**Ptask Library: A Quick Guide**

Giorgio Buttazzo and Giuseppe Lipari

RETIS Lab - *Scuola Superiore Sant’Anna*

*Version 0.4*

Summary

**Ptask** is a C-library for fast development of periodic and aperiodic real-time tasks under Linux. It is written on top of the **Pthread** library with the aim of simplifying the creation of threads with typical timing parameters, like periods and deadlines. Ptask functions allow programmers to quickly

* create periodic and aperiodic tasks;
* specify timing constraints such periods and relative deadlines;
* monitor deadline misses;
* monitor average and worst-case execution times;
* activate tasks with specific offsets;
* manage task groups;
* handle mode changes.

Table of contents

This document includes the following sections:

1. New data types.
2. System functions
3. Task functions
4. Measuring execution times
5. Handling mode changes
6. Examples
7. New data types

The following new types are defined in the ptask library:

**ptime** this is the type used for the time variables. It is basically a shortcut for a **long** integer.

**tspec** this type is used for specifying a precise time, and it is used by the library for internal time representation. It is a shortcut for **struct timespec**. The Ptask library provides appropriate functions to operate on **tspec** and to convert a **ptime** into a **tspec**, and viceversa.

**tpars** this type of structure is used to store all task parameters and it is initialized at task creation.

**ptask** this type is used for defining the task code. It is a shortcut for **void**.

1. System functions

void ptask\_init(int scheduler, int schedtype, int protocol);

Initializes the **ptask** library, resets the system time, set the scheduler for all the tasks and the resource access protocol for all the semaphores.

scheduler can be SCHED\_OTHER, SCHED\_FIFO, SCHED\_RR, SCHED\_DEADLINE.

schedtype can be PARTITIONED or GLOBAL, and it is only useful for multicore systems.

protocol can be NO\_PROTOCOL for classical semaphores, INHERITANCE for Priority Inheritance, or CEILING for Immediate Priority Ceiling.

*Note (from version 0.4): in order to use SCHED\_DEADLINE in PARTITIONED mode, the user has first to disable the admission control in Linux. This can be done by invoking the script no-admission.sh (included in the distribution) with root privileges.*

ptime ptask\_gettime(int unit);

Returns the current time (from the system start time) in the specified unit, which can be SEC, MILLI, MICRO, or NANO.

int ptask\_getnumcores();

Returns the number of available cores in the system.

1. Task functions

The Ptask library maintains a Task Control Block (TCB) for every task, used to store the task state, the current task parameters, and some information collected about the task during its execution. The content of the TCB is for internal use only, therefore it is not reported in this manual. Please refer to the source code for more information.

int ptask\_create(void (\*body)(void), int period, int prio, int flag);

Creates a concurrent task and returns the task index that can be used to differentiate multiple instances of the same task. The arguments have the following meaning:

bodyis the name of the function containing the task body;

periodspecifies the task period (equal to the relative deadline) in milliseconds.

priospecifies the task priority between 1 (low) and 99 (high);

flagspecifies the activation mode of the task (NOW or DEFERRED): if set to NOW, the task is immediately activated; if set to DEFERRED, the task will block on the **wait\_for\_activation**() until a **ptask\_activate**() is invoked by another task.

If task creation cannot be performed or an error occurs, the function returns the value -1.

*Note (from version 0.4): this function has been deprecated since 0.4 because it is not compatible with the SCHED\_DEADLINE scheduler.*

**int ptask\_create\_prio**(**void** (\*body)(**void**), **int** period, **int** prio,

**int** aflag);

Creates a concurrent task and returns the task index that can be used to differentiate multiple instances of the same task. This function is supposed to be used with fixed priority scheduling (SCHED\_FIFO or SCHED\_RR) only. The arguments have the following meaning:

bodyis the name of the function containing the task body;

periodspecifies the task period (equal to the relative deadline) in milliseconds.

priospecifies the task priority between 1 (low) and 99 (high);

flagspecifies the activation mode of the task (NOW or DEFERRED): if set to NOW, the task is immediately activated; if set to DEFERRED, the task will block on the **wait\_for\_activation**() until a **ptask\_activate**() is invoked by another task.

If task creation cannot be performed or an error occurs, the function returns the value -1.

*Note: since version 0.4*

**int** **ptask\_create\_edf**(**void** (\*task)(**void**), **int** period, **int** runtime,

**int** dline, **int** aflag);

Creates a concurrent task for the SCHED\_DEADLINE scheduler, and returns the task index that can be used to differentiate multiple instances of the same task. The arguments have the following meaning:

bodyis the name of the function containing the task body;

periodspecifies the task period (equal to the relative deadline) in milliseconds;

**runtime** specifies the budget of the task in milliseconds, it must be less than the period.;

dline specifies the relative deadline of the task in milliseconds;

flagspecifies the activation mode of the task (NOW or DEFERRED): if set to NOW, the task is immediately activated; if set to DEFERRED, the task will block on the **wait\_for\_activation**() until a **ptask\_activate**() is invoked by another task.

