at least part of the standard C run-time library is available. PIL does not explicitly export those run-time functions, and it can be assumed that they are available elsewhere. Under certain environments, not all C run-time functions are available. In such cases, PIL may implement parts, although certainly not all, of the C run-time functions with their standard prototypes.

It is not specified that any implementation of PIL implements C run-time functions. However, as noted above, any specific implementation may be supporting the functions. Consult the implementation-specific notes for details on the run-time library that is supported.

1.4 Berkeley Sockets

For platforms where networking support is available, the preferred API for network programming is the standard Berkeley Sockets API as defined in the POSIX.1g specification and discussed in "Unix Network Programming" written by Richard Stevens. PIL does not attempt to place an abstraction to this API or provide any extensions to it.

1.5 Return Codes

The API descriptions in Section 3, PIL API documents many of the return codes returned as result codes from PIL functions. However, implementations may return codes that are not explicitly documented. This arises from the way certain implementations need to implement those services.

Certain return codes, if not otherwise, documented, can be returned from almost any function.

These codes include:

`PIL_ERROR`	Unless otherwise documented for a particular API call, this denotes an implementation-specific general error. The debug implementation of PIL should provide more information on why this error occurred.
`PIL_INVALID_PARAM`	Although documented in most cases, any function can return this error code if it determines that any of the given parameters is not valid.
`PIL_NOMEM`	Many implementations of functions may require memory to be allocated that other implementations may not need. In this case, `PIL_NOMEM` is the proper return code.

Part 2: Compiler Guidelines

2 Compiler Guidelines

Compilers can compile the same source code differently. To facilitate ease of porting, certain language constructs should be avoided so that they do not need to be `#ifdef` 'd or changed for each compiler.

The used compilers are Microsoft Visual C++ for Windows applications and drivers, and the GNU C Compiler Collection (gcc) for other platforms. Visual C++, even in "C" mode, accepts a wide number of language extensions. `.gcc`, when compiling C code, complies to the standards. When compiling C++ code or C code in C++ mode, `.gcc` closely follows the ANSI C++ standard, allowing many of the constructs that Visual C++ does.

2.1 Avoid Pragmas

Pragmas are commands embedded in the source files to control the compiler. Microsoft Visual C++ accepts many pragmas. Some pragmas, such as `#pragma` warning and `#pragma` pack are useful and are used quite extensively. The GNU C compiler, for the most part, does not support pragmas.

2.2 Structure Packing

The standard API for C on many processors has alignment requirements for multi-byte fields that are tighter than the ones that many protocols assume. One method that has been used to address this is to throw a command-line switch to the compiler that causes all structures to be tightly packed. If this solution is not followed, access to misaligned structure fields ranges from slow to incorrect to fatal on many of these processors.

It is not necessary that all the non-packet data structures have to be hand-aligned by insertion of padding fields. What is needed is a method to declare only those structures that represent packet data on the wire, to be tightly packed. There is no standard syntax for doing this in C. With the right incantations of macros and includes, it is possible to write C code such that per-structure packing can be done with any typical modern compiler.

```
#include <packon.h>
PACKED_PREFIX struct a {
   char a;
   long b;
} PACKED_SUFFIX;
#include <packoff.h>
```

The macros `PACKED_PREFIX` and `PACKED_SUFFIX` are the compiler-dependent prefix and postfix notation for packing a structure, if the structure supports that method. They are defined by `include` files included by `packon.h`. For those compilers that control structure packing by a `#pragma`, the appropriate `include` files with the correct `#pragma` statements are included by `packon.h` and `packoff.h` . It is important not to forget `packoff.h`. This must be done by conditional `include` files, and not macro invocation, because the C standard says that `#pragma` may not be controlled by an `#ifdef`.

2.3 Anonymous Structures and Unions

Anonymous structures and unions are embedded structures or unions inside another structure or union that does not have a name.

```
struct A {
   union {
     int x;
     int y;
   };
};
```

The union of `x` and `y` inside structure `A` is anonymous. That is, the `x` and `y` elements of an instance of this structure can be addressed directly as if they were members of `A`.

Anonymous structures and unions are a Microsoft C/C++ extension. The GNU C compiler does not support them in any form. The GNU C++ compiler does. There is no option that can be fed to the GNU C compiler to support this.

There are two options to support anonymous structures and unions:

- Compile everything in C++ mode.

Note: C files compiled in C++ will have all function names in disorder. Care must be taken to preserve the names of exported externals.

- Give the anonymous structure or union a name. This means that you can no longer address `x` and `y` in the above example as `A.x` and `A.y` , but rather have something like `A.B.x` or `A.B.y` if the union is named `B` .

This feature is actively debated on the mailing lists for the GNU C compiler. It may be supported in a future release of the compiler. gcc versions up through 2.9.x (EGCS 1.x releases) do not support this feature.

2.4 Avoid Inline Assembly

Although both the Microsoft and GNU compilers both support assembly, Microsoft uses the Intel syntax for the instructions whereas the GNU compilers use the AT&T syntax. The difference lies in the order of the parameters. With the Intel syntax, the destination operand is specified first, followed by the source operands. In the AT&T syntax, the destination operand is specified last, prefixed by the source operands. Since they look similar, it can cause confusion. In addition, the way inline assembly is specified is different. Microsoft uses the `_asm` or `__asm` keyword, followed by the instruction or instructions inside curly braces. The GNU compilers use the `ASM()` pseudo macro.

Inline assembly also complicates ports to other architectures. The GNU C/C++ compiler is used on many different architectures, and if the code compiles successfully, code can be generated for dozens of architectures. Moving to other architectures is greatly simplified by sticking to standard, portable C/C++ code.

If inline assembly must be used, it should be appropriately `#ifdef` 'd with both the target platform and compiler. A non-assembly version of the code should also be available via `#ifdef`s for platforms that do not have appropriate assembly already created.

Part 3: PIL API

3 PIL API

3.1 Critical Sections (Required)

Critical sections are a method of synchronizing access to shared resources. They differ from semaphores in that they are essentially binary: either a thread has it or it does not and will block trying to get it. There is no timeout value associated with waiting for the critical section.

Another major difference is that the user of the critical section owns the memory for the critical section. The owning process must allocate the memory by declaring a variable of type `PilCriticalSection` for each critical section and initialize them by using `PIL_InitializeCriticalSection`. Once the critical section is no longer needed, it must be discarded using `PIL_DestroyCriticalSection`.

A thread attempts to obtain a critical section using `PIL_EnterCriticalSection`. When a thread is done with the critical section, it releases the critical section by calling `PIL_LeaveCriticalSection`. `PIL_TryEnterCriticalSection` can be used to test if a critical section can be acquired, but not to block it.

Once a thread owns a critical section, subsequent calls to `PIL_EnterCriticalSection` will not block. This prevents deadlocks with a thread waiting for itself. A thread must call `PIL_LeaveCriticalSection` an equal number of times as `PIL_EnterCriticalSection` to release the critical section.

3.1.1 PIL_DestroyCriticalSection

`void PIL_DestroyCriticalSection(PilCriticalSection* pCriticalSection)`

`PIL_DestroyCriticalSection` destroys any memory associated with a critical section that has been initialized via `PIL_InitializeCriticalSection`. The behavior is undefined to destroy a critical section that has not been initialized via `PIL_InitializeCriticalSection` or has threads waiting on it.

Defined in: `PILCS.H`

Return Value

This function does not return a value.

Parameters

pCriticalSection	The critical section object to be destroyed

3.1.2 PIL_EnterCriticalSection

`void PIL_EnterCriticalSection(PilCriticalSection* pCriticalSection)`

`PIL_EnterCriticalSection` enters a critical section gaining access to the shared resource/s. If another thread has the critical section, this call will block until the critical section may be acquired. There is no timeout. The thread will wait forever until it can gain access.

The owning thread may make calls to `PIL_EnterCriticalSection` without blocking, once a critical section has successfully been obtained.

Note: If a thread makes multiple calls to PIL_EnterCriticalSection, it must call `PIL_LeaveCriticalSection` equal number of times, or the critical section will not be released.

The behavior is undefined to enter a critical section that has not been initialized using `PIL_InitializeCriticalSection`.

Defined in: `PILCS.H`

Return Value

This function does not return a value.

Parameters

`pCriticalSection`	Pointer to the critical section to be entered

3.1.3 PIL_InitializeCriticalSection

```
void PIL_InitializeCriticalSection(PilCriticalSection*
pCriticalSection)
```

`PIL_InitializeCriticalSection` initializes the memory associated with a critical section object preparing it for use. All critical sections must first be initialized with this call.

Defined in: `PILCS.H`

Return Value

This function does not return a value.

