



Processs-in-Process (PiP)

2.0.0

Refernce Manual

August 28, 2020

Generated by Doxygen 1.8.5



# Contents

<b>1</b>	<b>Proces-in-Process (PiP) Overview</b>	<b>1</b>
<b>2</b>	<b>PiP Commands</b>	<b>9</b>
2.1	pipcc	9
2.2	pip-check	9
2.3	pip-exec	10
2.4	pip-man	10
2.5	pip-mode	10
2.6	pips	11
2.7	printpipmode	11
<b>3</b>	<b>PiP Functions</b>	<b>13</b>
3.1	PiP Initialization/Finalization	13
3.1.1	Detailed Description	13
3.1.1.1	PiP Initialization/Finalization	13
3.1.2	Function Documentation	13
3.1.2.1	pip_init	13
3.1.2.2	pip_fin	16
3.2	Spawning PiP task	17
3.2.1	Detailed Description	17
3.2.1.1	PiP Spawng PiP (ULP/BLT) task	17
3.2.2	Function Documentation	17
3.2.2.1	pip_spawn_from_main	17
3.2.2.2	pip_spawn_from_func	18
3.2.2.3	pip_spawn_hook	18
3.2.2.4	pip_task_spawn	19
3.2.2.5	pip_spawn	20
3.2.2.6	pip_blt_spawn	22
3.3	Terminating PiP Task	23
3.3.1	Detailed Description	24
3.3.1.1	Terminating PiP task	24
3.3.2	Function Documentation	24
3.3.2.1	pip_exit	24
3.3.2.2	pip_kill_all_tasks	24
3.3.2.3	pip_abort	25
3.4	Waiting for PiP task termination	25
3.4.1	Detailed Description	25
3.4.1.1	Waiting for PiP task termination	25
3.4.2	Function Documentation	25
3.4.2.1	pip_wait	25
3.4.2.2	pip_trywait	26
3.4.2.3	pip_wait_any	27
3.4.2.4	pip_trywait_any	27
3.5	Export/Import Functions	29
3.5.1	Detailed Description	29
3.5.1.1	PiP Export and Import	29

3.5.2	Function Documentation	29
3.5.2.1	pip_named_export	29
3.5.2.2	pip_named_import	30
3.5.2.3	pip_named_tryimport	31
3.5.2.4	pip_export	32
3.5.2.5	pip_import	32
3.6	PiP Miscellaneous Functions	33
3.6.1	Detailed Description	33
3.6.1.1	PiP miscellaneous functions	33
3.6.2	Function Documentation	34
3.6.2.1	pip_get_pipid	34
3.6.2.2	pip_is_initialized	34
3.6.2.3	pip_get_ntasks	34
3.6.2.4	pip_get_mode	35
3.6.2.5	pip_get_mode_str	35
3.6.2.6	pip_get_system_id	36
3.6.2.7	pip_isa_root	36
3.6.2.8	pip_isa_task	36
3.6.2.9	pip_is_threaded	37
3.6.2.10	pip_is_shared_fd	37
3.7	PiP Signaling Functions	37
3.7.1	Detailed Description	38
3.7.1.1	PiP signaling functions	38
3.7.2	Function Documentation	38
3.7.2.1	pip_kill	38
3.7.2.2	pip_sigmask	38
3.7.2.3	pip_signal_wait	39
<b>4</b>	<b>BLT/ULP Functions</b>	<b>41</b>
4.1	Yielding Functionns	41
4.1.1	Detailed Description	41
4.1.1.1	Yielding functions	41
4.1.2	Function Documentation	41
4.1.2.1	pip_yield	41
4.1.2.2	pip_yield_to	42
4.2	Task Queue Operations	42
4.2.1	Detailed Description	43
4.2.1.1	Task queue operations	43
4.2.2	Function Documentation	43
4.2.2.1	pip_task_queue_init	43
4.2.2.2	pip_task_queue_trylock	43
4.2.2.3	pip_task_queue_lock	44
4.2.2.4	pip_task_queue_unlock	44
4.2.2.5	pip_task_queue_isempty	44
4.2.2.6	pip_task_queue_count	45
4.2.2.7	pip_task_queue_enqueue	45
4.2.2.8	pip_task_queue_dequeue	45
4.2.2.9	pip_task_queue_describe	46
4.2.2.10	pip_task_queue_fin	46
4.3	Suspending and Resuming BLT/ULP	46
4.3.1	Detailed Description	47
4.3.1.1	Suspending and resuming BLT/ULP	47
4.3.2	Function Documentation	47
4.3.2.1	pip_suspend_and_enqueue	47
4.3.2.2	pip_suspend_and_enqueue_nolock	48
4.3.2.3	pip_dequeue_and_resume	48
4.3.2.4	pip_dequeue_and_resume_nolock	49
4.3.2.5	pip_dequeue_and_resume_N	49

4.3.2.6	pip_dequeue_and_resume_N_nolock	50
4.4	BLT/ULP Barrier Functions	51
4.4.1	Detailed Description	51
4.4.1.1	BLT/ULP barrier synchronization functions	51
4.4.2	Function Documentation	51
4.4.2.1	pip_barrier_init	51
4.4.2.2	pip_barrier_wait	52
4.4.2.3	pip_barrier_fin	52
4.5	BLT/ULP Mutex Functions	53
4.5.1	Detailed Description	53
4.5.2	Function Documentation	53
4.5.2.1	pip_mutex_init	53
4.5.2.2	pip_mutex_lock	54
4.5.2.3	pip_mutex_unlock	54
4.5.2.4	pip_mutex_fin	55
4.6	BLT/ULP Coupling/Decoupling Functions	55
4.6.1	Detailed Description	55
4.6.1.1	BLT/ULP coupling/decoupling functions	55
4.6.2	Function Documentation	56
4.6.2.1	pip_couple	56
4.6.2.2	pip_decouple	56
4.7	BLT/ULP Miscellaneous Function	56
4.7.1	Detailed Description	57
4.7.1.1	BLT/ULP miscellaneous function	57
4.7.2	Function Documentation	57
4.7.2.1	pip_task_self	57
4.7.2.2	pip_get_task_pipid	57
4.7.2.3	pip_get_task_by_pipid	58
4.7.2.4	pip_set_aux	58
4.7.2.5	pip_get_aux	59
4.7.2.6	pip_get_sched_domain	59



# Chapter 1

## Process-in-Process (PiP) Overview

### Process-in-Process (PiP)

PiP is a user-level library to have the best of the both worlds of multi-process and multi-thread parallel execution models. PiP allows a process to create sub-processes into the same virtual address space where the parent process runs. The parent process and sub-processes share the same address space, however, each process has its own variable set. So, each process runs independently from the other process. If some or all processes agree, then data own by a process can be accessed by the other processes. Those processes share the same address space, just like pthreads, and each process has its own variables like a process. The parent process is called PiP process and a sub-process are called a PiP task.

### PiP Versions

Currently there are three PiP library versions:

- Version 1 - Deprecated
- Version 2 - Stable version
- Version 3 - Stable version supporting BLT and ULP

In this document, **N** denotes the PiP version number.

### Bi-Level Thread (BLT, from v3)

PiP also provides new thread implementation named "Bi-Level Thread (BLT)", again, to take the best of two worlds, Kernel-Level Thread (KLT) and User-Level Thread (ULT) here. A BLT is a PiP task. When a PiP task is created it runs as a KLT. At any point the KLT can become a ULT by decoupling the associated kernel thread from the KLT. The decoupled kernel thread becomes idle. Later, the ULT can become KLT again by coupling with the kernel thread.

### User-Level Process (ULP, from v3)

As described, PiP allows PiP tasks to share the same virtual address space. This mans that a PiP task can context-switch to the other PiP task at user-level. This is called User-Level Process where processes may be derived from the same program or different programs. Threads basically share most of the kernel resources, such as address space, file descriptors, a process id, and so on whilst processes do not. Every process has its ows file descriptor space, for example. When a ULP is scheduled by a KLT having PID 1000, then the getpid() is called by the ULP returns 1000. Further, when the ULT is migrated to be scheduled by the other KLT, then the returned PID is different. So, when implemnting a ULP system, this syscall consistency must be preserved. In ULP on PiP, the

consistency can be maintained by utilizing the above BLT mechanism. When a ULT tries to call a system call, it is coupled with its kernel thread which was created at the beginning as a KLT. It should be note that Thread Local Storage (TLS) regions are also switched when switching ULP (and BLT) contexts.

## Execution Mode

There are several PiP implementation modes which can be selected at the runtime. These implementations can be categorized into two according to the behavior of PiP tasks,

- Process and
- (P)Thread

In the pthread mode, although each PiP task has its own variables unlike thread, PiP task behaves more like P-Thread, having a TID, having the same file descriptor space, having the same signal delivery semantics as Pthread does, and so on. In the process mode, PiP task behaves more like a process, having a PID, having an independent file descriptor space, having the same signal delivery semantics as Linux process does, and so on. The above mentioned ULP can only work with the process mode.

When the `PIP_MODE` environment variable set to "thread" or "pthread" then the PiP library runs based on the pthread mode, and it is set to "process" then it runs with the process mode. There are also three implementations in the process mode; "process:preload," "process:piplone" and "process:got." The "process:preload" mode must be with the `LD_PRELOAD` environment variable setting so that the `clone()` system call wrapper can work with. The "process:piplone" mode can only be specified with the PIP-patched glibc library (see below: GLIBC issues).

There several function provided by the PiP library to absorb the difference due to the execution mode

## License

This project is licensed under the 2-clause simplified BSD License - see the [LICENSE](LICENSE) file for details.

## Installation

### PiP Trial by using Docker image

Download and run the PiP Docker image.

```
$ docker pull rikenpip/pip-vN
$ sudo docker run -it rikenpip/pip-vN /bin/bash
```

### Source Repositories

The installation of PiP related packages must follow the order below;

1. Build PiP-glibc (optional)
2. Build PiP
3. Build PiP-gdb (optional)

Note that if PiP-gdb will not work at all without PiP-glibc. Further, PiP can only create up to around ten PiP tasks without installing PiP-glibc.

- **PiP-glibc** - patched GNU libc for PiP



- **PiP** - Process in Process (this package)
- **PiP-gdb** - patched gdb to debug PiP root and PiP tasks.

Before installing PiP, we strongly recommend you to install PiP-glibc.

After installing PiP, PiP-gdb can be installed too.

### Installation from the source code.

#### 1. Building PiP-glibc (optional)

Fetch source tree (CentOS7 or RHEL7):

```
$ git clone -b pip-centos7 git@git.sys.aics.riken.jp:software/PIP-glibc
```

Fetch source tree (CentOS8 or RHEL8):

```
$ git clone -b pip-centos8 git@git.sys.aics.riken.jp:software/PIP-glibc
```

Build PiP-glibc

```
$ mkdir GLIBC_BUILD_DIR $ cd GLIBC_BUILD_DIR $ GLIBC_SRC_DIR/build.sh --prefix=GLIBC_INSTALL_DIR
```

#### 2. Build PiP library

The same source code can be used for CentOS7 and CentOS8 (RHEL7 and RHEL8).

```
$ git clone -b pip-N git@git.sys.aics.riken.jp:software/PIP $ cd PIP_SRC_DIR $ ./configure --prefix=PIP_INSTALL_DIR [ --with-glibc-libdir=GLIBC_INSTALL_DIR/lib ] $ make install doxygen-install $ cd PIP_INSTALL_DIR/bin $ ./pipnlibs
```

If you want to make sure if the PiP library is correctly installed, then do the following;

```
$ cd PIP_SRC_DIR $ make install-test
```

Important note: The prefix directory of PiP-glibc and the prefix directory of PiP itself must NOT be the same.

#### 3. Build PiP-gdb (optional)

Fetch source tree (CentOS7 or RHEL7):

```
$ git clone -b pip-centos7 git@git.sys.aics.riken.jp:software/PIP-gdb
```

Fetch source tree (CentOS8 or RHEL8):

```
$ git clone -b pip-centos8 git@git.sys.aics.riken.jp:software/PIP-gdb
```

Build PiP-gdb

```
$ cd GLIBC_SRC_DIR $ ./build.sh --prefix=GLIBC_INSTALL_DIR --with-pip=PIP_INSTALL_DIR
```

The prefix directory of PiP-gdb can be the same with the prefix directory of PiP library.

### Installation from RPMs

RPM packages and their yum repository are also available for CentOS 7 / RHEL7.

```
$ sudo rpm -Uvh https://git.sys.r-ccs.riken.jp/PIP/package/el/7/noarch/pip-1/pip-release-N-0.noarch.rpm
$ sudo yum install pip-glibc
$ sudo yum install pip pip-debuginfo
$ sudo yum install pip-gdb
```

If PiP packages are installed by the above RPMs, **PIP\_INSTALL\_DIR** is `"/usr."`

### PiP documents

The following PiP documents are created by using **Doxygen**.

## Man pages

Man pages will be installed at **PIP\_INSTALL\_DIR**/share/man.

```
$ man -M PIP_INSTALL_DIR/share/man 7 libpip
```

Or, use the pip-man command (from v2).

```
$ PIP_INSTALL_DIR/bin/pip-man 7 libpip
```

The above two examples will show you the same document you are reading.

## PDF

PDF documents will be installed at **PIP\_INSTALL\_DIR**/share/doc/pip/pdf.