If task creation cannot be performed or an error occurs, the function returns the value -1.

In the current version of Linux (4.4), there is no way to create a task so that it is immediately scheduled by SCHED\_DEADLINE: the only possibility is to create the task with some other scheduler (for example: SCHED\_OTHER), and then modify its scheduling parameters once the task has been created.

Since ptask uses the Linux API, function ptask\_create\_edf() will create the task in SCHED\_OTHER; when the task starts executing, it will change its scheduling parameters to SCHED\_DEADLINE. Therefore, it is recommended to leave the possibility for the task to get a chance to execute and go into SCHED\_DEADLINE during the initialization phase, for example by setting its activation flag to DEFERRED, and the activate the task with a certain offset in the future.

Also, if the SCHED\_DEADLINE parameters cannot be set, the user will not receive any error at the time of task creation, but the error will be communicated later when the task effectively tries to set its scheduling parameters.

void ptask\_activate(int tid);

Activates the task with index tid.

void ptask\_activate\_at(int tid, ptime t);

Activates the task with index tid at the absolute time t. If t has already passed, the task is immediately activated.

void ptask\_wait\_for\_period();

It suspends the calling task until the beginning of its next period. The typical usage of this call in a task body is shown in Figure 1.

void ptask\_wait\_for\_activation();

It suspends the calling task until an explicit activation is invoked by another task. The typical usage of this call in a task body is shown in Figure 2.

|  |
| --- |
| **ptask** my\_periodic\_task()  {  **int** i;  i = **ptask\_get\_index**();  **while** (1) {  <do useful things as a function of i>  **ptask\_wait\_for\_period**();  }  } |

Figure 1: General structure of a periodic task.

|  |
| --- |
| **ptask** my\_aperiodic\_task()  {  **int** i;  i = **ptask\_get\_index**();  **ptask\_wait\_for\_activation**();  **while** (1) {  <do useful things as a function of i>  **ptask\_wait\_for\_activation**();  }  } |

Figure 2: General structure of an aperiodic task.

The example illustrated in Figure 3 shows how to define a periodic task that starts executing upon an explicit activation.

|  |
| --- |
| **ptask** activated\_periodic\_task()  {  **int** i;  i = **ptask\_get\_index**();  **ptask\_wait\_for\_activation**();  **while** (1) {  <do useful things as a function of i>  **ptask\_wait\_for\_period**();  }  } |

Figure 3: General structure of a periodic task with an explicit activation.

The following functions are used to obtain and modify the parameters of a running task.

int ptask\_get\_index();

Returns the index of the calling task.

int ptask\_get\_priority(int i);

Returns the priority of the task with index i.

void ptask\_set\_priority(int i, int prio);

Sets the priority of the task with index i to the value specified by prio, which must be a value between 1 (the lowest priority) and 99 (the highest priority).

int ptask\_get\_period(int i, int units);

Returns the period of the task with index i (in units).

void ptask\_set\_period(int i, int myper, int units);

Sets the period of the task with index i to myper (in units).

ptime ptask\_get\_deadline(int i, int units);

Returns the relative deadline of the task with index i (in units).

void ptask\_set\_deadline(int i, int mydline, int units);

Sets the relative deadline of the task with index i to mydline (in units).

int ptask\_deadline\_miss();

Returns 1 if the current time is greater than the absolute deadline of the current job, 0 otherwise.

int ptask\_migrate\_to(int i, int core\_id);

Moves the task with index i to the core specified by core\_id.

More specific parameters can be passed to a task at creation time through the following structure:

|  |
| --- |
| typedef struct {  tspec runtime; // task budget (only for SCHED\_DEADLINE)  tspec period; // task period (in ms)  tspec rdline; // relative deadline (in ms)  int priority; // from 1 (low) to 99 (high) (not used in SCHED\_DEADLINE)  int processor; // processor where task should be allocated  int act\_flag; // activation flag (NOW, DEFERRED)  int measure\_flag; // enable/disable exec. time measurements  void \*arg; // pointer to a task argument  rtmode\_t \*modes; // pointer to the mode handler  int mode\_list[RTMODE\_MAX\_MODES]; // the maximum number of modes  int nmodes; // num of modes in which the task is active  } tpars; |

In particular:

runtime specifies the budget of the task (only for SCHED\_DEADLINE)

period specifies the task period in ms;

rdline specifies the task relative deadline in ms (by default it is set equal to period);

priority specifies the task priority between 1 (low) and 99 (high); this is only used for fixed priority scheduling (SCHED\_FIFO and SCHED\_RR);

processor specifies the processor where the task has to be allocated (default value is 0);

act\_flag if set to NOW, the task is immediately activated, if set to DEFERRED (default value), the task will block on the **wait\_for\_activation**() until a **ptask\_activate**() is invoked by another task;

measure\_flag if set to a non-zero the library automatically profiles the execution time of the task;

arg pointer to a memory area used to pass arguments to the task; the structure and the content of such a memory are user-defined;

modes used to manage mode changes (see Section 5);

nmodes number of modes of the task;

mode\_list list of task modes.