Parameters

pCriticalSection	Pointer to a chunk of memory to be initialized as a critical section

3.1.4 PIL_LeaveCriticalSection

`void PIL_LeaveCriticalSection(PilCriticalSection* pCriticalSection)`

`PIL_LeaveCriticalSection` leaves a critical section, releasing the next thread to enter the critical section if one is waiting.

The behavior is undefined if a critical section is left without first being initialized using `PIL_InitializeCriticalSection`.

Defined in: `PILCS.H`

Return Value

This function does not return a value.

Parameters

pCriticalSection	Pointer to the critical section to be left

3.1.5 PIL_TryEnterCriticalSection

`BOOL PIL_TryEnterCriticalSection(PilCriticalSection* pCriticalSection)`

`PIL_TryEnterCriticalSection` attempts to enter a critical section. The difference between this function and `PIL_EnterCriticalSection` is that this function will return `FALSE` if it cannot enter the critical section because another thread has the critical section. It will not block. It will return `TRUE` if the critical section was successfully entered. If the function returns `TRUE`, the critical section was acquired successfully and must be released using `PIL_LeaveCriticalSection` when the thread is done with it.

It is undefined to enter a critical section that has not been initialized using `PIL_EnterCriticalSection`.

Defined in: `PILCS.H`

Return Value

Returns one of the following:

`TRUE`	The critical section was successfully entered
`FALSE`	Another thread currently owns the critical section

Parameters

pCriticalSection	Pointer to the critical section to try to enter

3.2 Events (Required)

Events are a synchronization method for determining if something has happened. They maintain only a signaled or non-signaled state. Threads can wait on an event to be woken up when something happens. They are similar to binary semaphores with the additional semantic that the state, signaled or non-signaled, can be changed without blocking. The only blocking call is PIL_WaitForEvent, which waits until an event becomes signaled.

Events come in two types: manual or automatic reset.

- Manual reset events require an explicit call to set the event to non-signaled once it has become signaled.
- Automatic reset events automatically reset back to non-signaled once a thread has been woken up.

In other words, manual reset events let all threads waiting on the event through, while the automatic reset events allow only a single thread.

Events are created via PIL_CreateEvent. At creation time, the signal state can be set to what type of event it is (manual or automatic reset). Events are destroyed using PIL_DestroyEvent when they are no longer necessary. PIL_ResetEvent sets a manual reset event back to the non-signaled state. PIL_SetEvent sets either type of event to the signaled state. Finally, PIL_WaitForEvent waits until an event is signaled, with an optional timeout.

3.2.1 PIL_CreateEvent

`PilEvent PIL_CreateEvent(BOOL bManualReset, BOOL bSignaled)`

`PIL_CreateEvent` creates a new manual or automatic reset event. *`bManualReset`* determines if the event is manual or automatic reset. TRUE (non-zero) creates a manual reset event, FALSE (zero) creates an automatic reset event.

`bSignaled` determines the initial state of the event. TRUE sets the initial state to signaled. FALSE makes the event non-signaled. For most purposes, this parameter will be FALSE .

Defined in: `PILEVNT.H`

Return Value

Returns one of the following:

`PilEvent`	The handle of the newly created event
`NULL`	The event could not be created

Parameters

`bManualReset`	The type of event to be created. TRUE if the event is a manual reset or FALSE if it is automatic.
`bSignaled`	TRUE if the initial state of the event is signaled or FALSE if it is non-signaled

3.2.2 PIL_DestroyEvent

`PilResult PIL_DestroyEvent(PilEvent Event)`

`PIL_DestroyEvent` destroys an event created with `PIL_CreateEvent`. The behavior is undefined if an event is destroyed with threads waiting on it.

Defined in: `PILEVNT.H`

Return Value

Returns one of the following:

`PIL_OK`	The event was successfully destroyed
`PIL_INVALID_HANDLE`	The handle of the event is not a valid event handle

Parameters

`Event`	The handle of the event to be destroyed

3.2.3 PIL_ResetEvent

`PilResult PIL_ResetEvent(PilEvent Event)`

`PIL_ResetEvent` resets the state of a manual reset event back to non-signaled. It does nothing for an automatic reset event.

Defined in: `PILEVNT.H`

Return Value

Return one of the following:

`PIL_OK`	The event has been successfully reset to non-signaled
`PIL_INVALID_HANDLE`	The handle of the event is not a valid event handle

Parameters

`Event`	The handle of the manual reset event to be reset to non-signaled

3.2.4 PIL_SetEvent

`PilResult PIL_SetEvent(PilEvent Event)`

`PIL_SetEvent` sets the state of an event to the signaled state. For manual reset events, the event stays in the signaled state and all threads waiting or come to wait when in the signaled state will be let through until the event is explicitly reset. For automatic reset events, when the event is signaled, the first thread to wait on this event is woken, and the state of the event is automatically set back to non-signaled.

Defined in: `PILEVNT.H`

Return Value

Returns one of the following:

`PIL_OK`	The event was successfully signaled
`PIL_INVALID_HANDLE`	The handle of the event is not a valid event handle

Parameters

`Event`	The handle of the event to be set to the signaled state

3.2.5 PIL_WaitForEvent

`PilResult PIL_WaitForEvent(PilEvent` *`Event`*`, uint32_t` *`dwTimeout`*`)`

`PIL_WaitForEvent` waits for an event to become signaled. The function returns when the event is set to the signaled state or when the timeout period expires. The state of the event after a successful wait depends on the type of event. For manual reset events, the `PIL_WaitForEvent` does not affect its state. For automatic reset events, the state is automatically reset to non-signaled after the first successful wait.

The timeout specifies how long the thread should be suspended waiting for the event to be signaled in milliseconds. The value of `PIL_INFINITE_TIMEOUT` will wait forever. A value of `0` allows the state of the event to be checked without blocking.

Note: On all platforms, the granularity of the wait may be between 10 to 20 milliseconds. In such a case, the wait will be rounded up to the nearest integral unit.

Defined in: `PILEVNT.H`

Return Value

Returns one of the following:

`PIL_OK`	The wait was successful and the event was signaled
PIL_INVALID_HANDLE	The handle of the event is not a valid event handle
PIL_TIMEOUT	The timeout period expired before the event became signaled

Parameters

`Event`	The handle of the event for which to wait
`dwTimeout`	The time, in milliseconds, to wait for the event or `PIL_INFINITE_TIMEOUT` to wait forever

3.3 Heap Management (Required)

PIL's heap management allows the client to create heaps and manage them separately from the underlying operating system. An application can also create multiple heaps to better manage buffering.

For example, if an application manipulates buffers of different sizes, a heap could be created so that each heap manages buffers of only one size. PIL does not have a pre-set limit on the number of heaps.

A new heap is created using PIL_CreateHeap. At this time the total size of the heap is given. Blocks of memory can be allocated from a specific heap using PIL_HeapAlloc. Blocks are returned to the heap using PIL_HeapFree. When a heap is no longer needed, it can be freed using PIL_DestroyHeap. Finally, blocks from a heap can be shrunk using PIL_HeapShrink. The heap manager does not allow blocks to be increased in size.

The heap manager uses 8 bytes of overhead per allocation. The initial size of the heap is guaranteed to be the size given during PIL_CreateHeap. Each sub-allocation requires 8 bytes out of the heap. So if a heap is created with a size of 10,240 bytes, you cannot get 10 allocations of 1,024 bytes out of the heap, but rather 9 allocations of 1.024 bytes with 952 bytes left in the heap. Memory is allocated in a first-fit algorithm. Free blocks of memory are automatically combined during PIL_HeapFree.

3.3.1 PIL_CreateHeap

```
PilHeap PIL_CreateHeap(uint32_t MaxHeapSize)
```

PIL_CreateHeap creates a new heap with the given number of bytes. When the heap is first created, it is guaranteed to contain *MaxHeapSize* bytes, if allocated in one chunk. Each sub-allocation requires 8 bytes of overhead, which is also allocated from the heap.

The maximum size of any one heap is 2^31 or 2,147,483,648 bytes.

Defined in: PILHEAP.H

Return Value

Returns one of the following:

PilHeap	The handle of the newly created heap
NULL	The heap could not be allocated from the underlying operating system

Parameters

MaxHeapSize	The maximum size of the heap in bytes

3.3.2 PIL_DestroyHeap

```
PilResult PIL_DestroyHeap(PilHeap hHeap)
```

PIL_DestroyHeap de-allocates a heap created using PIL_CreateHeap. All blocks allocated from that heap become invalid after this call.

Defined in: PILHEAP.H

Return Value

Returns one of the following:

`PIL_OK`	The heap was successfully destroyed
`PIL_INVALID_HANDLE`	The handle of the heap is not a valid heap handle

Parameters

`hHeap`	The handle of the heap to be destroyed

3.3.3 PIL_HeapAlloc

`void* PIL_HeapAlloc(PilHeap` *`hHeap`*`, uint32_t` *`Size`*`)`

`PIL_HeapAlloc` allocates *`Size`* bytes from the given heap. The maximum size of any single allocation is the size of the heap when created.