## Getting Started

### Compile and link your PiP programs

- pipcc(1) command (since v2)

You can use pipcc(1) command to compile and link your PiP programs.

```
$ pipcc -Wall -O2 -g -c pip-prog.c
$ pipcc -Wall -O2 -g -o pip-prog pip-prog.c
```

### Run your PiP programs

- pip-exec(1) command (piprun(1) in PiP v1)

Let's assume you have a non-PiP program(s) and want to run as PiP tasks. All you have to do is to compile your program by using the above pipcc(1) command and to use the pip-exec(1) command to run your program as PiP tasks.

```
$ pipcc myprog.c -o myprog
$ pip-exec -n 8 ./myprog
$ ./myprog
```

In this case, the pip-exec(1) command becomes the PiP root and your program runs as 8 PiP tasks. Your program can also run as a normal (non-PiP) program without using the pip-exec(1) command. Note that the 'myprog.c' may or may not call any PiP functions.

You may write your own PiP programs which includes the PiP root programming. In this case, your program can run without using the pip-exec(1) command.

If you get the following message when you try to run your program;

```
PiP-ERR(19673) './myprog' is not PIE
```

Then this means that the 'myprog' is not compiled by using the pipcc(1) command properly. You may check if your program(s) can run as a PiP root and/or PiP task by using the pip-check(1) command (from v2);

```
$ pip-check a.out
a.out : Root&Task
```

Above example shows that the 'a.out' program can run as a PiP root and PiP tasks.

- pips(1) command (from v2)

You can check if your PiP program is running or not by using the pips(1) command.

List the PiP tasks via the 'ps' command;

```
$ pips -l [ COMMAND ]
```

or, show the activities of PiP tasks via the 'top' command;

```
$ pips -t [ COMMAND ]
```

Here **COMMAND** is the name (not a path) of PiP program you are running.

Additionally you can kill all of your PiP tasks by using the same pips(1) command;

```
$ pips -s KILL [ COMMAND ]
```

## Debugging your PiP programs by the pip-gdb command

The following procedure attaches all PiP tasks, which are created by same PiP root task, as GDB inferiors.

```
$ pip-gdb
(gdb) attach PID
```

The attached inferiors can be seen by the following GDB command:

```
(gdb) info inferiors
Num  Description          Executable
  4   process 6453 (pip 2)  /somewhere/pip-task-2
  3   process 6452 (pip 1)  /somewhere/pip-task-1
  2   process 6451 (pip 0)  /somewhere/pip-task-0
* 1   process 6450 (pip root) /somewhere/pip-root
```

You can select and debug an inferior by the following GDB command:

```
(gdb) inferior 2
[Switching to inferior 2 [process 6451 (pip 0)] (/somewhere/pip-task-0)]
```

When an already-attached program calls 'pip\_spawn()' and becomes a PiP root task, the newly created PiP child tasks aren't attached automatically, but you can add empty inferiors and then attach the PiP child tasks to the inferiors. e.g.

```
.... type Control-Z to stop the root task.
^Z
Program received signal SIGTSTP, Stopped (user).
```

```
(gdb) add-inferior
Added inferior 2
(gdb) inferior 2
(gdb) attach 1902
```

```
(gdb) add-inferior
Added inferior 3
(gdb) inferior 3
(gdb) attach 1903
```

```
(gdb) add-inferior
Added inferior 4
(gdb) inferior 4
(gdb) attach 1904
```

```
(gdb) info inferiors
Num  Description          Executable
* 4   process 1904 (pip 2)  /somewhere/pip-task-2
  3   process 1903 (pip 1)  /somewhere/pip-task-1
  2   process 1902 (pip 0)  /somewhere/pip-task-0
  1   process 1897 (pip root) /somewhere/pip-root
```

You can attach all relevant PiP tasks by:

```
$ pip-gdb -p PID-of-your-PiP-program
```

(from v2)

If the PIP\_GDB\_PATH environment is set to the path pointing to PiP-gdb executable file, then PiP-gdb is automatically attached when an exception signal (SIGSEGV and SIGHUP by default) is delivered. The exception signals can also be defined by setting the PIP\_GDB\_SIGNALS environment. Signal names (case insensitive) can be concatenated by the '+' or '-' symbol. 'all' is reserved to specify most of the signals. For example, 'ALL-TERM' means all signals excepting SIGTERM, another example, 'PIPE+INT' means SIGPIPE and SIGINT. If one of the defined or default signals is delivered, then PiP-gdb will be attached. The PiP-gdb will show backtrace by default. If users specify PIP\_GDB\_COMMAND that a filename containing some GDB commands, then those GDB commands will be executed by the GDB, instead of backtrace, in batch mode. If the PIP\_STOP\_ON\_START environment is set (to any value), then the PiP library delivers SIGSTOP to a spawned PiP task which is about to start user program.

## Mailing List

[pip@ml.riken.jp](mailto:pip@ml.riken.jp)

## Publications

### Research papers

A. Hori, M. Si, B. Geroft, M. Takagi, J. Dayal, P. Balaji, and Y. Ishikawa. "Process-in-process: techniques for practical address-space sharing," In Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '18). ACM, New York, NY, USA, 131-143. DOI: <https://doi.org/10.1145/3208040.3208045>

## Commands

- pipcc
- pips
- printpipmode

## Functions

- pip1-spawn
- pip2-exit
- pip3-wait
- pip\_abort
- pip\_barrier\_fin
- pip\_barrier\_init
- pip\_barrier\_wait
- pip\_blt\_spawn
- pip\_couple

- `pip_decouple`
- `pip_dequeue_and_resume`
- `pip_dequeue_and_resume_N`
- `pip_dequeue_and_resume_N_nolock`
- `pip_dequeue_and_resume_nolock`
- `pip_exit`
- `pip_export`
- `pip_fin`
- `pip_get_aux`
- `pip_get_mode`
- `pip_get_mode_str`
- `pip_get_ntasks`
- `pip_get_pipid`
- `pip_get_sched_domain`
- `pip_get_system_id`
- `pip_get_task_by_pipid`
- `pip_get_task_pipid`
- `pip_import`
- `pip_init`
- `pip_isa_root`
- `pip_isa_task`
- `pip_is_initialized`
- `pip_is_shared_fd`
- `pip_is_threaded`
- `pip_kill`
- `pip_kill_all_tasks`
- `pip_mutex_fin`
- `pip_mutex_init`
- `pip_mutex_lock`
- `pip_mutex_unlock`
- `pip_named_export`
- `pip_named_import`
- `pip_named_tryimport`
- `pip_set_aux`
- `pip_sigmask`
- `pip_signal_wait`

- `pip_spawn`
- `pip_spawn_from_func`
- `pip_spawn_from_main`
- `pip_spawn_hook`
- `pip_suspend_and_enqueue`
- `pip_suspend_and_enqueue_nolock`
- `pip_task_queue_count`
- `pip_task_queue_dequeue`
- `pip_task_queue_describe`
- `pip_task_queue_enqueue`
- `pip_task_queue_fin`
- `pip_task_queue_init`
- `pip_task_queue_isempty`
- `pip_task_queue_lock`
- `pip_task_queue_trylock`
- `pip_task_queue_unlock`
- `pip_task_self`
- `pip_task_spawn`
- `pip_trywait`
- `pip_trywait_any`
- `pip_wait`
- `pip_wait_any`
- `pip_yield`
- `pip_yield_to`
- `ulp0-yield`
- `ulp1-task-queue`
- `ulp2-suspension`
- `ulp3-barrier`
- `ulp4-mutex`
- `ulp5-coupling`

## Author

Atsushi Hori

Riken Center for Computational Science (R-CCS)

Japan

## Chapter 2

# PiP Commands

### 2.1 pipcc

C compiler driver for PiP

#### Synopsis

```
pipcc [PIP-OPTIONS] [CC-COMMAND-OPTIONS_AND_ARGS]
```

#### Parameters

<i>-piproot</i>	the compile (and link) as a PiP root
<i>-piptask</i>	the compile (and link) as a PiP task
<i>-nopip</i>	No PiP related settings will be applied

#### Note

The **-piproot** and **-piptask** options can be specified at the same time. In this case, the compiled object can be both of PiP root and PiP task. This is also the default behavior when none of them is not specified.

#### Environment

if CC environment is set then \$(CC) will be used as a C compiler

#### See Also

pip-exec(1), pip-mode(1)

### 2.2 pip-check

PiP binary checking program if a program can run as a PiP root and/or PiP task

#### Synopsis

```
pipcheck [ OPTION ] pip-prog [...]
```

**Parameters**

<i>-r</i>	check if a.out can be PiP root
<i>-t</i>	check if a.out can be PiP task
<i>-b</i>	check if a.out can be PiP root and/or PiP task
<i>-v</i>	show reason
<i>-h</i>	show this message

**See Also**

pipcc(1)

## 2.3 pip-exec

run program(s) as PiP tasks

**Synopsis**

pip-exec [OPTIONS] <program> ... [ : ... ]

**Description**

**Run** a program as PiP task(s). Mutiple programs can be specified by separating them with ':' to share the same virtual address space with the `pip-exec` command.

**Parameters**

<i>-n N</i>	number of tasks
<i>-f FUNC</i>	function name to start
<i>-c CORE</i>	specify the CPU core number to bind core(s)
<i>-r</i>	core binding in the round-robin fashion

**See Also**

pipcc(1)

## 2.4 pip-man

show PiP man page

**Synopsis**

pip-man [MAN-OPT] MAN-TOPIC

**Description**

Show PiP man pages. It can also accept the man command options.

**See Also**

man(1)

## 2.5 pip-mode

Set PiP execution mode



**Synopsis**

```
pip-mode [OPTION] [PIP-COMMAND]
```

**Description**

The following options are available. If no of them specified, then the compiled output file can be used as both PiP root and PiP task.

**Parameters**

<code>-P</code>	'process' mode
<code>-L</code>	'process:preload' mode
<code>-C</code>	'process:clone' mode
<code>-G</code>	'process:got' mode
<code>-T</code>	'thread' mode
<code>-u</code>	Show usage

**See Also**

pip-exec(1)

## 2.6 pips

List or kill running PiP tasks

**Synopsis**

```
pips [ OPTION ] [ PIP-PROG-NAME ...]
```

**Description**

The following options are available.

**Parameters**

<code>-s SIGNAL</code>	Send the specified signal to the specified PiP tasks
<code>-k</code>	Same as 'pips -s TERM'
<code>-l</code>	List (ps command) running PiP tasks specified. This is the default action.
<code>--list</code>	Same as 'pips -l'
<code>-t</code>	Show running PiP tasks specified by using the top command.
<code>-v</code>	Verbose mode
<code>--top</code>	Same as 'pip -t'

**See Also**

ps(1), top(1)

## 2.7 printpipmode

Print current PiP mode

**Synopsis**

```
printpipmode
```

**See Also**

pip-mode(1)



## Chapter 3

# PiP Functions

### 3.1 PiP Initialization/Finalization

#### Functions

- int [pip\\_init](#) (int \*pipidp, int \*ntasks, void \*\*root\_expp, uint32\_t opts)
- int [pip\\_fin](#) (void)

#### 3.1.1 Detailed Description

##### 3.1.1.1 PiP Initialization/Finalization

#### Description

PiP initialization/finalization functions

#### 3.1.2 Function Documentation

3.1.2.1 int [pip\\_init](#) ( int \* *pipidp*, int \* *ntasks*, void \*\* *root\_expp*, uint32\_t *opts* )

#### Name

[pip\\_init](#)

#### Name

Initialize the PiP library

#### Synopsis

```
#include <pip.h>
int pip\_init( int *pipidp, int *ntasks, void **root_expp, uint32_t opts );
```

#### Description

This function initializes the PiP library. The PiP root process must call this. A PiP task is not required to call this function unless the PiP task calls any PiP functions.

When this function is called by a PiP root, *ntasks*, and *root\_expp* are input parameters. If this is called by a PiP task, then those parameters are output returning the same values input by the root.

A PiP task may or may not call this function. If `pip_init` is not called by a PiP task explicitly, then `pip_init` is called magically and implicitly even if the PiP task program is NOT linked with the PiP library.

## Parameters

out	<i>pipidp</i>	When this is called by the PiP root process, then this returns <code>PIP_PIPID_ROOT</code> , otherwise it returns the PiP ID of the calling PiP task.
in, out	<i>ntasks</i>	When called by the PiP root, it specifies the maximum number of PiP tasks. When called by a PiP task, then it returns the number specified by the PiP root.
in, out	<i>root_exp</i>	If the root PiP is ready to export a memory region to any PiP task(s), then this parameter is to pass the exporting address. If the PiP root is not ready to export or has nothing to export then this variable can be NULL. When called by a PiP task, it returns the exported address by the PiP root, if any.
in	<i>opts</i>	Specifying the PiP execution mode and See below.

## Execution mode option

Users may explicitly specify the PiP execution mode. This execution mode can be categorized in two; process mode and thread mode. In the process execution mode, each PiP task may have its own file descriptors, signal handlers, and so on, just like a process. Contrastingly, in the pthread execution mode, file descriptors and signal handlers are shared among PiP root and PiP tasks while maintaining the privatized variables.

To spawn a PiP task in the process mode, the PiP library modifies the `clone()` flag so that the created PiP task can exhibit the almost same way with that of normal Linux process. There are three ways implemented; using `LD_PRELOAD`, modifying GLIBC, and modifying `GIOT` entry of the `clone()` syscall. One of the option flag values; `PIP_MODE_PTHREAD`, `PIP_MODE_PROCESS`, `PIP_MODE_PROCESS_PRELOAD`, `PIP_MODE_PROCESS_PIPCLONE`, or `PIP_MODE_PROCESS_GOT` can be specified as the option flag. Or, users may specify the execution mode by the `PIP_MODE` environment described below.

## Returns

Zero is returned if this function succeeds. Otherwise an error number is returned.

## Return values

<i>EINVAL</i>	<i>ntasks</i> is negative
<i>EBUSY</i>	PiP root called this function twice or more without calling <code>pip_fin(1)</code> .
<i>EPERM</i>	<i>opts</i> is invalid or unacceptable
<i>EOVERFLOW</i>	<i>ntasks</i> is too large
<i>ELIBSCN</i>	version miss-match between PiP root and PiP task

## Environment

- **PIP\_MODE** Specifying the PiP execution mode. Its value can be either `thread`, `pthread`, `process`, `process:preload`, `process:pipclone`, or `process:got`.
- **LD\_PRELOAD** This is required to set appropriately to hold the path to the `pip_preload.so` file, if the PiP execution mode is `PIP_MODE_PROCESS_PRELOAD` (the `opts` in `pip_init`) and/or the `PIP_MODE` environment is set to `process:preload`. See also the `pip_mode(1)` command to set the environment variable appropriately and easily.
- **PIP\_STACKSZ** Specifying the stack size (in bytes). The **KMP\_STACKSIZE** and **OMP\_STACKSIZE** are also effective. The 't', 'g', 'm', 'k' and 'b' postfix character can be used.
- **PIP\_STOP\_ON\_START** Specifying the PiP ID to stop on start to debug the specified PiP task from the beginning. If the before hook is specified, then the PiP task will be stopped just before calling the before hook.
- **PIP\_GDB\_PATH** If this environment is set to the path pointing to the PiP-gdb executable file, then PiP-gdb is automatically attached when an exception signal (`SIGSEGV` and `SIGHUP` by default) is delivered. The signals which triggers the PiP-gdb invocation can be specified the `PIP_GDB_SIGNALS` environment described below.
- **PIP\_GDB\_COMMAND** If this `PIP_GDB_COMMAND` is set to a filename containing some GDB commands, then those GDB commands will be executed by the GDB in batch mode, instead of backtrace.

- **PIP\_GDB\_SIGNALS** Specifying the signal(s) resulting automatic PiP-gdb attach. Signal names (case insensitive) can be concatenated by the '+' or '-' symbol. 'all' is reserved to specify most of the signals. For example, 'ALL-TERM' means all signals excepting `SIGTERM`, another example, 'PIPE+INT' means `SIGPIPE` and `SIGINT`. Some signals such as `SIGKILL` and `SIGCONT` cannot be specified.
- **PIP\_SHOW\_MAPS** If the value is 'on' and one of the above execution signals is delivered, then the memory map will be shown.
- **PIP\_SHOW\_PIPS** If the value is 'on' and one of the above execution signals is delivered, then the process status by using the `pips` command (see also `pips(1)`) will be shown.

### Bugs

Is is NOT guaranteed that users can spawn tasks up to the number specified by the `ntasks` argument. There are some limitations come from outside of the PiP library (from GLIBC).

### See Also

`pip_named_export(3)`, `pip_export(3)`, `pip_fin(3)`, `pip-mode(1)`, `pips(1)`

### 3.1.2.2 int pip\_fin( void )

#### Name

`pip_fin`

#### Name

Finalize the PiP library

### Synopsis

```
#include <pip.h>
int pip_fin( void );
```

### Description

This function finalizes the PiP library. After calling this, most of the PiP functions will return the error code `EPERM`.

### Returns

zero is returned if this function succeeds. On error, error number is returned.

### Return values

<i>EPERM</i>	<code>pip_init</code> is not yet called
<i>EBUSY</i>	one or more PiP tasks are not yet terminated

### Notes

The behavior of calling `pip_init` after calling this `pip_fin` is not defined and recommended to do so.

### See Also

`pip_init(3)`

## 3.2 Spawning PiP task

### Functions

- void [pip\\_spawn\\_from\\_main](#) (pip\_spawn\_program\_t \*progp, char \*prog, char \*\*argv, char \*\*envv, void \*exp)  
*Setting information to invoke a PiP task starting from the main function.*
- void [pip\\_spawn\\_from\\_func](#) (pip\_spawn\_program\_t \*progp, char \*prog, char \*funcname, void \*arg, char \*\*envv, void \*exp)  
*Setting information to invoke a PiP task starting from a function defined in a program.*
- void [pip\\_spawn\\_hook](#) (pip\_spawn\_hook\_t \*hook, pip\_spawnhook\_t before, pip\_spawnhook\_t after, void \*hookarg)  
*Setting invocation hook information.*
- int [pip\\_task\\_spawn](#) (pip\_spawn\_program\_t \*progp, uint32\_t coreno, uint32\_t opts, int \*pipidp, pip\_spawn\_hook\_t \*hookp)  
*Spawning a PiP task.*
- int [pip\\_spawn](#) (char \*filename, char \*\*argv, char \*\*envv, uint32\_t coreno, int \*pipidp, pip\_spawnhook\_t before, pip\_spawnhook\_t after, void \*hookarg)  
*spawn a PiP task (PiP v1 API and deprecated)*
- int [pip\\_blt\\_spawn](#) (pip\_spawn\_program\_t \*progp, uint32\_t coreno, uint32\_t opts, int \*pipidp, pip\_task\_t \*\*bltp, pip\_task\_queue\_t \*queue, pip\_spawn\_hook\_t \*hookp)  
*spawn a PiP BLT/ULP (Bi-Level Task / User-Level Process)*

### 3.2.1 Detailed Description

#### 3.2.1.1 PiP Spawning PiP (ULP/BLT) task

##### Description

Spawning PiP task or ULP/BLT task

### 3.2.2 Function Documentation

#### 3.2.2.1 void pip\_spawn\_from\_main ( pip\_spawn\_program\_t \* *progp*, char \* *prog*, char \*\* *argv*, char \*\* *envv*, void \* *exp* )

##### Name

pip\_spawn\_from\_main

##### Synopsis

```
#include <pip.h>
void pip_spawn_from_main( pip_spawn_program_t *progp, char *prog, char **argv, char **envv, void *exp )
```

##### Description

This function sets up the `pip_spawn_program_t` structure for spawning a PiP task, starting from the `mmain` function.

##### Parameters

out	<i>progp</i>	Pointer to the <code>pip_spawn_program_t</code> structure in which the program invocation information will be set
-----	--------------	---

in	<i>prog</i>	Path to the executable file.
in	<i>argv</i>	Argument vector.
in	<i>envv</i>	Environment variables. If this is <code>NULL</code> , then the <code>environ</code> variable is used for the spawning PiP task.
in	<i>exp</i>	Export value to the spawning PiP task

**See Also**

`pip_task_spawn(3)`, `pip_spawn_from_func(3)`

**3.2.2.2** `void pip_spawn_from_func ( pip_spawn_program_t * progp, char * prog, char * funcname, void * arg, char ** envv, void * exp )`

**Name**

`pip_spawn_from_func`

**Synopsis**

```
#include <pip.h>
pip_spawn_from_func( pip_spawn_program_t *progp, char *prog, char *funcname, void *arg, char **envv,
void *exp );
```

**Description**

This function sets the required information to invoke a program, starting from the `main()` function. The function should have the function prototype as shown below;

```
int start_func( void *arg )
```

This start function must be globally defined in the program.. The returned integer of the start function will be treated in the same way as the `main` function. This implies that the `pip_wait` function family called from the PiP root can retrieve the return code.

**Parameters**

out	<i>progp</i>	Pointer to the <code>pip_spawn_program_t</code> structure in which the program invocation information will be set
in	<i>prog</i>	Path to the executable file.
in	<i>funcname</i>	Function name to be started
in	<i>arg</i>	Argument which will be passed to the start function
in	<i>envv</i>	Environment variables. If this is <code>NULL</code> , then the <code>environ</code> variable is used for the spawning PiP task.
in	<i>exp</i>	Export value to the spawning PiP task

**See Also**

`pip_task_spawn(3)`, `pip_spawn_from_main(3)`

**3.2.2.3** `void pip_spawn_hook ( pip_spawn_hook_t * hook, pip_spawnhook_t before, pip_spawnhook_t after, void * hookarg )`

**Name**

`pip_spawn_hook`

**Synopsis**

```
#include <pip.h>
void pip_spawn_hook( pip_spawn_hook_t *hook, pip_spawnhook_t before, pip_spawnhook_t after, void
*hookarg );
```



**Description**

The *before* and *after* functions are introduced to follow the programming model of the `fork` and `exec`. *before* function does the prologue found between the `fork` and `exec`. *after* function is to free the argument if it is `malloc()`ed, for example.

**Precondition**

It should be noted that the *before* and *after* functions are called in the *context* of PiP root, although they are running as a part of PiP task (i.e., having PID of the spawning PiP task). Conversely speaking, those functions cannot access the variables defined in the spawning PiP task.

The *before* and *after* hook functions should have the function prototype as shown below;

```
int hook_func( void *hookarg )
```

**Parameters**

out	<i>hook</i>	Pointer to the <code>pip_spawn_hook_t</code> structure in which the invocation hook information will be set
in	<i>before</i>	Just before the executing of the spawned PiP task, this function is called so that file descriptors inherited from the PiP root, for example, can deal with. This is only effective with the PiP process mode. This function is called with the argument <i>hookarg</i> described below.
in	<i>after</i>	This function is called when the PiP task terminates for the cleanup purpose. This function is called with the argument <i>hookarg</i> described below.
in	<i>hookarg</i>	The argument for the <i>before</i> and <i>after</i> function call.

**Note**

Note that the file descriptors and signal handlers are shared between PiP root and PiP tasks in the pthread execution mode.

**See Also**

`pip_task_spawn(3)`

**3.2.2.4** `int pip_task_spawn ( pip_spawn_program_t * progp, uint32_t coreno, uint32_t opts, int * pipidp, pip_spawn_hook_t * hookp )`

**Name**

`pip_task_spawn`

**Synopsis**

```
#include <pip.h>
int pip_task_spawn( pip_spawn_program_t *progp, uint32_t coreno, uint32_t opts, int *pipidp, pip_spawn_
hook_t *hookp );
```

**Description**

This function spawns a PiP task specified by `progp`.

In the process execution mode, the file descriptors having the `FD_CLOEXEC` flag is closed and will not be passed to the spawned PiP task. This simulated close-on-exec will not take place in the pthread execution mode.

## Parameters

out	<i>progp</i>	<b>pip_spawn_program_t</b>
in	<i>coreno</i>	CPU core number for the PiP task to be bound to. By default, <code>coreno</code> is set to zero, for example, then the calling task will be bound to the 'first' core available. This is in mind that the available core numbers are not contiguous. To specify an absolute core number, <code>coreno</code> must be bitwise-ORed with <code>PIP_CPUCORE_ABS</code> . If <code>PIP_CPUCORE_ASIS</code> is specified, then the core binding will not take place.
in	<i>opts</i>	option flags
in, out	<i>pipidp</i>	Specify PiP ID of the spawned PiP task. If <code>PIP_PIPID_ANY</code> is specified, then the PiP ID of the spawned PiP task is up to the PiP library and the assigned PiP ID will be returned.
in	<i>hookp</i>	Hook information to be invoked before and after the program invocation.

## Returns

Zero is returned if this function succeeds. On error, an error number is returned.

## Return values

<i>EPERM</i>	PiP library is not yet initialized
<i>EPERM</i>	PiP task tries to spawn child task
<i>EINVAL</i>	<code>progp</code> is NULL
<i>EINVAL</i>	<code>opts</code> is invalid and/or unacceptable
<i>EINVAL</i>	the value of <code>pipidp</code> is invalid
<i>EINVAL</i>	the <code>coreno</code> is larger than or equal to <code>PIP_CPUCORE_CORENO_MAX</code>
<i>EBUSY</i>	specified PiP ID is already occupied
<i>ENOMEM</i>	not enough memory
<i>ENXIO</i>	<code>dlopen</code> failss

## Bugs

In theory, there is no reason to restrict for a PiP task to spawn another PiP task. However, the current glibc implementation does not allow to do so.

If the root process is multithreaded, only the main thread can call this function.

## See Also

`pip_task_spawn(3)`, `pip_spawn_from_main(3)`, `pip_spawn_from_func(3)`, `pip_spawn_hook(3)`, `pip_spawn(3)`, `pip_blt_spawn(3)`

**3.2.2.5** `int pip_spawn ( char * filename, char ** argv, char ** envv, uint32_t coreno, int * pipidp, pip_spawnhook_t before, pip_spawnhook_t after, void * hookarg )`

## Name

`pip_spawn`

## Synopsis

```
#include <pip.h>
int pip_spawn( char *filename, char **argv, char **envv, uint32_t coreno, int *pipidp, pip_spawnhook_t before,
pip_spawnhook_t after, void *hookarg);
```

## Description

This function spawns a PiP task.

In the process execution mode, the file descriptors having the `FD_CLOEXEC` flag is closed and will not be passed to the spawned PiP task. This simulated close-on-exec will not take place in the pthread execution mode.

#### Parameters

in	<i>filename</i>	The executable to run as a PiP task
in	<i>argv</i>	Argument(s) for the spawned PiP task
in	<i>envv</i>	Environment variables for the spawned PiP task
in	<i>coreno</i>	CPU core number for the PiP task to be bound to. By default, <i>coreno</i> is set to zero, for example, then the calling task will be bound to the first core available. This is in mind that the available core numbers are not contiguous. To specify an absolute core number, <i>coreno</i> must be bitwise-ORed with <code>PIP_CPUCORE_ABS</code> . If <code>PIP_CPUCORE_ASIS</code> is specified, then the core binding will not take place.
in, out	<i>pipidp</i>	Specify PiP ID of the spawned PiP task. If <code>PIP_PIPID_ANY</code> is specified, then the PiP ID of the spawned PiP task is up to the PiP library and the assigned PiP ID will be returned.
in	<i>before</i>	Just before the executing of the spawned PiP task, this function is called so that file descriptors inherited from the PiP root, for example, can deal with. This is only effective with the PiP process mode. This function is called with the argument <i>hookarg</i> described below.
in	<i>after</i>	This function is called when the PiP task terminates for the cleanup purpose. This function is called with the argument <i>hookarg</i> described below.
in	<i>hookarg</i>	The argument for the <i>before</i> and <i>after</i> function call.