Defined in: `PILHEAP.H`

Return Value

One of the following:

`void*`	Pointer to the newly allocated memory block
`NULL`	The heap does not contain enough free memory to allocate the block

Parameters

`hHeap`	The handle of the heap from which to allocate memory
`Size`	The size, in bytes, of the block to be allocated

3.3.4 PIL_HeapFree

`PilResult PIL_HeapFree(PilHeap` *`hHeap`*`, void*` *`pBuf`*`)`

`PIL_HeapFree` returns a block allocated using `PIL_HeapAlloc` back into the heap. When a block is freed, the heap manager performs a quick heap compaction to maximize the free space in the heap. The behavior is undefined to free a block to a heap that was not allocated from the heap.

Defined in: `PILHEAP.H`

Parameters

`hHeap`	The handle of the heap into which to the block has to be returned
`pBuf`	The pointer of the buffer to be freed back into the heap

3.3.5 PIL_HeapShrink

```
PilResult PIL_HeapShrink(PilHeap hHeap, void* pBuf, uint32_t NewSize)
```

PIL_HeapShrink allows a block allocated from a heap to be made smaller, freeing the reset of the block back into the heap. Not all blocks can be shrunk due to overhead in the heap. A block cannot be made larger using this function. The behavior is undefined if a block is shrunk in a heap from which it was not allocated.

Defined in: PILHEAP.H

Return Value

Returns one of the following:

PIL_OK	The operation was successfully completed
PIL_INVALID_HANDLE	The handle of the heap is not a valid heap handle
PIL_INVALID_PARAM	The new size is not less than the old size
PIL_ERROR	The block cannot be resized due to lack of space in the heap

Parameters

hHeap	The handle of the heap to which the block belongs
pBuf	The pointer to the buffer to be shrunk
NewSize	The new, smaller size of the block

3.4 Initialization and De-initialization (Required)

PIL has an explicit initialization and de-initialization function. These are required to be called by any application before using PIL functions and after they are finished with the PIL layer. This allows PIL to perform implementation-specific initialization and de-initialization in a portable fashion. The behavior that results from calling a PIL function without first calling PIL_Init is undefined.

3.4.1 PIL_Init

```
PilResult PIL_Init(void)
```

PIL_Init must be called before any other PIL function may be used, including any utility or run-time functions that are exported by PIL. This allows PIL to do implementation-specific initialization in a portable manner.

PIL_Init is a synchronous call. It will not return until the initialization is complete. If another call is made into PIL_Init before the first call completes, it will also block until initialization is complete. Once the initialization is complete, subsequent calls to PIL_Init will return an error code of PIL_INITIALIZED .

Defined in: PILINIT.H

Return Value

Returns one of the following:

`PIL_OK`	PIL initialization is successful
`PIL_NOMEM`	There is insufficient memory for PIL to initialize
`PIL_INITIALIZED`	`PIL_Init` has already been completed

3.4.2 PIL_DeInit

```
void PIL_DeInit(void)
```

`PIL_DeInit` must be called when PIL is no longer required. No PIL function may be called after this function is called. This allows PIL to release any implementation-specific resources in a portable manner.

`PIL_DeInit` is a synchronous call.

Defined in: `PILINIT.H`

Return Value

This function does not return a value.

3.5 Mailbox Management (Required)

The mailbox manager in PIL provides basic platform-independent messaging services. Mailboxes allow multiple threads to pass information back and forth between them. It also provides a way to synchronize messages. In other words, a thread can block waiting for messages in its mailbox. When a message arrives, it wakes up, reads the message, and does whatever action is required.

The format of messages is not defined. When the mailbox is created, the application tells PIL how big each message is and how many to store. PIL will allocate the necessary memory to maintain the queue of messages. PIL retains a copy of the data and not a pointer. Therefore, once a message has been written into the mailbox, the original does not have to be preserved. Pointers can easily be passed in this paradigm by specifying the size of the pointer for the message size.

A new mailbox is created using `PIL_CreateMailbox`. Here the size of each message and the number of messages to store is communicated. There is no inherent limit on the number of mailboxes, the size of messages, or the number of messages other than the memory required to store these structures. When a mailbox is no longer needed, it can be destroyed using `PIL_DestroyMailbox`.

`PIL_ReadMailbox` reads a message from the mailbox. If there are no messages in the mailbox, it can optionally block waiting for messages. `PIL_WriteMailbox` copies a message into the mailbox, waking any threads that may be waiting on the mailbox. `PIL_WriteMailbox` can also optionally block if the mailbox is full.

Note: If multiple threads are waiting to read a mailbox, messages will be given to threads in a first in first out order. That is, the first thread waiting will get the first message, the next thread the second, and so on. The same holds true for writing to a mailbox.

3.5.1 PIL_CreateMailbox

```
PilMailbox PIL_CreateMailbox(uint32_t MessageLength, uint32_t
QueueLen)
```

`PIL_CreateMailbox` creates a new message queue. *`QueueLength`* messages can be stored in the mailbox, each occupying *`MessageLength`* bytes. `PIL_CreateMailbox` pre-allocates all the memory necessary to store all *`QueueLength`* messages so that at run-time the mailbox either has available slots or it does not. It does not go out to the OS and try to get more.

Defined in: `PILMBOX.H`

Return Value

Returns one of the following:

`PilMailbox`	The handle of the newly created mailbox
`NULL`	The mailbox could not be created. The most common reason is that there is not enough memory to allocate the mailbox data structures.

Parameters

`MessageLength`	The maximum number of messages that can be stored in the mailbox
`QueueLen`	The size of each message, in bytes, stored in each mailbox slot

3.5.2 PIL_DestroyMailbox

```
PilResult PIL_DestroyMailbox(PilMailbox hMailbox)
```

`PIL_DestroyMailbox` destroys a mailbox created with `PIL_CreateMailbox`, releasing all memory associated with the mailbox. The behavior is undefined if a mailbox is destroyed that has readers or writers blocked on it. All messages in the queue are lost when the mailbox is destroyed, and if the destroyed message has pointers to any allocated resources, those resources will not be freed.

Defined in: `PILMBOX.H`

Return Value

Returns one of the following:

`PIL_OK`	The operation was successfully completed
`PIL_INVALID_HANDLE`	The handle of the mailbox is not a valid mailbox handle

Parameters

`hMailbox`	The handle of the mailbox to be destroyed

3.5.3 PIL_ReadMailbox

```
PilResult PIL_ReadMailbox(PilMailbox hMailbox, void* pmbmMsg,
uint32_t Timeout)
```

`PIL_ReadMailbox` reads the next message from the mailbox, copying the message into the buffer provided by the caller in *`pMessage`*. If a message is not available, the call will be blocked for a period specified by *Timeout*.

This function may be used in non-blocking contexts, such as timer and APC callbacks, if the timeout is zero.

Defined in: `PILMBOX.H`

Return Value

Returns one of the following:

`PIL_OK`	A message was successfully read and copied into the given buffer
`PIL_INVALID_HANDLE`	The handle of the mailbox is not a valid mailbox handle
`PIL_INVALID_PARAM`	pMessage is not a valid pointer
`PIL_TIMEOUT`	The timeout period expired before a message was available

Parameters

`hMailbox`	The handle of the mailbox from which a message has to be read
`pmbmMsg`	A pointer to a buffer where the message contents will be copied
`Timeout`	The time, in milliseconds, to wait for the message or `PIL_INFINITE_TIMEOUT` to wait forever

3.5.4 PIL_WriteMailbox

```
PilResult PIL_WriteMailbox(PilMailbox hMailbox, void* pmbmMsg,
uint32_t Timeout)
```

`PIL_WriteMailbox` copies the message pointed to by *`pMessage`* into the next available slot in the mailbox queue. If no slot is available to write the message, `PIL_WriteMailbox` will block for *`Timeout`* milliseconds.

The function may be used in non-blocking contexts, such as timer and APC callbacks, if the timeout is zero.

Defined in: `PILMBOX.H`

Return Value

Returns one of the following:

PIL_OK	The message was successfully written into the mailbox
PIL_INVALID_HANDLE	The handle of the mailbox is not a valid mailbox handle
PIL_INVALID_PARAM	*pMessage* is not a valid pointer
PIL_TIMEOUT	The timeout period expired before a slot was available

Parameters

hMailbox	The handle of the mailbox in which a message has to be written
pmbmMsg	A pointer to a buffer from which the message contents are copied
Timeout	The time, in milliseconds, to wait for a free slot, or PIL_INFINITE_TIMEOUT to wait forever

3.6 Memory (Required)

Memory routines allow a client to acquire or return memory to the underlying operating system. Memory can be allocated by specifying the number of bytes using PIL_AllocateMemory or a specified number of pages using PIL_AllocatePages. Both routines allow a flag to specify if the memory should be initialized to zero, and optionally with pages, to lock the pages in memory. The inverse operation is PIL_FreeMemory and PIL_FreePages. These two functions cannot be intermixed, that is, if the memory is allocated using PIL_AllocateMemory, it needs to be freed with PIL_FreeMemory.