#### Returns

Return 0 on success. Return an error code on error.

#### Return values

<i>EPERM</i>	PiP library is not yet initialized
<i>EPERM</i>	PiP task tries to spawn child task
<i>EINVAL</i>	<i>progp</i> is NULL
<i>EINVAL</i>	<i>opts</i> is invalid and/or unacceptable
<i>EINVAL</i>	the value of <i>pipidp</i> is invalid
<i>EINVAL</i>	the <i>coreno</i> is larger than or equal to <code>PIP_CPUCORE_CORENO_MAX</code>
<i>EBUSY</i>	specified PiP ID is already occupied
<i>ENOMEM</i>	not enough memory
<i>ENXIO</i>	<i>dlopen</i> failss

#### Bugs

In theory, there is no reason to restrict for a PiP task to spawn another PiP task. However, the current glibc implementation does not allow to do so.

If the root process is multithreaded, only the main thread can call this function.

#### See Also

`pip_task_spawn(3)`, `pip_spawn_from_main(3)`, `pip_spawn_from_func(3)`, `pip_spawn_hook(3)`, `pip_task_spawn(3)`, `pip_blt_spawn(3)`

3.2.2.6 `int pip_blt_spawn ( pip_spawn_program_t * progp, uint32_t coreno, uint32_t opts, int * pipidp, pip_task_t ** bltp, pip_task_queue_t * queue, pip_spawn_hook_t * hookp )`

#### Name

`pip_blt_spawn`

#### Synopsis

```
#include <pip.h>
int pip_blt_spawn( pip_spawn_program_t *progp, uint32_t coreno, uint32_t opts, int *pipidp, pip_task_t **bltp,
pip_task_queue_t *queue, pip_spawn_hook_t *hookp );
```

#### Description

This function spawns a BLT (PiP task) specified by `progp`. The created and returned BLT is another form of a PiP task. It is an opaque object, essentially a double-linked list. Thus created BLT can be enqueued or dequeued to/from a `pip_task_queue_t`.

In the process execution mode, the file descriptors having the `FD_CLOEXEC` flag is closed and will not be passed to the spawned PiP task. This simulated close-on-exec will not take place in the pthread execution mode.

#### Parameters

out	<i>progp</i>	<b>pip_spawn_program_t</b>
in	<i>coreno</i>	CPU core number for the PiP task to be bound to. By default, <i>coreno</i> is set to zero, for example, then the calling task will be bound to the first core available. This is in mind that the available core numbers are not contiguous. To specify an absolute core number, <i>coreno</i> must be bitwise-ORed with <code>PIP_CPUCORE_ABS</code> . If <code>PIP_CPUCORE_ASIS</code> is specified, then the core binding will not take place.
in	<i>opts</i>	option flags. If <code>PIP_TASK_INACTIVE</code> is set, the created BLT is suspended and enqueued to the specified <i>queue</i> . Otherwise the BLT will schedules the BLTs in <i>queue</i> .
in, out	<i>pipidp</i>	Specify PiP ID of the spawned PiP task. If <code>PIP_PIPID_ANY</code> is specified, then the PiP ID of the spawned PiP task is up to the PiP library and the assigned PiP ID will be returned. The PiP execution mode can also be specified (see below).
in, out	<i>bltp</i>	returns created BLT
in	<i>queue</i>	PiP task queue. See the above <i>opts</i> description.
in	<i>hookp</i>	Hook information to be invoked before and after the program invocation.

#### Returns

Return 0 on success. Return an error code on error.

#### Return values

<i>EPERM</i>	PiP library is not yet initialized
<i>EPERM</i>	PiP task tries to spawn child task
<i>EINVAL</i>	<i>progp</i> is NULL
<i>EINVAL</i>	<i>opts</i> is invalid and/or unacceptable
<i>EINVAL</i>	the value off <i>pipidp</i> is invalid

<i>EBUSY</i>	specified PiP ID is already occupied
<i>ENOMEM</i>	not enough memory
<i>ENXIO</i>	dlopen failss

#### Execution mode option

Users may explicitly specify the PiP execution mode. This execution mode can be categorized in two; process mode and thread mode. In the process execution mode, each PiP task may have its own file descriptors, signal handlers, and so on, just like a process. Contrastingly, in the pthread execution mode, file descriptors and signal handlers are shared among PiP root and PiP tasks while maintaining the privatized variables.

To spawn a PiP task in the process mode, the PiP library modifies the **clone()** flag so that the created PiP task can exhibit the almost same way with that of normal Linux process. There are three ways implemented; using LD\_PRELOAD, modifying GLIBC, and modifying GLOB entry of the **clone()** syscall. One of the option flag values; **PIP\_MODE\_PTHREAD**, **PIP\_MODE\_PROCESS**, **PIP\_MODE\_PROCESS\_PRELOAD**, **PIP\_MODE\_PROCESS\_PIPCLONE**, or **PIP\_MODE\_PROCESS\_GOT** can be specified as the option flag. Or, users may specify the execution mode by the **PIP\_MODE** environment described below.

#### Note

In theory, there is no reason to restrict for a PiP task to spawn another PiP task. However, the current implementation fails to do so. If the root process is multithreaded, only the main thread can call this function.

#### Environment

- **PIP\_MODE** Specifying the PiP execution mode. The value can be one of; 'process', 'process:preload', 'process:got' and 'thread' (or 'pthread').
- **PIP\_STACKSZ** Specifying the stack size (in bytes). The **KMP\_STACKSIZE** and **OMP\_STACKSIZE** can also be specified. The 't', 'g', 'm', 'k' and 'b' postfix character can be used.
- **PIP\_STOP\_ON\_START** Specifying the PiP ID to stop on start PiP task program to debug from the beginning. If the before hook is specified, then the PiP task will be stopped just before calling the before hook.
- **PIP\_STACKSZ** Specifying the stack size (in bytes). The **KMP\_STACKSIZE** and **OMP\_STACKSIZE** can also be specified. The 't', 'g', 'm', 'k' and 'b' postfix character can be used.

#### Bugs

In theory, there is no reason to restrict for a PiP task to spawn another PiP task. However, the current glibc implementation does not allow to do so.

If the root process is multithreaded, only the main thread can call this function.

#### See Also

`pip_task_spawn(3)`, `pip_spawn_from_main(3)`, `pip_spawn_from_func(3)`, `pip_spawn_hook(3)`, `pip_task_spawn(3)`, `pip_spawn(3)`

## 3.3 Terminating PiP Task

### Functions

- void `pip_exit` (int status)  
*terminate the calling PiP task*
- int `pip_kill_all_tasks` (void)  
*kill all PiP tasks*
- void `pip_abort` (void)  
*Kill all PiP tasks and then kill PiP root.*

### 3.3.1 Detailed Description

#### 3.3.1.1 Terminating PiP task

##### Description

Function to terminate PiP task normally or abnormally (abort).

### 3.3.2 Function Documentation

#### 3.3.2.1 void pip\_exit ( int *status* )

##### Name

pip\_exit

##### Synopsis

```
#include <pip.h>
void pip_exit( int status );
```

##### Description

When the main function or the start function of a PiP task returns with an integer value, then it has the same effect of calling `pip_exit` with the returned value.

##### Parameters

<i>in</i>	<i>status</i>	This status is returned to PiP root.
-----------	---------------	--------------------------------------

##### Note

This function can be used regardless to the PiP execution mode. `exit(3)` is called in the process mode and `pthread_exit(3)` is called in the pthread mode.

##### See Also

pip\_wait(3), pip\_trywait(3), pip\_wait\_any(3), pip\_trywait\_any(3)

#### 3.3.2.2 int pip\_kill\_all\_tasks ( void )

##### Name

pip\_kill\_all\_tasks

##### Synopsis

```
#include <pip.h>
int pip_kill_all_tasks( void );
```

##### Note

This function must be called from PiP root.

##### Returns

Return 0 on success. Return an error code on error.

## Return values

<i>EPERM</i>	The PiP library is not initialized yet
<i>EPERM</i>	Not called from root

## 3.3.2.3 void pip\_abort ( void )

## Name

pip\_abort

## Synopsis

```
#include <pip.h>
void pip_abort( void );
```

## 3.4 Waiting for PiP task termination

## Functions

- int [pip\\_wait](#) (int pipid, int \*status)  
*wait for the termination of a PiP task*
- int [pip\\_trywait](#) (int pipid, int \*status)  
*wait for the termination of a PiP task in a non-blocking way*
- int [pip\\_wait\\_any](#) (int \*pipid, int \*status)  
*Wait for the termination of any PiP task.*
- int [pip\\_trywait\\_any](#) (int \*pipid, int \*status)  
*non-blocking version of pip\_wait\_any*

## 3.4.1 Detailed Description

## 3.4.1.1 Waiting for PiP task termination

## Description

Functions to wait for PiP task termination. All functions listed here must only be called from PiP root.

## 3.4.2 Function Documentation

## 3.4.2.1 int pip\_wait ( int pipid, int \* status )

## Name

pip\_wait

## Synopsis

```
#include <pip.h>
int pip_wait( int pipid, int *status );
```

## Description

This function can be used regardless to the PiP execution mode. This function blocks until the specified PiP task terminates. The macros such as `WIFEXITED` and so on defined in Glibc can be applied to the returned `status` value.

**Parameters**

in	<i>pipid</i>	PiP ID to wait for.
out	<i>status</i>	Status value of the terminated PiP task

**Returns**

Return 0 on success. Return an error code on error.

**Return values**

<i>EPERM</i>	PiP library is not initialized yet
<i>EPERM</i>	This function is called other than PiP root
<i>EDEADLK</i>	The specified <i>pipid</i> is the one of PiP root
<i>ECHILD</i>	The target PiP task does not exist or it was already terminated and waited for

**See Also**

`pip_exit(3)`, `pip_trywait(3)`, `pip_wait_any(3)`, `pip_trywait_any(3)`

**3.4.2.2 int pip\_trywait ( int *pipid*, int \* *status* )****Name**

`pip_trywait`

**Synopsis**

```
#include <pip.h>
int pip_trywait( int pipid, int *status );
```

**Description**

This function can be used regardless to the PiP execution mode. This function behaves like the `wait` function of glibc and the macros such as `WIFEXITED` and so on can be applied to the returned `status` value.

**Synopsis**

```
#include <pip.h>
int pip_trywait( int pipid, int *status );
```

**Parameters**

in	<i>pipid</i>	PiP ID to wait for.
out	<i>status</i>	Status value of the terminated PiP task

**Note**

This function can be used regardless to the PiP execution mode.

**Returns**

Return 0 on success. Return an error code on error.



## Return values

<i>EPERM</i>	The PiP library is not initialized yet
<i>EPERM</i>	This function is called other than PiP root
<i>EDEADLK</i>	The specified <code>pipid</code> is the one of PiP root
<i>ECHILD</i>	The target PiP task does not exist or it was already terminated and waited for

## See Also

`pip_exit(3)`, `pip_wait(3)`, `pip_wait_any(3)`, `pip_trywait_any(3)`

3.4.2.3 `int pip_wait_any ( int * pipid, int * status )`

## Name

`pip_wait_any`

## Synopsis

```
#include <pip.h>
int pip_wait_any( int *pipid, int *status );
```

## Description

This function can be used regardless to the PiP execution mode. This function blocks until any of PiP tasks terminates. The macros such as `WIFEXITED` and so on defined in Glibc can be applied to the returned `status` value.

## Parameters

out	<i>pipid</i>	PiP ID of terminated PiP task.
out	<i>status</i>	Exit value of the terminated PiP task

## Returns

Return 0 on success. Return an error code on error.

## Return values

<i>EPERM</i>	The PiP library is not initialized yet
<i>EPERM</i>	This function is called other than PiP root
<i>ECHILD</i>	The target PiP task does not exist or it was already terminated and waited for

## See Also

`pip_exit(3)`, `pip_wait(3)`, `pip_trywait(3)`, `pip_trywait_any(3)`

3.4.2.4 `int pip_trywait_any ( int * pipid, int * status )`

## Name

`pip_trywait_any`

## Synopsis

```
#include <pip.h>
int pip_trywait_any( int *pipid, int *status );
```

**Description**

This function can be used regardless to the PiP execution mode. This function blocks until any of PiP tasks terminates. The macros such as `WIFEXITED` and so on defined in `Glibc` can be applied to the returned `status` value.

## Parameters

out	<i>pipid</i>	PiP ID of terminated PiP task.
out	<i>status</i>	Exit value of the terminated PiP task

## Returns

Return 0 on success. Return an error code on error.

## Return values

<i>EPERM</i>	The PiP library is not initialized yet
<i>EPERM</i>	This function is called other than PiP root
<i>ECHILD</i>	There is no PiP task to wait for

## See Also

`pip_exit(3)`, `pip_wait(3)`, `pip_trywait(3)`, `pip_wait_any(3)`

## 3.5 Export/Import Functions

### Functions

- int `pip_named_export` (void \*exp, const char \*format,...) `__attribute__((format(printf, export an address of the calling PiP root or a PiP task to the others.`
- int int `pip_named_import` (int pipid, void \*\*expp, const char \*format,...) `__attribute__((format(printf, import the named exported address`
- int int int `pip_named_tryimport` (int pipid, void \*\*expp, const char \*format,...) `__attribute__((format(printf, import the named exported address (non-blocking)`
- int int int int `pip_export` (void \*exp)  
*export an address*
- int `pip_import` (int pipid, void \*\*expp)  
*import exported address of a PiP task*

### 3.5.1 Detailed Description

#### 3.5.1.1 PiP Export and Import

##### Description

Export and import functions to exchange addresses among tasks

### 3.5.2 Function Documentation

#### 3.5.2.1 int `pip_named_export` ( void \* exp, const char \* format, ... )

##### Name

`pip_named_export`

##### Synopsis

```
#include <pip.h>
int pip_named_export( void *exp, const char *format, ... )
```

**Description**

Pass an address of a memory region to the other PiP task. Unlike the simple `pip_export` and `pip_import` functions which can only export one address per task, `pip_named_export` and `pip_named_import` can associate a name with an address so that PiP root or PiP task can exchange arbitrary number of addresses.

**Parameters**

in	<i>exp</i>	an address to be passed to the other PiP task
in	<i>format</i>	a <code>printf</code> format to give the exported address a name. If this is <code>NULL</code> , then the name is assumed to be "".

**Returns**

Return 0 on success. Return an error code on error.

**Return values**

<i>EPERM</i>	<code>pip_init</code> is not yet called.
<i>EBUSY</i>	The name is already registered.
<i>ENOMEM</i>	Not enough memory

**Note**

The addresses exported by `pip_named_export` cannot be imported by calling `pip_import`, and vice versa.

**See Also**

`pip_named_import(3)`

3.5.2.2 `int int pip_named_import ( int pipid, void ** expp, const char * format, ... )`

**Name**

`pip_named_import`

**Synopsis**

```
#include <pip.h>
int pip_named_import( int pipid, void **expp, const char *format, ... )
```

**Description**

Import an address exported by the specified PiP task and having the specified name. If it is not exported yet, the calling task will be blocked. The

**Parameters**

in	<i>pipid</i>	The PiP ID to import the exposed address
out	<i>expp</i>	The starting address of the exposed region of the PiP task specified by the <i>pipid</i> .

<i>in</i>	<i>format</i>	a <code>printf</code> format to give the exported address a name
-----------	---------------	--

**Note**

There is possibility of deadlock when two or more tasks are mutually waiting for exported addresses.

The addresses exported by `pip_export` cannot be imported by calling `pip_named_import`, and vice versa.

**Returns**

zero is returned if this function succeeds. On error, an error number is returned.

**Return values**

<i>EPERM</i>	<code>pip_init</code> is not yet called.
<i>EINVAL</i>	The specified <code>pipid</code> is invalid
<i>ENOMEM</i>	Not enough memory
<i>ECANCELED</i>	The target task is terminated
<i>EDEADLK</i>	<code>pipid</code> is the calling task and tries to block itself

**See Also**

`pip_named_export(3)`, `pip_named_tryimport(3)`, `pip_export(3)`, `pip_import(3)`

**3.5.2.3 int int int pip\_named\_tryimport ( int *pipid*, void \*\* *expp*, const char \* *format*, ... )****Name**

`pip_named_tryimport`

**Synopsis**

```
#include <pip.h>
int pip_named_tryimport( int pipid, void **expp, const char *format, ... )
```

**Description**

Import an address exported by the specified PiP task and having the specified name. If it is not exported yet, this returns `EAGAIN`.

**Parameters**

<i>in</i>	<i>pipid</i>	The PiP ID to import the exposed address
<i>out</i>	<i>expp</i>	The starting address of the exposed region of the PiP task specified by the <i>pipid</i> .
<i>in</i>	<i>format</i>	a <code>printf</code> format to give the exported address a name

**Note**

The addresses exported by `pip_export` cannot be imported by calling `pip_named_import`, and vice versa.

**Returns**

Zero is returned if this function succeeds. On error, an error number is returned.

## Return values

<i>EPERM</i>	<code>pip_init</code> is not yet called.
<i>EINVAL</i>	The specified <code>pipid</code> is invalid
<i>ENOMEM</i>	Not enough memory
<i>ECANCELED</i>	The target task is terminated
<i>EAGAIN</i>	Target is not exported yet

## See Also

`pip_named_export(3)`, `pip_named_import(3)`, `pip_export(3)`, `pip_import(3)`

3.5.2.4 `int pip_export ( void * exp )`

## Name

`pip_export`

## Synopsis

```
#include <pip.h>
int pip_export( void *exp );
```

## Description

Pass an address of a memory region to the other PiP task. This is a very naive implementation in PiP v1 and deprecated. Once a task export an address, there is no way to change the exported address or undo export.

## Parameters

<i>in</i>	<i>exp</i>	An addresss
-----------	------------	-------------

## Returns

Return 0 on success. Return an error code on error.

## Return values

<i>EPERM</i>	PiP library is not initialized yet
--------------	------------------------------------

## See Also

`pip_import(3)`, `pip_named_export(3)`, `pip_named_import(3)`, `pip_named_tryimport(3)`

3.5.2.5 `int pip_import ( int pipid, void ** exp )`

## Name

`pip_import`

## Synopsis

```
#include <pip.h>
int pip_import( void **expp );
```

## Description

Get an address exported by the specified PiP task. This is a very naive implementation in PiP v1 and deprecated. If the address is not yet exported at the time of calling this function, then `NULL` is returned.

## Parameters

in	<i>pipid</i>	The PiP ID to import the exported address
out	<i>exp</i>	The exported address

## Returns

Return 0 on success. Return an error code on error.

## Return values

<i>EPERM</i>	PiP library is not initialized yet
--------------	------------------------------------

## See Also

`pip_export(3)`, `pip_named_export(3)`, `pip_named_import(3)`, `pip_named_tryimport(3)`

## 3.6 PiP Miscellaneous Functions

### Functions

- int `pip_get_pipid` (int \*pipid)  
*get PiP ID of the calling task*
- int `pip_is_initialized` (void)  
*Query is PiP library is already initialized.*
- int `pip_get_ntasks` (int \*ntasksp)  
*get the maximum number of the PiP tasks*
- int `pip_get_mode` (int \*modep)  
*get the PiP execution mode*
- const char \* `pip_get_mode_str` (void)  
*get a character string of the current execution mode*
- int `pip_get_system_id` (int pipid, pip\_id\_t \*idp)  
*deliver a process or thread ID defined by the system*
- int `pip_isa_root` (void)  
*check if calling PiP task is a PiP root or not*
- int `pip_isa_task` (void)  
*check if calling PiP task is a PiP task or not*
- int `pip_is_threaded` (int \*flagp)  
*check if PiP execution mode is pthread or not*
- int `pip_is_shared_fd` (int \*flagp)  
*check if file descriptors are shared or not. This is equivalent with the `pip_is_threaded` function.*

### 3.6.1 Detailed Description

#### 3.6.1.1 PiP miscellaneous functions

##### Description

Miscellaneous functions for PiP task (not BLT/ULP)

### 3.6.2 Function Documentation

#### 3.6.2.1 `int pip_get_pipid ( int * pipidp )`

##### Name

`pip_get_pipid`

##### Synopsis

```
#include <pip.h>
int pip_get_pipid( int *pipidp );
```

##### Parameters

out	<i>pipidp</i>	This parameter points to the variable which will be set to the PiP ID of the calling task
-----	---------------	---

##### Returns

Return 0 on success. Return an error code on error.

##### Return values

<i>EPERM</i>	PiP library is not initialized yet
--------------	------------------------------------

#### 3.6.2.2 `int pip_is_initialized ( void )`

##### Name

`pip_is_initialized`

##### Synopsis

```
#include <pip.h>
int pip_is_initialized( void );
```

##### Returns

Return a non-zero value if PiP is already initialized. Otherwise this returns zero.

#### 3.6.2.3 `int pip_get_ntasks ( int * ntasksp )`

##### Name

`pip_get_ntasks`

##### Synopsis

```
#include <pip.h>
int pip_get_ntasks( int *ntasksp );
```

##### Parameters



out	<i>ntasksp</i>	Maximum number of PiP tasks is returned
-----	----------------	---

**Returns**

Return 0 on success. Return an error code on error.

**Return values**

<i>EPERM</i>	PiP library is not yet initialized
--------------	------------------------------------

**3.6.2.4 int pip\_get\_mode ( int \* modep )****Name**

pip\_get\_mode

**Synopsis**

```
#include <pip.h>
int pip_get_mode( int *modep );
```

**Parameters**

out	<i>modep</i>	Returned PiP execution mode
-----	--------------	-----------------------------

**Returns**

Return 0 on success. Return an error code on error.

**Return values**

<i>EPERM</i>	PiP library is not yet initialized
--------------	------------------------------------

**See Also**

pip\_get\_mode\_str(3)

**3.6.2.5 const char\* pip\_get\_mode\_str ( void )****Name**

pip\_get\_mode\_str

**Synopsis**

```
#include <pip.h>
char *pip_get_mode_str( void );
```

**Returns**

Return the name string of the current execution mode. If PiP library is not initialized yet, then this return NULL.

### 3.6.2.6 int pip\_get\_system\_id ( int *pipid*, pip\_id\_t \* *idp* )

#### Name

pip\_get\_system\_id

#### Synopsis

```
#include <pip.h>
int pip_get_system_id( int *pipid, uintptr_t *idp );
```

#### Description

The returned object depends on the PiP execution mode. In the process mode it returns TID (Thread ID, not PID) and in the thread mode it returns thread (*pthread\_t*) associated with the PiP task. This function can be used regardless of the PiP execution mode.

#### Parameters

out	<i>pipid</i>	PiP ID of a target PiP task
out	<i>idp</i>	a pointer to store the ID value

#### Returns

Return 0 on success. Return an error code on error.

#### Return values

<i>EPERM</i>	The PiP library is not initialized yet
--------------	--

### 3.6.2.7 int pip\_isa\_root ( void )

#### Name

pip\_isa\_root

#### Synopsis

```
#include <pip.h>
int pip_isa_root( void );
```

#### Returns

Return a non-zero value if the caller is the PiP root. Otherwise this returns zero.

### 3.6.2.8 int pip\_isa\_task ( void )

#### Name

pip\_isa\_task

#### Synopsis

```
#include <pip.h>
int pip_isa_task( void );
```

#### Returns

Return a non-zero value if the caller is the PiP task. Otherwise this returns zero.

## 3.6.2.9 int pip\_is\_threaded ( int \* flagp )

## Name

pip\_is\_threaded

## Synopsis

```
#include <pip.h>
int pip_is_threaded( int *flagp );
```

## Parameters

out	flagp	set to a non-zero value if PiP execution mode is Pthread
-----	-------	--

## Returns

Return 0 on success. Return an error code on error.

## Return values

EPERM	The PiP library is not initialized yet
-------	--

## 3.6.2.10 int pip\_is\_shared\_fd ( int \* flagp )

## Name

pip\_is\_shared\_fd

## Synopsis

```
#include <pip.h>
int pip_is_shared_fd( int *flagp );
```

## Parameters

out	flagp	set to a non-zero value if FDs are shared
-----	-------	---

## Returns

Return 0 on success. Return an error code on error.

## Return values

EPERM	The PiP library is not initialized yet
-------	--

## 3.7 PiP Signaling Functions

### Functions

- int [pip\\_kill](#) (int pipid, int signal)  
*deliver a signal to PiP task*
- int [pip\\_sigmask](#) (int how, const sigset\_t \*sigmask, sigset\_t \*oldmask)  
*set signal mask of the current PiP task*
- int [pip\\_signal\\_wait](#) (int signal)  
*wait for a signal*

### 3.7.1 Detailed Description

#### 3.7.1.1 PiP signaling functions

##### Description

Signal manipulating functions. All functions listed here are agnostic to the PiP execution mode.

### 3.7.2 Function Documentation

#### 3.7.2.1 `int pip_kill( int pipid, int signal )`

##### Name

`pip_kill`

##### Synopsis

```
#include <pip.h>
int pip_kill( int pipid, int signal );
```

##### Parameters

out	<i>pipid</i>	PiP ID of a target PiP task to deliver the signal
out	<i>signal</i>	signal number to be delivered

##### Returns

Return 0 on success. Return an error code on error.

##### Return values

<i>EPERM</i>	PiP library is not yet initialized
<i>EINVAL</i>	An invalid signal number or invalid PiP ID is specified

##### See Also

`tkill(2)`

#### 3.7.2.2 `int pip_sigmask( int how, const sigset_t * sigmask, sigset_t * oldmask )`

##### Name

`pip_sigmask`

##### Synopsis

```
#include <pip.h>
int pip_sigmask( int how, const sigset_t * sigmask, sigset_t * oldmask );
```

##### Description

This function is agnostic to the PiP execution mode.

## Parameters

in	<i>how</i>	see <b>sigprocmask</b> or <b>pthread_sigmask</b>
in	<i>sigmask</i>	signal mask
out	<i>oldmask</i>	old signal mask

## Returns

Return 0 on success. Return an error code on error.

## Return values

<i>EPERM</i>	PiP library is not yet initialized
<i>EINVAL</i>	An invalid signal number or invalid PiP ID is specified

## See Also

sigprocmask, pthread\_sigmask

3.7.2.3 int pip\_signal\_wait ( int *signal* )

## Name

pip\_signal\_wait

## Synopsis

```
#include <pip.h>
int pip_signal_wait( int signal );
```

## Description

This function is agnostic to the PiP execution mode.