PIL also provides a mechanism to get the page size from the underlying environment in a portable manner via the PIL_GetPageSize function.

3.6.1 PIL_AllocateMemory

```
void* PIL_AllocateMemory(uint32_t Size, uint32_t Flags)
```

PIL_AllocateMemory allocates a new block of memory from the underlying operating system. The memory block can be optionally initialized to zero by specifying the PIL_MEM_ZERO_INIT flag. There is no special alignment on the block of memory.

Defined in: PILMEM.H

Return Value

Returns one of the following:

`void*`	Pointer to the newly allocated block of memory
`NULL`	The memory could not be allocated

Parameters

`Size`	The size of the block in bytes
`Flags`	Special options for the allocation. Currently only `PIL_MEM_ZERO_INIT` is supported.

3.6.2 PIL_AllocatePages

`void* PIL_AllocatePages(uint32_t` *`NumPages`*`, uint32_t` *`Flags`*`, uint32_t*` *`pPhysAddr`*`)`

`PIL_AllocatePages` allocates one or more contiguous pages. *`pPhysAddr`*, when specified, will contain the physical address of the pages on platforms where it is supported. Otherwise, it contains the same value as the linear address.

All platforms may not support the `PIL_MEM_PAGE_LOCK` flag. The allocation will fail if this flag is specified and page locked memory is not supported.

Defined in: `PILMEM.H`

Return Value

Returns one of the following:

`void*`	The linear address of the newly allocated block
`NULL`	The pages could not be allocated or could not be locked

Parameters

`NumPages`	The number of pages to allocate. This must be at least 1.
`Flags`	Optional flags for allocation. `PIL_MEM_ZERO_INIT` initializes all the pages to 0. `PIL_MEM_PAGE_LOCK` locks the pages in memory so they are not swapped out to disk.
`pPhysAddr`	Optional pointer to store the physical address of the newly allocated pages

3.6.3 PIL_FreeMemory

```
void PIL_FreeMemory(void* pMem)
```

`PIL_FreeMemory` frees allocated memory using `PIL_AllocateMemory`. Freeing memory via `PIL_FreeMemory` that was not allocated using `PIL_AllocateMemory` is undefined, and may cause the system to crash.

Defined in: `PILMEM.H`

Return Value

This function does not return a value.

Parameters

pMem	Pointer to the linear address of the memory to be freed

3.6.4 PIL_FreePages

```
void PIL_FreePages(void* pMem, uint32_t NumPages)
```

`PIL_FreePages` frees allocated pages via `PIL_AllocatePages`. The number of pages specified to `PIL_FreePages` must match those specified to `PIL_AllocatePages` or problems may result. Freeing memory via `PIL_FreePages` that was not allocated using `PIL_AllocatePages` is undefined and may cause the system to crash.

Defined in: `PILMEM.H`

Return Value

This function does not return a value.

Parameters

pMem	The linear address of the pages
NumPages	The number of pages to be freed

3.6.5 PIL_GetPageSize

```
uint32_t PIL_GetPageSize(void)
```

`PIL_GetPageSize` returns the size of a page in bytes on the native operating environment. This page size value can then be used to calculate the number of pages necessary with `PIL_AllocatePages`.

Defined in: `PILMEM.H`

Return Value

The size of the page in bytes.

3.7 Semaphores (Required)

Semaphores are the basic synchronization mechanism in PIL. All PIL semaphores are counting semaphores, but a macro is provided to make a binary semaphore. Semaphores are first-come-first-serve, that is, the priority of the thread waiting on the semaphore does not matter.

A semaphore is given both an initial count and a maximum count when created. The maximum count is the total number of resources the semaphore protects. For simple synchronization, the count is 1. The initial count is the number of free resources upon creation. Most of the time, this is 0, so that all threads initially block on the semaphore, or the initial count is the same as the maximum count.

Note: Specifying a value between the initial and maximum count may not work on all platforms.

Semaphores are created with `PIL_CreateSemaphore`. This call will return a handle to the newly created semaphore when successful. `PIL_AcquireSemaphore` will wait on the semaphore. In other words, `PIL_AcquireSemaphore` reduces the count of the semaphore by 1. The thread will block if the count is 0. The timeout parameter to `PIL_AcquireSemaphore` determines how long to wait.

`PIL_ReleaseSemaphore` increments the count of the semaphore by 1, signaling a thread waiting on the semaphore. `PIL_ReleaseSemaphore` cannot block, but a context switch may occur if the signaled thread is of higher priority.

All semaphores need to be destroyed using `PIL_DestroySemaphore` when they are no longer needed.

Note: Destroying a semaphore used by another thread is undefined, and will probably cause the machine to crash.

3.7.1 PIL_AcquireSemaphore

`PilResult PIL_AcquireSemaphore(PilSemaphore` *`Sem`*`, uint32_t` *`Timeout`*`)`

`PIL_AcquireSemaphore` attempts to grab the semaphore, possibly blocking for a period of time. It effectively decrements the semaphore's count by one.

A timeout value of zero specifies that the acquire should not block. If the semaphore is free, `PIL_OK` is returned and the semaphore is acquired. If the semaphore is not free, `PIL_AcquireSemaphore` will return `PIL_TIMEOUT` to signal that the semaphore is not free and that the zero-length timeout expired.

Note: On all platforms, the timeout granularity may not be one millisecond but may be something larger such as 10 or 20 milliseconds. Values will be rounded up to the nearest integral value.

Defined in: `PILSEM.H`

Return Value

Returns one of the following:

`PIL_OK`	The semaphore was acquired successfully
`PIL_TIMEOUT`	The timeout period expired before the semaphore could be acquired

Parameters

Sem	The handle of the semaphore to be acquired
Timeout	The amount of time, in milliseconds, to wait for the semaphore; or PIL_INFINITE_TIMEOUT to wait forever

3.7.2 PIL_CreateBinarySemaphore

```
define PIL_CreateBinarySemaphore(BOOL bInitialCount)
```

PIL_CreateBinarySemaphore is equivalent to PIL_CreateSemaphore(bInitalCount, 1). Please refer to PIL_CreateSemaphore for more details.

Defined in: PILSEM.H

Return Value

Returns one of the following:

PilSemaphore	The handle of the newly created semaphore object
NULL	The semaphore could not be created

Parameters

bInitialCount	TRUE (non-zero) if the semaphore is initially free, FALSE (zero) if it is not

3.7.3 PIL_CreateSemaphore

```
PilSemaphore PIL_CreateSemaphore(uint32_t InitialCount, uint32_t MaximumCount)
```

PIL_CreateSemaphore creates a new semaphore object and returns the handle of the new object. The function returns NULL if the object cannot be created, most likely because of memory constraints.

The *InitialCount* parameter determines the starting value of the counting semaphore. For most purposes, this will either be 0 i.e., there are no free resources, or the same as the *MaximumCount* parameter i.e., all resources free. The *InitialCount* must be less than or equal to the *MaximumCount*.

The *MaximumCount* parameter determines the maximum count of the semaphore. It must be at least 1 and greater than or equal to *InitialCount*.

Defined in: PILSEM.H

Return Value

Returns one of the following:

`PilSemaphore`	The handle of the newly created semaphore object
`NULL`	The semaphore could not be created

Parameters

`InitialCount`	The count the semaphore should have when it is created
`MaximumCount`	The maximum count the semaphore can have

3.7.4 PIL_DestroySemaphore

`PilResult PIL_DestroySemaphore(PilSemaphore` *`Sem`*`)`

`PIL_DestroySemaphore` destroys a semaphore created with `PIL_CreateSemaphore`. All semaphores must be destroyed using this function for the memory to be reclaimed. Semaphores should not be destroyed while they are still being used.

Defined in: `PILSEM.H`

Return Value

Returns one of the following:

`PIL_OK`	The semaphore has been successfully destroyed
`PIL_INVALID_HANDLE`	The handle of the semaphore is not a valid semaphore handle

Parameters

`Sem`	The handle of the semaphore to be destroyed

3.7.5 PIL_GetSemaphoreCount

`PilResult PIL_GetSemaphoreCount(PilSemaphore` *`Sem`*`, uint32_t*` *`pValue`*`)`

`PIL_GetSemaphoreCount` returns the current count of the semaphore between the initial and maximum values that it was created with.

Defined in: `PILSEM.H`

Return Value

Returns one of the following:

`PIL_OK`	The operation was successfully completed

PIL_INVALID_HANDLE	The semaphore handle is not a valid handle for a semaphore
PIL_INVALID_PARAM	The pointer to the value is not a valid pointer

Parameters

Sem	The handle of the semaphore for which the count has to be received
pValue	Pointer to a variable in which the semaphore value has to be placed

3.7.6 PIL_ReleaseSemaphore

`PilResult PIL_ReleaseSemaphore(PilSemaphore Sem)`

`PIL_ReleaseSemaphore` releases a semaphore, essentially incrementing the count of the semaphore. A semaphore does not have to be first acquired before it can be released. It is not an error to release a semaphore that is already at the maximum count; it will be ignored.

Defined in: `PILSEM.H`

Return Value

Returns one of the following:

PIL_OK	The semaphore was successfully released or incremented
PIL_INVALID_HANDLE	The handle of the semaphore is not a valid semaphore handle

Parameters

Sem	The handle of the semaphore to be released

3.8 Threading (Required)

A thread, also known as a task, is the basic unit of execution. It contains a copy of the CPU execution state for itself. It shares an address with other threads created under a process, or with all other threads if it runs in the kernel.

All threads are created using the PIL_CreateThread API. Certain attributes of the thread can be specified when it is created such as its priority, stack size, a name, and the function that represents the thread's starting address. Once created, the thread starts executing independently unless the PIL_THREAD_SUSPENDED flag is used when the thread is created, which delays the thread from executing until a PIL_ResumeThread call is made on the thread.

Certain thread functions operate on a thread by another thread, and some are for use by the thread itself. Examples of the former are PIL_DestroyThread, PIL_GetThreadPriority, PIL_ResumeThread, PIL_SuspendThread, and PIL_TerminateThread. All these functions require the handle of the thread. A thread may obtain its own handle by calling PIL_GetCurrentThread. Functions used by the thread include PIL_DelayThread and PIL_ExitThread.

All threads must call PIL_ExitThread before their thread function returns. Failure to do so may cause unpredicted results. The creator of the thread needs to call PIL_DestroyThread to free up the data structures associated with the thread, after a thread has successfully exited. PIL_DestroyThread returns the exit code for the thread.

A thread can forcibly terminate another thread by calling PIL_TerminateThread.

Note: On all platforms, the thread data structures may not be completely cleaned up when calling this function. PIL_DestroyThread is still required to release the memory associated with the thread.

PIL allows a single, 32-bit value to be associated with each thread handle. The value may be set using the PIL_SetThreadValue function and retrieved via the PIL_GetThreadValue function.

In some cases, the priority level mapping that the PIL does may not be sufficient. PIL provides a mechanism to retrieve and set the native priority level of a thread without any interpretation. These native priority values are operating system-specific. PIL_GetNativeThreadPriority retrieves the native priority from the underlying operating system. Inversely, PIL_SetNativeThreadPriority allows the thread priority to be set without any priority mapping by PIL.

PIL_WaitForThread allows the thread to wait until another thread has exited. The waiting thread has the option to check if the other thread has exited, wait for a short while for it to exit, or wait forever until the thread exits.

3.8.1 PIL_CreateThread

```
PilThread PIL_CreateThread(PilThreadRoutine StartAddress, uint32_t
StackSize, char* Name, uint32_t Priority, void* Param, uint32_t
Options)
```

`PIL_CreateThread` creates a thread and optionally starts executing it. The *StartingAddress* parameter is the address (i.e., a function) where the thread will begin executing. The specified function must have the following prototype:

```
uint32_t PilThreadRoutine ( void* Param );
```

Note: Thread functions are assumed that they use standard C calling convention (`cdecl`).

The *`Param`* specified in the `PIL_CreateThread` call will be passed to the thread function. The contents of the parameter are not specified and they are completely up to the caller.

`dwStackSize` is the size of the stack the thread should have, in bytes.

Note: Not all platforms allow threads to have a specified stack size. For portability, a stack size should be specified. To conserve memory, the stack should not be very large.

`Name` is a string that describes the thread. It is optional, and can help in resource tracking if one is specified.

The priority of the thread can range from 1 (lowest) to 8 (highest).

`PIL_CreateThread` currently has only one option for *`dwOptions`*: `PIL_THREAD_SUSPENDED` . When this option is specified, the thread will not begin executing immediately upon creation. It has to be started using the PIL_ResumeThread API.

Defined in: PILTHD.H

Return Value

Returns one of the following:

`PilThread`	The handle of the newly created thread
`NULL`	The thread could not be created

Parameters

`StartAddress`	The starting address where the thread will begin execution
`StackSize`	The size of the thread's stack in bytes
`Name`	The optional name of the thread
`Priority`	The priority at which the thread will be executed
`Param`	An optional parameter to pass to the thread function
`Options`	Thread creation options

3.8.2 PIL_DelayThread

`PilResult PIL_DelayThread(uint32_t` *`Delay`*`)`

`PIL_DelayThread` delays the execution of the currently executing thread for a specified number of milliseconds. The thread cannot be alerted during this wait, so it cannot be signaled.

Note: On all platforms, the resolution of the wait may not be 1 millisecond but could be 10 or 20 milliseconds. In such cases, the delay will be rounded up to the nearest supported interval.

Defined in: `PILTHD.H`

Return Value

Returns one of the following:

`PIL_OK`	The delay was completed successfully

Parameters

`Delay`	The number of milliseconds to delay the currently executing thread

3.8.3 PIL_DestroyThread

`PilResult PIL_DestroyThread(PilThread` *`Thread`*`, uint32_t*` *`pStatus`*`)`

`PIL_DestroyThread` de-allocates all memory associated with a thread.

Note: This assumes that the thread is already terminated. After this call, the thread handle is invalid and cannot be used, even by the thread itself. All threads must be destroyed using the function after they terminate, to free associated memory.

pStatus is an optional parameter that allows the caller to retrieve the exit status of the thread when it is terminated. The exit status is the value passed to PIL_ExitThread when the thread voluntarily exits. This parameter may be NULL if the status is not required.

Defined in: PILTHD.H

Return Value

Returns one of the following:

PIL_OK	The thread was destroyed successfully
PIL_INVALID_HANDLE	The handle of the thread is not a valid thread handle

Parameters

Thread	The handle of the thread to be destroyed
pStatus	An optional pointer to a variable to retrieve the exit status of the thread

3.8.4 PIL_ExitThread

```
void PIL_ExitThread(uint8_t Status)
```

PIL_ExitThread stops execution of the current thread.

Note: PIL_ExitThread does not destroy the thread, but simply stops execution. PIL_DestroyThread must be called to clean up after the thread stops execution. This function never returns.

The thread may specify a reason code as to why it exits. If there is no reason, it should be set to 0.

Defined in: PILTHD.H

Return Value

This function does not return.

Parameters

Status	The reason code the thread has for exiting

3.8.5 PIL_GetCurrentThread

```
PilThread PIL_GetCurrentThread(void)
```

PIL_GetCurrentThread returns the handle of the currently executing thread for use in other thread functions. This function cannot fail and will always return a valid handle.

Note: PIL_GetCurrentThread may not be lightweight and may take longer as the number of threads increase.

Defined in: PILTHD.H

Return Value

The handle of the currently executing thread

3.8.6 PIL_GetNativeThreadPriority

`PilResult PIL_GetNativeThreadPriority(PilThread` *`Thread`*`, uint32_t*` *`pPri`*`)`

`PIL_GetNativeThreadPriority` retrieves the current priority of the given thread as maintained by the underlying operating system. The values returned by this function are operating system specific and are useful only when priorities outside what PIL offers are required.

Defined in: PILTHD.H

Return Value

Returns one of the following:

`PIL_OK`	The priority was retrieved successfully
`PIL_INVALID_HANDLE`	The handle of the thread is not a valid thread handle
`PIL_INVALID_PARAM`	The pointer to the priority value is not a valid pointer

Parameters

`Thread`	The handle of the thread for which the priority has to be obtained
`pPri`	A pointer to a variable in which the priority value has to be placed

3.8.7 PIL_GetThreadPriority

`PilResult PIL_GetThreadPriority(PilThread` *`Thread`*`, uint32_t*` *`pPri`*`)`

`PIL_GetThreadPriority` retrieves the current priority of the given thread.

Defined in: `PILTHD.H`

Return Value

Returns one of the following:

`PIL_OK`	The priority was retrieved successfully
`PIL_INVALID_HANDLE`	The handle of the thread is not a valid thread handle
`PIL_INVALID_PARAM`	The pointer to the priority value is not a valid pointer

Parameters

Thread	The handle of the thread for which the priority has to be obtained
pPri	A pointer to a variable in which the priority value has to be placed

3.8.8 PIL_GetThreadValue

`PilResult PIL_GetThreadValue(PilThread Thread, uint32_t* pValue)`

`PIL_GetThreadValue` retrieves the 32-bit value associated with the given thread. The function will return 0 if no value has been previously set.