## Parameters

in	<i>signal</i>	signal to wait
----	---------------	----------------

## Returns

Return 0 on success. Return an error code on error.

## Note

This function does NOT return the `EINTR` error. This case is treated as normal return;

## See Also

sigwait, sigsuspend



# Chapter 4

## BLT/ULP Functions

### 4.1 Yielding Functionns

#### Functions

- int [pip\\_yield](#) (int flag)  
*Yield.*
- int [pip\\_yield\\_to](#) (pip\_task\_t \*task)  
*Yield to the specified PiP task.*

#### 4.1.1 Detailed Description

##### 4.1.1.1 Yielding functions

#### Description

Yielding execution of the calling BLT/ULP

#### 4.1.2 Function Documentation

##### 4.1.2.1 int [pip\\_yield](#) ( int *flag* )

#### Name

[pip\\_yield](#)

#### Synopsis

```
#include <pip.h>
int pip\_yield( int flag );
```

#### Parameters

<i>in</i>	<i>flag</i>	to specify the behavior of yielding. See below.
-----------	-------------	---

#### Returns

No context-switch takes place during the call, then this returns zero. If the context-switch to the other BLT happens, then this returns `EINTR`.

## Parameters

<i>flag</i>	If <b>PIP_YIELD_USER</b> , the calling task is scheduling PiP task(s) then the calling task switch to the next eligible-to-run BLT. If <b>PIP_YIELD_SYSTEM</b> , regardless if the calling task is active or inactive, it calls <code>sched_yield</code> . If <b>PIP_YIELD_DEFAULT</b> or zero, then both <b>PIP_YIELD_USER</b> and <b>PIP_YIELD_SYSTEM</b> will be effective.
-------------	--

## See Also

`pip_yield_to(3)`

4.1.2.2 `int pip_yield_to ( pip_task_t * task )`

## Name

`pip_yield_to`

## Synopsis

```
#include <pip.h>
int pip_yield( pip_task_t *task );
```

## Description

Context-switch to the specified PiP task. If `task` is `NULL`, then this works the same as what `pip_yield(3)` does with `PIP_YIELD_DEFAULT`.

## Parameters

<i>in</i>	<i>task</i>	Target PiP task to switch.
-----------	-------------	----------------------------

## Returns

Return `Zero` or `EINTR` on success. Return an error code on error.

## Return values

<i>EPERM</i>	PiP library is not yet initialized or already
<i>EPERM</i>	The specified task belongs to the other scheduling domain.

## See Also

`pip_yield(3)`

## 4.2 Task Queue Operations

## Functions

- `int pip_task_queue_init (pip_task_queue_t *queue, pip_task_queue_methods_t *methods)`  
*Initialize task queue.*
- `int pip_task_queue_trylock (pip_task_queue_t *queue)`  
*Try locking task queue.*
- `void pip_task_queue_lock (pip_task_queue_t *queue)`  
*Lock task queue.*
- `void pip_task_queue_unlock (pip_task_queue_t *queue)`  
*Unlock task queue.*



- int [pip\\_task\\_queue\\_isempty](#) (pip\_task\_queue\_t \*queue)  
*Query function if the current task has some tasks to be scheduled with.*
- int [pip\\_task\\_queue\\_count](#) (pip\_task\_queue\_t \*queue, int \*np)  
*Count the length of task queue.*
- void [pip\\_task\\_queue\\_enqueue](#) (pip\_task\_queue\_t \*queue, pip\_task\_t \*task)  
*Enqueue a BLT.*
- pip\_task\_t \* [pip\\_task\\_queue\\_dequeue](#) (pip\_task\_queue\_t \*queue)  
*Dequeue a task from a task queue.*
- void [pip\\_task\\_queue\\_describe](#) (pip\_task\_queue\_t \*queue, FILE \*fp)  
*Describe queue.*
- int [pip\\_task\\_queue\\_fin](#) (pip\_task\_queue\_t \*queue)  
*Finalize a task queue.*

## 4.2.1 Detailed Description

### 4.2.1.1 Task queue operations

#### Description

Manipulating ULP/BLT task queue functions

## 4.2.2 Function Documentation

### 4.2.2.1 int pip\_task\_queue\_init ( pip\_task\_queue\_t \* *queue*, pip\_task\_queue\_methods\_t \* *methods* )

#### Name

pip\_task\_queue\_init

#### Synopsis

```
#include <pip.h>
int pip_task_queue_init( pip_task_queue_t *queue, pip_task_queue_methods_t *methods );
```

#### Parameters

in	<i>queue</i>	A task queue
in	<i>methods</i>	Must be set to NULL. Researved for future use.

#### Returns

Always return 0.

### 4.2.2.2 int pip\_task\_queue\_trylock ( pip\_task\_queue\_t \* *queue* )

#### Name

pip\_task\_queue\_trylock

#### Synopsis

```
#include <pip.h>
int pip\_task\_queue\_trylock( pip_task_queue_t *queue );
```

**Parameters**

<i>in</i>	<i>queue</i>	A task queue
-----------	--------------	--------------

**Returns**

Returns a non-zero value if lock succeeds.

**4.2.2.3 void pip\_task\_queue\_lock ( pip\_task\_queue\_t \* *queue* )****Name**

pip\_task\_queue\_lock

**Synopsis**

```
#include <pip.h>
int pip_task_queue_lock( pip_task_queue_t *queue );
```

**Parameters**

<i>in</i>	<i>queue</i>	A task queue
-----------	--------------	--------------

**Returns**

This function returns no error

**4.2.2.4 void pip\_task\_queue\_unlock ( pip\_task\_queue\_t \* *queue* )****Name**

pip\_task\_queue\_unlock

**Synopsis**

```
#include <pip.h>
int pip_task_queue_unlock( pip_task_queue_t *queue );
```

**Parameters**

<i>in</i>	<i>queue</i>	A task queue
-----------	--------------	--------------

**Returns**

This function returns no error

**4.2.2.5 int pip\_task\_queue\_isempty ( pip\_task\_queue\_t \* *queue* )****Name**

pip\_task\_queue\_isempty

**Synopsis**

```
#include <pip.h>
int pip_task_queue_isempty( pip_task_queue_t *queue );
```

## Parameters

in	<i>queue</i>	A task queue
----	--------------	--------------

## Returns

Returns a non-zero value if the queue is empty

4.2.2.6 int pip\_task\_queue\_count( pip\_task\_queue\_t \* *queue*, int \* *np* )

## Name

pip\_task\_queue\_count

## Synopsis

```
#include <pip.h>
int pip_task_queue_count( pip_task_queue_t *queue, int *np );
```

## Parameters

in	<i>queue</i>	A task queue
out	<i>np</i>	the queue length returned

## Returns

Return 0 on success. Return an error code on error.

## Return values

<i>EINVAL</i>	queue is NULL
<i>EINVAL</i>	np is NULL

4.2.2.7 void pip\_task\_queue\_enqueue( pip\_task\_queue\_t \* *queue*, pip\_task\_t \* *task* )

## Name

pip\_task\_queue\_enqueue

## Synopsis

```
#include <pip.h>
void pip_task_queue_enqueue( pip_task_queue_t *queue, pip_task_t *task );
```

## Parameters

in	<i>queue</i>	A task queue
in	<i>task</i>	A task to be enqueued

## Note

It is the user responsibility to lock (and unlock) the queue.

4.2.2.8 pip\_task\_t\* pip\_task\_queue\_dequeue( pip\_task\_queue\_t \* *queue* )

## Name

pip\_task\_queue\_dequeue

**Synopsis**

```
#include <pip.h>
pip_task_t* pip_task_queue_dequeue( pip_task_queue_t *queue );
```

**Parameters**

<i>in</i>	<i>queue</i>	A task queue
-----------	--------------	--------------

**Returns**

Dequeued task iss returned. If the queue is empty then `NULL` is returned.

**Note**

It is the user responsibility to lock (and unlock) the queue.

4.2.2.9 void pip\_task\_queue\_describe ( pip\_task\_queue\_t \* *queue*, FILE \* *fp* )

**Name**

pip\_task\_queue\_describe

**Synopsis**

```
#include <pip.h>
void pip_task_queue_describe( pip_task_queue_t *queue, FILE *fp );
```

**Parameters**

<i>in</i>	<i>queue</i>	A task queue
<i>in</i>	<i>fp</i>	a File pointer

4.2.2.10 int pip\_task\_queue\_fin ( pip\_task\_queue\_t \* *queue* )

**Name**

pip\_task\_queue\_fin

**Synopsis**

```
#include <pip.h>
int pip_task_queue_fin( pip_task_queue_t *queue );
```

**Parameters**

<i>in</i>	<i>queue</i>	A task queue
-----------	--------------	--------------

**Returns**

Zero is returned always

## 4.3 Suspending and Resuming BLT/ULP

**Functions**

- int [pip\\_suspend\\_and\\_enqueue](#) (pip\_task\_queue\_t \*queue, pip\_enqueue\_callback\_t callback, void \*cbarg)

*suspend the current task and enqueue it with lock*

- int [pip\\_suspend\\_and\\_enqueue\\_nolock](#) (pip\_task\_queue\_t \*queue, pip\_enqueue\_callback\_t callback, void \*cbarg)

*suspend the current task and enqueue it without locking the queue*

- int [pip\\_dequeue\\_and\\_resume](#) (pip\_task\_queue\_t \*queue, pip\_task\_t \*sched)

*dequeue a task and make it runnable*

- int [pip\\_dequeue\\_and\\_resume\\_nolock](#) (pip\_task\_queue\_t \*queue, pip\_task\_t \*sched)

*dequeue a task and make it runnable*

- int [pip\\_dequeue\\_and\\_resume\\_N](#) (pip\_task\_queue\_t \*queue, pip\_task\_t \*sched, int \*np)

*dequeue multiple tasks and resume the execution of them*

- int [pip\\_dequeue\\_and\\_resume\\_N\\_nolock](#) (pip\_task\_queue\_t \*queue, pip\_task\_t \*sched, int \*np)

*dequeue tasks and resume the execution of them*

### 4.3.1 Detailed Description

#### 4.3.1.1 Suspending and resuming BLT/ULP

##### Description

Suspending and resuming BLT/ULP

### 4.3.2 Function Documentation

#### 4.3.2.1 int pip\_suspend\_and\_enqueue ( pip\_task\_queue\_t \* queue, pip\_enqueue\_callback\_t callback, void \* cbarg )

##### Name

pip\_suspend\_and\_enqueue

##### Synopsis

```
#include <pip.h>
```

```
int pip_suspend_and_enqueue( pip_task_queue_t *queue, pip_enqueue_callback_t callback, void *cbarg );
```

##### Description

The **queue** is locked just before the calling task is enqueued and unlocked after the calling task is enqueued. After then the **callback** function is called.

As the result of this suspension, a context-switch takes place if there is at least one eligible-to-run task in the scheduling queue (this is hidden from users). If there is no other task to schedule then the kernel thread of the current task will be blocked.

##### Parameters

in	<i>queue</i>	A task queue
in	<i>callback</i>	A callback function which is called immediately after the task is enqueued
in	<i>cbarg</i>	An argument given to the callback function

##### Returns

Return 0 on success. Return an error code on error.

## Return values

<i>EPERM</i>	PiP library is not initialized yet
<i>EINVAL</i>	queue is NULL

## See Also

pip\_enqueue\_and\_suspend\_nolock(3), pip\_dequeue\_and\_resume(3)

#### 4.3.2.2 int pip\_suspend\_and\_enqueue\_nolock ( pip\_task\_queue\_t \* *queue*, pip\_enqueue\_callback\_t *callback*, void \* *cbarg* )

## Name

pip\_suspend\_and\_enqueue\_nolock

## Synopsis

```
#include <pip.h>
int pip_suspend_and_enqueue_nolock( pip_task_queue_t *queue, pip_enqueue_callback_t callback, void
*cbarg );
```

## Description

Unlike `pip_suspend_and_enqueue`, this function never locks the queue. It is the user's responsibility to lock the queue before calling this function and unlock the queue after calling this function. The **callback** function can be used for unlocking.

As the result of this suspension, a context-switch takes place if there is at least one eligible-to-run task in the scheduling queue (this is hidden from users). If there is no other task to schedule then the kernel thread of the current task will be blocked.

## Parameters

in	<i>queue</i>	A task queue
in	<i>callback</i>	A callback function which is called when enqueued
in	<i>cbarg</i>	An argument given to the callback function

## Returns

Return 0 on success. Return an error code on error.

## Return values

<i>EPERM</i>	PiP library is not initialized yet
<i>EINVAL</i>	queue is NULL

#### 4.3.2.3 int pip\_dequeue\_and\_resume ( pip\_task\_queue\_t \* *queue*, pip\_task\_t \* *sched* )

## Name

pip\_dequeue\_and\_resume

## Description

The **queue** is locked and then unlocked when to dequeued a task.

## Synopsis

```
#include <pip.h>
int pip_dequeue_and_resume( pip_task_queue_t *queue, pip_task_t *sched );
```

## Parameters

in	<i>queue</i>	A task queue
in	<i>sched</i>	A task to specify a scheduling domain

## Returns

If succeedss, 0 is returned. Otherwise an error code is returned.

## Return values

<i>EPERM</i>	PiP library is not initialized yet
<i>EINVAL</i>	queue is NULL
<i>ENOENT</i>	queue is empty.