Defined in: `PILTHD.H`

Return Value

Returns one of the following:

`PIL_OK`	The value was retrieved successfully
`PIL_INVALID_HANDLE`	The handle of the thread is not a valid thread handle
`PIL_INVALID_PARAM`	The pointer to the value is not a valid pointer

Parameters

Thread	The handle of the thread for which the value has to be obtained
pValue	Pointer to a location where the value has to be stored

3.8.9 PIL_ResumeThread

`PilResult PIL_ResumeThread(PilThread Thread)`

`PIL_ResumeThread` resumes the execution of a thread suspended at creation time via the `PIL_THREAD_SUSPENDED` flag or via `PIL_SuspendThread`. Execution continues at the same priority level the thread was at when it was suspended.

Resuming a thread that is not suspended has no effect.

Defined in: `PILTHD.H`

Return Value

Returns one of the following:

`PIL_OK`	The thread was resumed successfully
`PIL_INVALID_HANDLE`	The handle of the thread is not a valid thread handle
`PIL_ERROR`	The thread could not be resumed. This is probably a result of the thread being in a state where it cannot be resumed due to an operating system dependent situation.

Parameters

`Thread`	The handle of the thread to be resumed

3.8.10 PIL_SetNativeThreadPriority

`PilResult PIL_SetNativeThreadPriority(PilThread` *`Thread`*`, uint32_t` *`Priority`*`)`

`PIL_SetNativeThreadPriority` sets the priority of a thread by using values native to the underlying operating system.

Note: Values used by this function are operating system specific, and are useful for setting priorities outside of what PIL offers.

Defined in: `PILTHD.H`

Return Value

Returns one of the following:

`PIL_OK`	The thread's priority was adjusted successfully
`PIL_INVALID_HANDLE`	The handle of the thread is not a valid thread handle
`PIL_INVALID_PARAM`	The new priority value is not a valid priority

Parameters

`Thread`	The handle of the thread for which the priority has to be adjusted
`Priority`	The new priority of the thread

3.8.11 PIL_SetThreadPriority

`PilResult PIL_SetThreadPriority(PilThread` *`Thread`*`, uint32_t` *`Priority`*`)`

`PIL_SetThreadPriority` sets the priority of a thread. Threads may adjust their own priority by specifying `PIL_GetCurrentThread` as the first parameter. Though this call will not block, a task

switch may occur if the priority of the thread is adjusted above the priority of the currently executing thread.

Defined in: PILTHD.H

Return Value

Returns one of the following:

PIL_OK	The thread's priority was adjusted successfully
PIL_INVALID_HANDLE	The handle of the thread is not a valid thread handle
PIL_INVALID_PARAM	The new priority value is not a valid priority

Parameters

Thread	The handle of the thread for which the priority has to be adjusted
Priority	The new priority of the thread

3.8.12 PIL_SetThreadValue

PilResult PIL_SetThreadValue(PilThread *Thread*, uint32_t *Value*)

PIL_SetThreadValue allows a single 32-bit value to be associated with a particular thread handle. Only one 32-bit value is provided for each thread handle and this is shared by all. No interpretation of this value is done by PIL.

Defined in: PILTHD.H

Return Value

Returns one of the following:

PIL_OK	The thread's priority was adjusted successfully
PIL_INVALID_HANDLE	The handle of the thread is not a valid thread handle

Parameters

Thread	The handle of the thread for which the value has to be set
Value	The 32-bit value to be stored with this thread

3.8.13 PIL_SuspendThread

PilResult PIL_SuspendThread(PilThread *Thread*)

PIL_SuspendThread causes the given thread to stop executing. This is different from adjusting the thread priority to minimum. Once this call returns, the thread stops executing and will not continue until a PIL_ResumeThread call is made.

Suspending a thread that is currently blocked with a timeout is undefined.

Suspending a thread that is already suspended has no effect.

Defined in: PILTHD.H

Return Value

Returns one of the following:

PIL_OK	The thread was suspended successfully
PIL_INVALID_HANDLE	The handle of the thread is not a valid thread handle
PIL_ERROR	The thread could not be suspended. This is probably a result of the thread being in a state where it cannot be suspended due to an operating system dependent situation.

Parameters

Thread	The handle of the thread to be suspended

3.8.14 PIL_TerminateThread

```
PilResult PIL_TerminateThread(PilThread Thread)
```

PIL_TerminateThread forcibly terminates a thread, not giving it a chance to do any cleanup. This should be used only in abnormal circumstances.

Note: The thread must still be destroyed using PIL_DestroyThread.

Defined in: PILTHD.H

Return Value

Returns one of the following:

PIL_OK	The thread was terminated successfully
PIL_INVALID_HANDLE	The handle of the thread is not a valid thread handle

Parameters

Thread	The handle of the thread to be terminated

3.8.15 PIL_WaitForThread

```
PilResult PIL_WaitForThread(PilThread Thread, uint32_t dwTimeout)
```

PIL_WaitForThread will block until the given thread exits. The timeout value specifies the time period for the thread to exit. A value of PIL_INFINITE_TIMEOUT will wait forever. A timeout value of 0 will check if the thread has exited.

Note: On all platforms, the granularity of the wait may not be one millisecond but may be more like 10 or 20 milliseconds. In such a case, the wait will be rounded up to the nearest integral unit.

Defined in: PILTHD.H

Return Value

Returns one of the following:

`PIL_OK`	The wait was successful and the thread has exited
`PIL_INVALID_HANDLE`	The handle of the event is not a valid event handle
`PIL_TIMEOUT`	The timeout period expired before the event became signaled

Parameters

`Thread`	The handle of the thread for which to wait
`dwTimeout`	The time, in milliseconds, to wait for the thread or `PIL_INFINITE_TIMEOUT` to wait forever

3.9 Timers (Required)

Timers provide a mechanism to generate a callback after a certain time. They can also be set to provide a callback on a periodic basis.

Timers are created using `PIL_CreateTimer`, which specifies the type: single or repeating. `PIL_CreateTimer` also specifies the initial interval of the timer. A timer is scheduled to fire using `PIL_StartTimer`. A timer can be removed from the list of active timers using `PIL_StopTimer`. `PIL_ResetTimer` allows the timer to be reconfigured with a different callback function or interval. All timers need to be freed using `PIL_DestroyTimer`.

Due to platform differences in the execution of the timer callbacks, it cannot be assumed that the timer runs in any specific context. Therefore, many operations are disallowed during the execution of the timer routine. They include:

- Any PIL function that creates or destroys an object
- Memory allocation functions
- Blocking synchronization functions
- Read/Write mailbox functions with a non-zero timeout

3.9.1 PIL_CreateTimer

```
PilTimer PIL_CreateTimer(PilTimerRoutine TimerRoutine, uint32_t
Interval, void* Context, PilTimerType Type)
```

`PIL_CreateTimer` creates a new timer object. It does not automatically schedule the timer to fire. `PIL_StartTimer` starts a newly created timer.

The timer routine must have the following prototype:

```
void TimerRoutine( PilTimer Timer, void* Context );
```

The handle of the timer that just expired is passed to the timer routine along with the user defined context pass from `PIL_CreateTimer` or `PIL_ResetTimer`.

Interval specifies the amount of time before the timer expires. For repeating timers, it is the period during which it fires.

Note: Not all platforms have a resolution of one millisecond. The resolution is more like 10 or 20 milliseconds. On those platforms, the interval will be rounded up to the next integral unit.

Defined in: PILTMR.H

Return Value

Returns of the following:

PilTimer	The handle of the newly created timer
NULL	The timer could not be created

Parameters

TimerRoutine	The function to call when the timer expires
Interval	The amount of time to wait, in milliseconds, until the timer fires
Context	The context information to pass to the timer routine
Type	PIL_TIMER_SINGLE or PIL_TIMER_REPEATING

3.9.2 PIL_DestroyTimer

```
PilResult PIL_DestroyTimer(PilTimer Timer)
```

PIL_DestroyTimer destroys a timer object created via PIL_CreateTimer. The timer must first be stopped by waiting for it to expire or by calling PIL_StopTimer. The behavior of destroying a timer that is still scheduled is undefined.

Defined in: PILTMR.H

Return Value

Returns one of the following:

PIL_OK	The timer was destroyed successfully
PIL_INVALID_HANDLE	The handle of the timer is not a valid timer handle

Parameters

Timer	The handle of the timer to destroy

3.9.3 PIL_DisableTimers

```
PilResult PIL_DisableTimers(void)
```

`PIL_DisableTimers` globally disables timer callbacks until `PIL_EnableTimers` is called. After this call, no timer callback is called. Care should be taken to avoid long periods of time when timers are disabled so that the timeout values are not skewed.

Defined in: `PILTMR.H`

Return Value

Returns one of the following:

`PIL_OK`	The timers have been disabled successfully
`PIL_ERROR`	The timers are already disabled

3.9.4 PIL_EnableTimers

```
PilResult PIL_EnableTimers(void)
```

`PIL_EnableTimers` re-enables timer callbacks that have been disabled via `PIL_DisableTimers`.

Note: Timers that expired during the period when timers were disabled will most likely fire immediately.

Defined in: `PILTMR.H`

Return Value

Returns one of the following:

`PIL_OK`	The timers have been enabled successfully
`PIL_ERROR`	The timers are already enabled

3.9.5 PIL_ResetTimer

```
PilResult PIL_ResetTimer(PilTimer Timer, PilTimerRoutine
TimerRoutine, uint32_t Interval, void* Context, PilTimerType Type)
```

`PIL_ResetTimer` allows an already created timer to be re-configured. The prototype is very similar to `PIL_CreateTimer`. Refer to `PIL_CreateTimer` for descriptions of the parameters.

Defined in: `PILTMR.H`

Return Value

Returns one of the following:

`PIL_OK`	The timer was successfully reset
`PIL_INVALID_HANDLE`	The handle of the timer is not a valid timer handle
`PIL_INVALID_PARAM`	A parameter is not valid

Parameters

`Timer`	The handle of the timer to reset
`TimerRoutine`	The new routine to call when the timer fires
`Interval`	The new interval to wait before calling the timer routine
`Context`	The new context to pass to the timer routine
`Type`	`PIL_TIMER_SINGLE` or `PIL_TIMER_REPEATING`

3.9.6 PIL_StartTimer

`PilResult PIL_StartTimer(PilTimer` *`Timer`*`)`

`PIL_StartTimer` schedules a timer according to the parameters specified when it was created or when reset with `PIL_ResetTimer`. A repeating timer cannot be scheduled more than once unless first stopped. A single shot timer cannot be scheduled more than once unless it is stopped or has fired.

Defined in: `PILTMR.H`

Return Value

Returns one of the following:

`PIL_OK`	The timer was successfully scheduled
PIL_INVALID_HANDLE	The handle of the timer is not a valid timer handle
`PIL_ERROR`	The timer is already scheduled

Parameters

`Timer`	The handle of the timer to be scheduled

3.9.7 PIL_StopTimer

`PilResult PIL_StopTimer(PilTimer` *`Timer`*`)`

`PIL_StopTimer` stops a timer that has been scheduled. Repeating timers must be stopped with this function before they can be destroyed.

Defined in: PILTMR.H

Return Value

Returns one of the following:

PIL_OK	The timer was successfully stopped
PIL_INVALID_HANDLE	The handle of the timer is not a valid timer handle
PIL_ERROR	The timer is not scheduled

Parameters

Timer	The handle of the timer to be stopped

3.10 Dynamic Libraries (Optional)

Dynamic libraries allow modules to be loaded and symbols to be resolved at run-time. For environments where the underlying OS supports dynamic libraries, PIL utilizes those services. In other environments, PIL has to emulate the dynamic loading mechanism. The actual implementation of this emulation is platform specific. Refer the platform specific notes for each implementation to see if dynamic libraries are native or emulated.

3.10.1 PILOS_FreeLibrary

```
PilResult PILOS_FreeLibrary(PilLib Lib)
```

PILOS_FreeLibrary frees a library loaded via PILOS_LoadLibrary.

Defined in: PILLIB.H

Return Value

Returns one of the following:

PIL_OK	The library was successfully freed
PIL_INVALID_HANDLE	The handle of the library is not a valid library handle

Parameters

Lib	The handle of the library to be freed

3.10.2 PILOS_GetProcAddress

```
PilProc PILOS_GetProcAddress(PilLib Lib, char* ProcName)
```

PILOS_GetProcAddress looks up a function in a dynamic library loaded via PILOS_LoadLibrary. The address of the function is returned and it should be cast to the appropriate function prototype. The PilProc type is defined as:

```
typedef int32_t (*PilProc)();
```

Defined in: PILLIB.H

Return Value

Returns one of the following:

`PilProc`	The address of the symbol
`NULL`	The symbol could not be found in this dynamic library

Parameters

`Lib`	The library in which a function has to be looked up
`ProcName`	The name of the function to be retrieved

3.10.3 PILOS_LoadLibrary

`PilLib PILOS_LoadLibrary(char* LibraryName)`

`PILOS_LoadLibrary` loads a dynamic library module into memory, returning the handle of the library. Function points can be retrieved from this module by using `PILOS_GetProcAddress`.

Defined in: `PILLIB`.H

Return Value

Returns one of the following:

`PilLib`	The handle of the newly loaded library
`NULL`	The library could not be loaded

Parameters

`LibraryName`	The name of the library to be loaded

3.11 Input/Output Register Access (Optional)

The Input/output register access service is mainly for drivers. It allows a driver to access the hardware ports in a portable manner. It is not entirely portable since not all architectures use port IO. However, large sections of driver code can be ported between environments easily when using these functions rather than the native OS.

Support is broken into two pieces:

The first lets a driver find its associated hardware on the PCI bus by searching by either class or by vendor/device ID. PIL provides functions to access the PCI configuration space for a device, once the device is found.

The other type of access is raw port IO. This requires the port number to be known. It does not require searching for a device on the PCI bus.

3.11.1 PILOS_FindPciDeviceSignature

`PilPciDevice PILOS_FindPciDeviceSignature(uint16_t Vendor, uint16_t Id, PilPciDevice Last)`

`PILOS_FindPciDeviceSignature` locates a specific PCI device by its signature (vendor id/product id).

Defined in: `PILIO.H`

Return Value

Returns one of the following:

`PilPciDevice`	The handle of the PCI device
`NULL`	The specific PCI device could not be found

Parameters

`Vendor`	The vendor number to match from the device configuration space
`Id`	The device ID number to be matched from the device configuration space
`Last`	The value returned from a previous call to this function from which search has to be to continued

3.11.2 PILOS_FindPciDeviceClass

`PilPciDevice PILOS_FindPciDeviceClass(uint16_t Class, PilPciDevice Last)`

`PILOS_FindPciDeviceClass` locates a PCI device based on its class.

Defined in: `PILIO.H`

Return Value

Returns one of the following:

`PilPciDevice`	The handle of the PCI device
`NULL`	The specific PCI device could not be found

Parameters

`Class`	The class code to be matched from the device configuration space
`Last`	The value returned from a previous call to this function from which search has to be continued

3.11.3 PILOS_ReadPortByte

`uint8_t PILOS_ReadPortByte(uint16_t wPort)`

`PILOS_ReadPortByte` reads a single byte from *wPort*.

Defined in: `PILIO.H`

Return Value

The byte value read from the port.

Parameters

wPort	The port number from which a byte has to be read

3.11.4 PILOS_ReadPortWord

`uint16_t PILOS_ReadPortWord(uint16_t wPort)`

`PILOS_ReadPortWord` reads a single word from *wPort*.

Defined in: PILIO.H

Return Value

The word value read from the port.

Parameters

wPort	The port number from which a word has to be read

3.11.5 PILOS_ReadPortDword

`uint32_t PILOS_ReadPortDword(uint16_t wPort)`

`PILOS_ReadPortDword` reads a single `dword` from *wPort*.

Defined in: PILIO.H

Return Value

The `dword` value read from the port.

Parameters

wPort	The port number from which a `dword` has to be read

3.11.6 PILOS_WritePortByte

`void PILOS_WritePortByte(uint16_t wPort, uint8_t Value)`

`PILOS_WritePortByte` writes a single byte to *`wPort`*.

Defined in: `PILIO.H`

Return Value

This function does not return a value.

Parameters

`wPort`	The port to which the value has to be written
`Value`	The value to write to *`wPort`*

3.11.7 PILOS_WritePortWord

`void PILOS_WritePortWord(uint16_t wPort, uint16_t Value)`

`PILOS_WritePortWord` writes a single word to *`wPort`*.

Defined in: `PILIO.H`

Return Value

This function does not return a value.

Parameters

`wPort`	The port to which the value has to be written
`Value`	The value to write to *`wPort`*

3.11.8 PILOS_WritePortDword

`void PILOS_WritePortDword(uint16_t wPort, uint32_t Value)`

`PILOS_WritePortDword` writes a single `dword` to *`wPort`*.

Defined in: `PILIO.H`

Return Value

This function does not return a value.

Parameters

`wPort`	The port to which the value has to be written
`Value`	The value to write to *`wPort`*

3.11.9 PILOS_ReadPciConfigByte

`uint8_t PILOS_ReadPciConfigByte(PilPciDevice *Device*, uint8_t *Offset*)`

`PILOS_ReadPciConfigByte` reads a single byte value from the device at the given offset in the PCI configuration space.