## 4.3.2.4 int pip\_dequeue\_and\_resume\_nolock ( pip\_task\_queue\_t \* queue, pip\_task\_t \* sched )

## Name

pip\_dequeue\_and\_resume\_nolock

## Synopsis

```
#include <pip.h>
int pip_dequeue_and_resume( pip_task_queue_t *queue, pip_task_t *sched );
```

## Description

Task in the queue is dequeued and scheduled by the specified *sched*. If *sched* is NULL, then the task is enqueued into the scheduling queue of calling task.

It is the user's responsibility to lock the queue beofre calling this function and unlock the queue after calling this function.

## Parameters

in	<i>queue</i>	A task queue
in	<i>sched</i>	A task to specify a scheduling domain

## Returns

This function returns no error

## Return values

<i>EPERM</i>	PiP library is not initialized yet
<i>EINVAL</i>	queue is NULL
<i>ENOENT</i>	queue is empty.

## 4.3.2.5 int pip\_dequeue\_and\_resume\_N ( pip\_task\_queue\_t \* queue, pip\_task\_t \* sched, int \* np )

## Name

pip\_dequeue\_and\_resume\_N

## Synopsis

```
#include <pip.h>
int pip_dequeue_and_resume_N( pip_task_queue_t *queue, pip_task_t *sched, int *np );
```

**Description**

The specified number of tasks are dequeued and scheduled by the specified *sched*. If *sched* is NULL, then the task is enqueued into the scheduling queue of calling task.

The **queue** is locked and unlocked when dequeued.

**Parameters**

in	<i>queue</i>	A task queue
in	<i>sched</i>	A task to specify a scheduling domain
in, out	<i>np</i>	A pointer to an interger which spcifies the number of tasks dequeued and actual number of tasks dequeued is returned. When PIP_TASK_ALL is specified, then all tasks in the queue will be resumed.

**Returns**

This function returns no error

**Return values**

<i>EPERM</i>	PiP library is not initialized yet
<i>EINVAL</i>	<i>queue</i> is NULL
<i>EINVAL</i>	the specified number of tasks is invalid
<i>ENOENT</i>	<i>queue</i> is empty.

It is the user's responsibility to lock the queue beofre calling this function and unlock the queue after calling this function.

4.3.2.6 int pip\_dequeue\_and\_resume\_N\_nolock ( pip\_task\_queue\_t \* *queue*, pip\_task\_t \* *sched*, int \* *np* )

**Name**

pip\_dequeue\_and\_resume\_N\_nolock

**Synopsis**

```
#include <pip.h>
int pip_dequeue_and_resume_N_nolock( pip_task_queue_t *queue, pip_task_t *sched, int *np );
```

**Description**

The specified number of tasks are dequeued and scheduled by the specified *sched*. If *sched* is NULL, then the task is enqueued into the scheduling queue of calling task.

It is the user's responsibility to lock the queue beofre calling this function and unlock the queue after calling this function.

**Parameters**

in	<i>queue</i>	A task queue
in	<i>sched</i>	A task to specify a scheduling domain
in, out	<i>np</i>	A pointer to an interger which spcifies the number of tasks dequeued and actual number of tasks dequeued is returned. When PIP_TASK_ALL is specified, then all tasks in the queue will be resumed.

**Returns**

This function returns no error



## Return values

<i>EPERM</i>	PiP library is not initialized yet
<i>EINVAL</i>	queue is NULL
<i>EINVAL</i>	the specified number of tasks is invalid
<i>ENOENT</i>	queue is empty.

## 4.4 BLT/ULP Barrier Functions

### Functions

- int [pip\\_barrier\\_init](#) (pip\_barrier\_t \*barrp, int n)  
*initialize barrier synchronization structure*
- int [pip\\_barrier\\_wait](#) (pip\_barrier\_t \*barrp)  
*wait on barrier synchronization in a busy-wait way int [pip\\_barrier\\_wait](#)( pip\_barrier\_t \*barrp );*
- int [pip\\_barrier\\_fin](#) (pip\_barrier\_t \*barrp)  
*finalize barrier synchronization structure*

### 4.4.1 Detailed Description

#### 4.4.1.1 BLT/ULP barrier synchronization functions

##### Description

BLT/ULP barrier synchronization functions

##### Description

BLT/ULP mutex functions

### 4.4.2 Function Documentation

#### 4.4.2.1 int pip\_barrier\_init ( pip\_barrier\_t \* barrp, int n )

##### Name

pip\_barrier\_init

##### Synopsis

```
#include <pip.h>
int pip_barrier_init( pip_barrier_t *barrp, int n );
```

##### Parameters

in	<i>barrp</i>	pointer to a PiP barrier structure
in	<i>n</i>	number of participants of this barrier synchronization

##### Returns

Return 0 on success. Return an error code on error.

## Return values

<i>EPERM</i>	PiP library is not yet initialized or already finalized
<i>EINAVL</i>	n is invalid

## Note

This barrier works on PiP tasks only.

## See Also

pip\_barrier\_init(3), pip\_barrier\_fin(3),

## 4.4.2.2 int pip\_barrier\_wait ( pip\_barrier\_t \* barrp )

## Name

pip\_barrier\_wait

## Synopsis

```
#include <pip.h>
int pip_barrier_wait( pip_barrier_t *barrp );
```

## Parameters

in	<i>barrp</i>	pointer to a PiP barrier structure
----	--------------	------------------------------------

## Returns

Return 0 on success. Return an error code on error.

## Return values

<i>EPERM</i>	PiP library is not yet initialized or already finalized
--------------	---

## See Also

pip\_barrier\_init(3), pip\_barrier\_fin(3),

## 4.4.2.3 int pip\_barrier\_fin ( pip\_barrier\_t \* barrp )

## Name

pip\_barrier\_fin

## Synopsis

```
#include <pip.h>
int pip_barrier_fin( pip_barrier_t *barrp );
```

## Parameters

---

<code>in</code>	<code>barrp</code>	pointer to a PiP barrier structure
-----------------	--------------------	------------------------------------

**Returns**

Return 0 on success. Return an error code on error.

**Return values**

<code>EPERM</code>	PiP library is not yet initialized or already finalized
<code>EBUSY</code>	there are some tasks waiting for barrier synchronization

**See Also**

`pip_barrier_init(3)`, `pip_barrier_wait(3)`,

## 4.5 BLT/ULP Mutex Functions

**Functions**

- `int pip_mutex_init (pip_mutex_t *mutex)`  
*Initialize PiP mutex.*
- `int pip_mutex_lock (pip_mutex_t *mutex)`  
*Lock PiP mutex.*
- `int pip_mutex_unlock (pip_mutex_t *mutex)`  
*Unlock PiP mutex.*
- `int pip_mutex_fin (pip_mutex_t *mutex)`  
*Finalize PiP mutex.*

### 4.5.1 Detailed Description

### 4.5.2 Function Documentation

#### 4.5.2.1 `int pip_mutex_init ( pip_mutex_t * mutex )`

**Name**

`pip_mutex_init`

**Synopsis**

```
#include <pip.h>
int pip_mutex_init( pip_mutex_t *mutex );
```

**Parameters**

<code>in, out</code>	<code>mutex</code>	pointer to the PiP task mutex
----------------------	--------------------	-------------------------------

**Returns**

Return 0 on success. Return an error code on error.

## Return values

<i>EPERM</i>	PiP library is not yet initialized or already finalized
--------------	---

## See Also

`pip_mutex_lock(3)`, `pip_mutex_unlock(3)`, `pip_mutex_fin(3)`

4.5.2.2 `int pip_mutex_lock ( pip_mutex_t * mutex )`

## Name

`pip_mutex_lock`

## Synopsis

```
#include <pip.h>
int pip_mutex_lock( pip_mutex_t *mutex );
```

## Parameters

<i>in</i>	<i>mutex</i>	pointer to the PiP task mutex
-----------	--------------	-------------------------------

## Returns

Return 0 on success. Return an error code on error.

## Return values

<i>EPERM</i>	PiP library is not yet initialized or already finalized
--------------	---

## See Also

`pip_mutex_init(3)`, `pip_mutex_unlock(3)`, `pip_mutex_fin(3)`

4.5.2.3 `int pip_mutex_unlock ( pip_mutex_t * mutex )`

## Name

`pip_mutex_unlock`

## Synopsis

```
#include <pip.h>
int pip_mutex_unlock( pip_mutex_t *mutex );
```

## Parameters

<i>in</i>	<i>mutex</i>	pointer to the PiP task mutex
-----------	--------------	-------------------------------

## Returns

Return 0 on success. Return an error code on error.

## Return values

<i>EPERM</i>	PiP library is not yet initialized or already finalized
--------------	---

## See Also

`pip_mutex_init(3)`, `pip_mutex_lock(3)`, `pip_mutex_fin(3)`

4.5.2.4 `int pip_mutex_fin ( pip_mutex_t * mutex )`

## Name

`pip_mutex_fin`

## Synopsis

```
#include <pip.h>
int pip_mutex_fin( pip_mutex_t *mutex );
```

## Parameters

<i>in, out</i>	<i>mutex</i>	pointer to the PiP task mutex
----------------	--------------	-------------------------------

## Returns

Return 0 on success. Return an error code on error.

## Return values

<i>EPERM</i>	PiP library is not yet initialized or already finalized
<i>EBUSY</i>	There is one or more waiting PiP task

## See Also

`pip_mutex_lock(3)`, `pip_mutex_unlock(3)`

## 4.6 BLT/ULP Coupling/Decoupling Functions

### Functions

- `int pip_couple (void)`  
*Couple the curren task with the original kernel thread.*
- `int pip_decouple (pip_task_t *task)`  
*Decouple the curren task from the kernel thread.*

### 4.6.1 Detailed Description

#### 4.6.1.1 BLT/ULP coupling/decoupling functions

## Description

BLT/ULP coupling/decoupling functions

## 4.6.2 Function Documentation

### 4.6.2.1 `int pip_couple ( void )`

#### Name

`pip_couple`

#### Synopsis

```
#include <pip.h>
int pip_couple( void );
```

#### Returns

Return 0 on success. Return an error code on error.

#### Return values

<i>EPERM</i>	PiP library is not yet initialized or already finalized
<i>EBUSY</i>	the current task is already coupled with a kernel thread

### 4.6.2.2 `int pip_decouple ( pip_task_t * task )`

#### Name

`pip_decouple`

#### Synopsis

```
#include <pip.h>
int pip_decouple( pip_task_t *sched )
```

#### Parameters

<code>in</code>	<i>task</i>	specify the scheduling task to schedule the decoupled task (calling this function). If <code>NULL</code> , then the previously coupled <code>pip_task</code> takes place.
-----------------	-------------	---

#### Returns

Return 0 on success. Return an error code on error.

#### Return values

<i>EPERM</i>	PiP library is not yet initialized or already finalized
<i>EBUSY</i>	the current task is already decoupled from a kernel thread

## 4.7 BLT/ULP Miscellaneous Function

### Functions

- `pip_task_t * pip_task_self (void)`  
Return the current task.
- `int pip_get_task_pipid (pip_task_t *task, int *pipidp)`  
Return PIPID of a PiP task.
- `int pip_get_task_by_pipid (int pipid, pip_task_t **taskp)`

- get PiP task from PiP ID*
- int [pip\\_set\\_aux](#) (pip\_task\_t \*task, void \*aux)  
*Associate user data with a PiP task.*
- int [pip\\_get\\_aux](#) (pip\_task\_t \*task, void \*\*auxp)  
*Retrieve the user data associated with a PiP task.*
- int [pip\\_get\\_sched\\_domain](#) (pip\_task\_t \*\*domainp)  
*Return the task representing the scheduling domain.*

## 4.7.1 Detailed Description

### 4.7.1.1 BLT/ULP miscellaneous function

#### Description

BLT/ULP miscellaneous function

## 4.7.2 Function Documentation

### 4.7.2.1 pip\_task\_t\* pip\_task\_self ( void )

#### Name

pip\_task\_self

#### Synopsis

```
#include <pip.h>
pip_task_t *pip_task_self( void );
```

#### Returns

Return the current task.

### 4.7.2.2 int pip\_get\_task\_pipid ( pip\_task\_t \* task, int \* pipidp )

#### Name

pip\_get\_task\_pipid

#### Synopsis

```
#include <pip.h>
int pip_get_task_pipid( pip_task_t *task, int *pipidp );
```

#### Parameters

in	<i>task</i>	a PiP task
out	<i>pipidp</i>	PiP ID of the specified task

#### Returns

Return 0 on success. Return an error code on error.

## Return values

<i>EINAVL</i>	task is NULL
<i>EPERM</i>	PiP library is not yet initialized or already finalized

## 4.7.2.3 int pip\_get\_task\_by\_pipid ( int pipid, pip\_task\_t \*\* taskp )

## Name

pip\_get\_task\_by\_pipid

## Synopsis

```
#include <pip.h>
int pip_get_task_by_pipid( int pipid, pip_task_t **taskp );
```

## Parameters

in	<i>pipid</i>	PiP ID
out	<i>taskp</i>	returning PiP task of the specified PiP ID

## Returns

Return 0 on success. Return an error code on error.

## Return values

<i>EPERM</i>	PiP library is not yet initialized or already finalized
<i>ENOENT</i>	No such PiP task
<i>ERANGE</i>	The specified <i>pipid</i> is out of range

## 4.7.2.4 int pip\_set\_aux ( pip\_task\_t \* task, void \* aux )

## Name

pip\_set\_aux

## Synopsis

```
#include <pip.h>
int pip_set_aux( pip_task_t *task, void *aux );
```

## Parameters

in	<i>task</i>	PiP task. If NULL, then the data is associated with the current PiP task
in	<i>aux</i>	Pointer to the user dat to associate with

## Returns

Return 0 on success. Return an error code on error.