Defined in: `PILIO.H`

Return Value

The value from the offset in the PCI configuration space.

Parameters

Device	The specific device to read from
Offset	The byte offset into the device's PCI configuration space from which to read

3.11.10 PILOS_ReadPciConfigWord

`uint16_t PILOS_ReadPciConfigWord(PilPciDevice *Device*, uint8_t *Offset*)`

`PILOS_ReadPciConfigWord` reads a single word value from the device at the given offset in the PCI configuration space.

Defined in: `PILIO.H`

Return Value

The value from the offset in the PCI configuration space.

Parameters

Device	The specific device to read from
Offset	The byte offset into the device's PCI configuration space from which to read

3.11.11 PILOS_ReadPciConfigDword

`uint32_t PILOS_ReadPciConfigDword(PilPciDevice *Device*, uint8_t *Offset*)`

`PILOS_ReadPciConfigDword` reads a single `dword` value from the device at the given offset in the PCI configuration space.

Defined in: `PILIO.H`

Return Value

The value from the offset in the PCI configuration space.

Parameters

Device	The specific device to read from
Offset	The byte offset into the device's PCI configuration space from which to read

3.11.12 PILOS_WritePciConfigByte

`void PILOS_WritePciConfigByte(PilPciDevice Device, uint8_t Offset, uint8_t Value)`

`PILOS_WritePciConfigByte` writes a single byte value into the device's PCI configuration space at the given byte offset.

Defined in: `PILIO.H`

Return Value

This function does not return a value.

Parameters

Device	The device from which PCI configuration data has to be written
Offset	The byte offset into the PCI configuration space from which to read
Value	The value to write into the PCI configuration space

3.11.13 PILOS_WritePciConfigWord

`void PILOS_WritePciConfigWord(PilPciDevice Device, uint8_t Offset, uint16_t Value)`

`PILOS_WritePciConfigWord` writes a single word value into the device's PCI configuration space at the given byte offset.

Defined in: `PILIO.H`

Return Value

This function does not return a value.

Parameters

Device	The device from which PCI configuration data has to be written
Offset	The byte offset into the PCI configuration space from which to read
Value	The value to write into the PCI configuration space

3.11.14 PILOS_WritePciConfigDword

```
void PILOS_WritePciConfigDword(PilPciDevice Device, uint8_t Offset,
uint32_t Value)
```

`PILOS_WritePciConfigDword` writes a single `dword` value into the device's PCI configuration space at the given byte offset.

Defined in: `PILIO.H`

Return Value

This function does not return a value.

Parameters

`Device`	The device from which PCI configuration data has to be written
`Offset`	The byte offset into the PCI configuration space from which to read
`Value`	The value to write into the PCI configuration space

3.12 Interlocked Services (Optional)

Interlocked services allow operations to be performed on memory in an atomic way, protecting the operation from other threads and other processors. In a multiprocessor environment, these functions assure the processor exclusive use of any shared memory.

3.12.1 PILOS_InterlockedCompareExchange

```
int32_t PILOS_InterlockedCompareExchange(int32_t* Destination,
int32_t Exchange, int32_t Comperand)
```

`PILOS_InterlockedCompareExchange` atomically compares the *`Comperand`* value with the *Destination* value, exchanging the *`Destination`* value and *`Exchange`* value if they are equal. The values are treated as 32-bit unsigned values.

Defined in: `PILINTRL.H`

Return Value

The original value of *`Destination`*.

Parameters

`Destination`	The address of the destination variable
`Exchange`	The exchange value if the comparison succeeds
`Comperand`	The value to be compared with the destination

3.12.2 PILOS_InterlockedDecrement

`int32_t PILOS_InterlockedDecrement(int32_t* Value)`

`PILOS_InterlockedDecrement` atomically decrements the given 32-bit value by one and returns the result.

Defined in: `PILINTRL.H`

Return Value

The resulting decremented value.

Parameters

`Value`	Pointer to the value to be decremented

3.12.3 PILOS_InterlockedExchange

`int32_t PILOS_InterlockedExchange(int32_t* Target, int32_t Value)`

`PILOS_InterlockedExchange` atomically assigns the value of *`Value`* to *`Target`*, returning the old value of *`Target`* as a result.

Defined in: `PILINTRL.H`

Return Value

The prior value of target before the exchange.

Parameters

`Target`	Pointer to the long value to be exchanged
`Value`	The new value for the target

3.12.4 PILOS_InterlockedIncrement

`int32_t PILOS_InterlockedIncrement(int32_t* Value)`

`PILOS_InterlockedIncrement` atomically increments *Value* and returns the result.

Defined in: `PILINTRL.H`

Return Value

The 32-bit resulting incremented value.

Parameters

`Value`	Pointer to the value to be incremented

3.13 Miscellaneous Services

The PIL miscellaneous services are functions that do not fit into any other category.

3.13.1 PIL_Abort

`void PIL_Abort(char* fmt)`

`PIL_Abort` will display the `printf`-style message on the appropriate device and immediately abort execution. The actual effect of aborting execution is implementation dependent. This function never returns a value.

Defined in: `PILMISC.H`

Return Value

This function does not return.

Parameters

fmt	A print-style format string followed by optional additional parameters

3.13.2 PIL_GetCurrentTime

`uint64_t PIL_GetCurrentTime(void)`

`PIL_GetCurrentTime` returns a 64-bit time value in milliseconds. Depending on the implementation, this function may not return the actual clock time. The time returned might be a few milliseconds since an event that occurred in the past (usually system start). However, it is guaranteed that the difference between any two calls of this function is the actual elapsed time in milliseconds.

Defined in: `PILMISC.H`

Return Value

The current time value in milliseconds.

3.14 DEBUG Services

The `DEBUG` services debug applications written to the PIL API. Most of these services automatically turn off and do not generate code when compiled in release mode.

3.14.1 ASSERT

`void ASSERT(BOOL bCondition)`

The `ASSERT` macro checks the given condition and prints a message on the appropriate output device if that condition is not true. The format of the message is as follows:

*** ASSERTION FAILED *filename* line *number* ***

Where *filename* and *number* will be filled in with the appropriate information.

Note: When the DEBUG symbol is not defined, this macro does not generate any code.

Defined in: PILDEBUG.H

Parameters

bCondition	The condition expression that must be TRUE or the assertion fails

3.14.2 ASSERTMSG

```
void ASSERTMSG(BOOL bCondition, char* Msg)
```

The ASSERTMSG macro works just like the ASSERT macro except that it allows a custom message to be printed when the assertion fails:

*** ASSERTION FAILED *filename* line *number* *** *** *custom message*

Where *filename* , *number* , and *custom message* will be filled in with the appropriate information.

Note: When the DEBUG symbol is not defined, this macro will not generate any code.

Defined in: PILDEBUG.H

Parameters

bCondition	The condition expression that must be TRUE or the assertion fails
Msg	The custom message to display if the assertion fails

3.14.3 TRACE

`int32_t TRACE(char* fmt)`

The `TRACE` macro works just like the `printf()` C run-time library function except that when the `DEBUG` symbol is not defined, it does not generate any code. `TRACE` accepts all formatting commands that `printf()` does. Consult an appropriate C run-time library reference for acceptable parameters.

Defined in: `PILDEBUG.H`

Return Value

The total number of characters output to the debug terminal.

Parameters

fmt	The `printf`-style format string followed by additional parameters required by that format string

3.14.4 TRACEx

`int32_t TRACEx(char* fmt)`

The `TRACEx` macros operate in the same manner as the TRACE macro except that the number of parameters is fixed. `TRACE1` through `TRACE6` are defined.

These macros will not generate any code unless the `DEBUG` symbol is defined.

Defined in: `PILDEBUG.H`

Return Value

The total number of characters output to the debug terminal.

Parameters

fmt	The `printf`-type format string followed by one to six additional parameters

3.15 Resource Tracking

In special builds, PIL includes functionality to track all allocated resources. This can be very useful in debugging to find memory leaks. It can also be used to determine the amount each resource is really being allocated at run-time.

The functions described in this section enable applications to access the resource information that PIL is tracking. This enables an application to be built which will dynamically monitor resource allocation. Also included is a simple function to dump all resources into an appropriate debugging device.

Resource tracking is always enabled in the DEBUG builds of PIL. This aids in detecting bad handles. A retail build of PIL can also have resource tracking by enabling the RESTRACK flag when compiling.

3.15.1 PIL_DumpResources

```
void PIL_DumpResources(void)
```

Dumps the currently allocated resources on a suitable debugging terminal.

Defined in: PILDEBUG.H

Return Value

This function does not return a value.

3.15.2 PIL_GetResourceCount

```
PilResult PIL_GetResourceCount(void)
```

Retrieves the current number of each tracked resource.

Defined in: PILDEBUG.H

Return Value

PIL_OK	The operation was successfully completed