## Return values

<i>EPERM</i>	PiP library is not yet initialized or already finalized
--------------	---

## See Also

pip\_get\_aux(3)



4.7.2.5 `int pip_get_aux ( pip_task_t * task, void ** auxp )`

## Name

`pip_get_aux`

## Synopsis

```
#include <pip.h>
int pip_get_aux( pip_task_t *task, void **auxp );
```

## Parameters

<code>in</code>	<code>task</code>	PiP task. If NULL, then the data is associated with the current PiP task
<code>out</code>	<code>auxp</code>	Returned user data

## Returns

Return 0 on success. Return an error code on error.

## Return values

<code>EINAVL</code>	<code>domainp</code> is NULL or <code>auxp</code> is NULL
<code>EPERM</code>	PiP library is not yet initialized or already finalized

## See Also

`pip_set_aux(3)`4.7.2.6 `int pip_get_sched_domain ( pip_task_t ** domainp )`

## Name

`pip_get_sched_domain`

## Synopsis

```
#include <pip.h>
int pip_get_sched_domain( pip_task_t **domainp );
```

## Parameters

<code>out</code>	<code>domainp</code>	Returned scheduling domain of the current task
------------------	----------------------	--

## Returns

Return 0 on success. Return an error code on error.

## Return values

<code>EPERM</code>	PiP library is not yet initialized or already finalized
--------------------	---

# Index

## BLT/ULP Barrier Functions, 51

- pip\_barrier\_fin, 52
- pip\_barrier\_init, 51
- pip\_barrier\_wait, 52

## BLT/ULP Coupling/Decoupling Functions, 55

- pip\_couple, 56
- pip\_decouple, 56

## BLT/ULP Miscellaneous Function, 56

- pip\_get\_aux, 58
- pip\_get\_sched\_domain, 59
- pip\_get\_task\_by\_pipid, 58
- pip\_get\_task\_pipid, 57
- pip\_set\_aux, 58
- pip\_task\_self, 57

## BLT/ULP Mutex Functions, 53

- pip\_mutex\_fin, 55
- pip\_mutex\_init, 53
- pip\_mutex\_lock, 54
- pip\_mutex\_unlock, 54

## Export/Import Functions, 29

- pip\_export, 32
- pip\_import, 32
- pip\_named\_export, 29
- pip\_named\_import, 30
- pip\_named\_tryimport, 31

## PiP Initialization/Finalization, 13

- pip\_fin, 16
- pip\_init, 13

## PiP Miscellaneous Functions, 33

- pip\_get\_mode, 35
- pip\_get\_mode\_str, 35
- pip\_get\_ntasks, 34
- pip\_get\_pipid, 34
- pip\_get\_system\_id, 35
- pip\_is\_initialized, 34
- pip\_is\_shared\_fd, 37
- pip\_is\_threaded, 36
- pip\_isa\_root, 36
- pip\_isa\_task, 36

## PiP Signaling Functions, 37

- pip\_kill, 38
- pip\_sigmask, 38
- pip\_signal\_wait, 39

## pip\_abort

- Terminating PiP Task, 25

## pip\_barrier\_fin

- BLT/ULP Barrier Functions, 52

## pip\_barrier\_init

## BLT/ULP Barrier Functions, 51

### pip\_barrier\_wait

- BLT/ULP Barrier Functions, 52

### pip\_blt\_spawn

- Spawning PiP task, 21

### pip\_couple

- BLT/ULP Coupling/Decoupling Functions, 56

### pip\_decouple

- BLT/ULP Coupling/Decoupling Functions, 56

### pip\_dequeue\_and\_resume

- Suspending and Resuming BLT/ULP, 48

### pip\_dequeue\_and\_resume\_N

- Suspending and Resuming BLT/ULP, 49

### pip\_dequeue\_and\_resume\_N\_nolock

- Suspending and Resuming BLT/ULP, 50

### pip\_dequeue\_and\_resume\_nolock

- Suspending and Resuming BLT/ULP, 49

### pip\_exit

- Terminating PiP Task, 24

### pip\_export

- Export/Import Functions, 32

### pip\_fin

- PiP Initialization/Finalization, 16

### pip\_get\_aux

- BLT/ULP Miscellaneous Function, 58

### pip\_get\_mode

- PiP Miscellaneous Functions, 35

### pip\_get\_mode\_str

- PiP Miscellaneous Functions, 35

### pip\_get\_ntasks

- PiP Miscellaneous Functions, 34

### pip\_get\_pipid

- PiP Miscellaneous Functions, 34

### pip\_get\_sched\_domain

- BLT/ULP Miscellaneous Function, 59

### pip\_get\_system\_id

- PiP Miscellaneous Functions, 35

### pip\_get\_task\_by\_pipid

- BLT/ULP Miscellaneous Function, 58

### pip\_get\_task\_pipid

- BLT/ULP Miscellaneous Function, 57

### pip\_import

- Export/Import Functions, 32

### pip\_init

- PiP Initialization/Finalization, 13

### pip\_is\_initialized

- PiP Miscellaneous Functions, 34

### pip\_is\_shared\_fd

- PiP Miscellaneous Functions, 37

- pip\_is\_threaded
  - PiP Miscellaneous Functions, [36](#)
- pip\_isa\_root
  - PiP Miscellaneous Functions, [36](#)
- pip\_isa\_task
  - PiP Miscellaneous Functions, [36](#)
- pip\_kill
  - PiP Signaling Functions, [38](#)
- pip\_kill\_all\_tasks
  - Terminating PiP Task, [24](#)
- pip\_mutex\_fin
  - BLT/ULP Mutex Functions, [55](#)
- pip\_mutex\_init
  - BLT/ULP Mutex Functions, [53](#)
- pip\_mutex\_lock
  - BLT/ULP Mutex Functions, [54](#)
- pip\_mutex\_unlock
  - BLT/ULP Mutex Functions, [54](#)
- pip\_named\_export
  - Export/Import Functions, [29](#)
- pip\_named\_import
  - Export/Import Functions, [30](#)
- pip\_named\_tryimport
  - Export/Import Functions, [31](#)
- pip\_set\_aux
  - BLT/ULP Miscellaneous Function, [58](#)
- pip\_sigmask
  - PiP Signaling Functions, [38](#)
- pip\_signal\_wait
  - PiP Signaling Functions, [39](#)
- pip\_spawn
  - Spawning PiP task, [20](#)
- pip\_spawn\_from\_func
  - Spawning PiP task, [18](#)
- pip\_spawn\_from\_main
  - Spawning PiP task, [17](#)
- pip\_spawn\_hook
  - Spawning PiP task, [18](#)
- pip\_suspend\_and\_enqueue
  - Suspending and Resuming BLT/ULP, [47](#)
- pip\_suspend\_and\_enqueue\_nolock
  - Suspending and Resuming BLT/ULP, [48](#)
- pip\_task\_queue\_count
  - Task Queue Operations, [45](#)
- pip\_task\_queue\_dequeue
  - Task Queue Operations, [45](#)
- pip\_task\_queue\_describe
  - Task Queue Operations, [46](#)
- pip\_task\_queue\_enqueue
  - Task Queue Operations, [45](#)
- pip\_task\_queue\_fin
  - Task Queue Operations, [46](#)
- pip\_task\_queue\_init
  - Task Queue Operations, [43](#)
- pip\_task\_queue\_isempty
  - Task Queue Operations, [44](#)
- pip\_task\_queue\_lock
  - Task Queue Operations, [44](#)
- pip\_task\_queue\_trylock
  - Task Queue Operations, [43](#)
- pip\_task\_queue\_unlock
  - Task Queue Operations, [44](#)
- pip\_task\_self
  - BLT/ULP Miscellaneous Function, [57](#)
- pip\_task\_spawn
  - Spawning PiP task, [19](#)
- pip\_trywait
  - Waiting for PiP task termination, [26](#)
- pip\_trywait\_any
  - Waiting for PiP task termination, [27](#)
- pip\_wait
  - Waiting for PiP task termination, [25](#)
- pip\_wait\_any
  - Waiting for PiP task termination, [27](#)
- pip\_yield
  - Yielding Functionns, [41](#)
- pip\_yield\_to
  - Yielding Functionns, [42](#)
- pip\_abort, [25](#)
- pip\_barrier\_fin, [52](#)
- pip\_barrier\_init, [51](#)
- pip\_barrier\_wait, [52](#)
- pip\_blt\_spawn, [22](#)
- pip\_couple, [56](#)
- pip\_decouple, [56](#)
- pip\_dequeue\_and\_resume, [48](#)
- pip\_dequeue\_and\_resume\_N, [49](#)
- pip\_dequeue\_and\_resume\_N\_nolock, [50](#)
- pip\_dequeue\_and\_resume\_nolock, [49](#)
- pip\_exit, [24](#)
- pip\_export, [32](#)
- pip\_fin, [16](#)
- pip\_get\_aux, [59](#)
- pip\_get\_mode, [35](#)
- pip\_get\_mode\_str, [35](#)
- pip\_get\_ntasks, [34](#)
- pip\_get\_pipid, [34](#)
- pip\_get\_sched\_domain, [59](#)
- pip\_get\_system\_id, [36](#)
- pip\_get\_task\_by\_pipid, [58](#)
- pip\_get\_task\_pipid, [57](#)
- pip\_import, [32](#)
- pip\_init, [13](#)
- pip\_is\_initialized, [34](#)
- pip\_is\_shared\_fd, [37](#)
- pip\_is\_threaded, [37](#)
- pip\_isa\_root, [36](#)
- pip\_isa\_task, [36](#)
- pip\_kill, [38](#)
- pip\_kill\_all\_tasks, [24](#)
- pip\_mutex\_fin, [55](#)
- pip\_mutex\_init, [53](#)
- pip\_mutex\_lock, [54](#)
- pip\_mutex\_unlock, [54](#)
- pip\_named\_export, [29](#)
- pip\_named\_import, [30](#)

- [pip\\_named\\_tryimport, 31](#)
  - [pip\\_set\\_aux, 58](#)
  - [pip\\_sigmask, 38](#)
  - [pip\\_signal\\_wait, 39](#)
  - [pip\\_spawn, 20](#)
  - [pip\\_spawn\\_from\\_func, 18](#)
  - [pip\\_spawn\\_from\\_main, 17](#)
  - [pip\\_spawn\\_hook, 18](#)
  - [pip\\_suspend\\_and\\_enqueue, 47](#)
  - [pip\\_suspend\\_and\\_enqueue\\_nolock, 48](#)
  - [pip\\_task\\_queue\\_count, 45](#)
  - [pip\\_task\\_queue\\_dequeue, 45](#)
  - [pip\\_task\\_queue\\_describe, 46](#)
  - [pip\\_task\\_queue\\_enqueue, 45](#)
  - [pip\\_task\\_queue\\_fin, 46](#)
  - [pip\\_task\\_queue\\_init, 43](#)
  - [pip\\_task\\_queue\\_isempty, 44](#)
  - [pip\\_task\\_queue\\_lock, 44](#)
  - [pip\\_task\\_queue\\_trylock, 43](#)
  - [pip\\_task\\_queue\\_unlock, 44](#)
  - [pip\\_task\\_self, 57](#)
  - [pip\\_task\\_spawn, 19](#)
  - [pip\\_trywait, 26](#)
  - [pip\\_trywait\\_any, 27](#)
  - [pip\\_wait, 25](#)
  - [pip\\_wait\\_any, 27](#)
  - [pip\\_yield, 41](#)
  - [pip\\_yield\\_to, 42](#)
- Spawning PiP task, [17](#)
  - [pip\\_blt\\_spawn, 21](#)
  - [pip\\_spawn, 20](#)
  - [pip\\_spawn\\_from\\_func, 18](#)
  - [pip\\_spawn\\_from\\_main, 17](#)
  - [pip\\_spawn\\_hook, 18](#)
  - [pip\\_task\\_spawn, 19](#)
- Suspending and Resuming BLT/ULP, [46](#)
  - [pip\\_dequeue\\_and\\_resume, 48](#)
  - [pip\\_dequeue\\_and\\_resume\\_N, 49](#)
  - [pip\\_dequeue\\_and\\_resume\\_N\\_nolock, 50](#)
  - [pip\\_dequeue\\_and\\_resume\\_nolock, 49](#)
  - [pip\\_suspend\\_and\\_enqueue, 47](#)
  - [pip\\_suspend\\_and\\_enqueue\\_nolock, 48](#)
- Task Queue Operations, [42](#)
  - [pip\\_task\\_queue\\_count, 45](#)
  - [pip\\_task\\_queue\\_dequeue, 45](#)
  - [pip\\_task\\_queue\\_describe, 46](#)
  - [pip\\_task\\_queue\\_enqueue, 45](#)
  - [pip\\_task\\_queue\\_fin, 46](#)
  - [pip\\_task\\_queue\\_init, 43](#)
  - [pip\\_task\\_queue\\_isempty, 44](#)
  - [pip\\_task\\_queue\\_lock, 44](#)
  - [pip\\_task\\_queue\\_trylock, 43](#)
  - [pip\\_task\\_queue\\_unlock, 44](#)
- Terminating PiP Task, [23](#)
  - [pip\\_abort, 25](#)
  - [pip\\_exit, 24](#)
  - [pip\\_kill\\_all\\_tasks, 24](#)
- Waiting for PiP task termination, [25](#)
  - [pip\\_trywait, 26](#)
  - [pip\\_trywait\\_any, 27](#)
  - [pip\\_wait, 25](#)
  - [pip\\_wait\\_any, 27](#)
- Yielding Functionns, [41](#)
  - [pip\\_yield, 41](#)
  - [pip\\_yield\\_to, 42](#)