Such parameters can be set either directly or by using the following functions (for efficiency reasons, and following a common practice in C programming, these functions are actually implemented as macros).

void ptask\_param\_init(tpars tp);

Initializes the task parameters in the tp structure with the default values.

void ptask\_param\_period(tpars tp, int myper, int units);

Initializes the task period in the tp structure with the value specified by myper expressed in given units (SEC, MILLI, MICRO, or NANO).

void ptask\_param\_deadline(tpars tp, int mydline, int units);

Initializes the task relative deadline in the tp structure with the value specified by mydline expressed in given units (SEC, MILLI, MICRO, or NANO).

void ptask\_param\_priority(tpars tp, int myprio);

Initializes the task priority in the tp structure with the value specified by myprio, which must be a value between 1 (the lowest priority) and 99 (the highest priority).

void ptask\_param\_activation(tpars tp, int myact);

Initializes the task activation mode in the tp structure with the value specified by myact, which can be either NOW, for immediately activation, or DEFERRED; in this case the task will block on the **wait\_for\_activation**() until a **ptask\_activate**() is invoked by another task.

void ptask\_param\_processor(tpars tp, int proc\_id);

Specifies the index of the processor on which the task is supposed to run. Note that this is valid only if the PARTITIONED strategy has been set by **ptask\_init()**. This call has no effect when the scheduling strategy is set to GLOBAL.

void ptask\_param\_measure(tpars tp);

Sets the measuring flag to 1, so enabling **ptask** to automatically profile the execution time of the task.

void ptask\_param\_argument(tpars tp, void \*arg);

Passes to the tp structure the user-defined arguments pointed by arg.

void ptask\_param\_modes(tpars tp, rtmode \*modes, int nmodes);

Allows specifying the set of execution modes in the task that is going to be created. The argument modes is a pointer to a structure that defines the system modes and **nmodes** specifies the number of modes in which this task is going to be active.

void ptask\_param\_mode\_add(tpars tp, int mode\_num);

Specifies that the current task is active in mode mode\_num. See Section 5 for an example of use of this function.

Once all the specific parameters are set, the task can be created using the following function.

int ptask\_create\_param(void (\*body)(void), tpars \*tp);

Creates a concurrent task and returns the task index that can be used to differentiate multiple instances of the same task. The arguments have the following meaning:

body is the name of the function containing the task body;

tp is a pointer to the task parameter structure; if tp is set to NULL, then the task is created with the following default values:

type: APERIODIC

period 1000 ms

rdline 1000 ms

priority 1

act\_flag DEFERRED

processor 0

measure NO

arg NULL

modes NULL

nmodes 0

mode\_list an empty array

1. Measuring execution times

To measure the execution time of a task it is necessary to set its measure\_flag when the task is created. Then, after the task has completed its execution, it is possible to obtain its execution time by calling the following functions.

WARNING: calling these functions while the task is executing may give inconsistent values, because the internal data structures are not protected by semaphores for containing the overhead. Therefore, it is up to the user to make sure that the task is no executing before calling this function.

tspec ptask\_stat\_getwcet(int i)

Returns the maximum execution time among all the jobs of task i since its first activation.

tspec ptask\_stat\_getavg(int i)

Returns the average execution time among all the jobs of task i since its first activation.

int ptask\_stat\_getnuminstances(int i)

Returns the number of jobs of task i activated since its creation.

tspec ptask\_stat\_gettotal(int i)

Returns the total execution time consumed by all the jobs of task i since its creation.

1. Handling mode changes

The Ptask library allows the user to specify a set of execution modes for the whole system and handle mode changes transparently through the following functions.

int rtmode\_init(rtmode\_t \*g, int nmodes);

Initializes the mode manager and the data structure pointed by g, which will contain the task groups involved in each mode. The second parameter nmodes is the total number of system modes.

void rtmode\_changemode(rtmode\_t \*g, int new\_mode\_id);

This function has to be called every time we want the system to perform a mode change. The parameter new\_mode\_id specifies the new mode in which the system must switch. The mode change is performed by an internal mode manager that executes the mode change protocol. In the current implementation, before activating the tasks involved in the new mode, the mode manager waits for the largest absolute deadline of those tasks that are only present in the old mode. An example of mode change is shown in Figure 4.

|  |
| --- |
| **#define** MODE\_OFF 0  **#define** MODE\_ON 1  **#define** MODE\_FAIL 2  **ptask** taskbody()  {  **ptask\_wait\_for\_activation**();  **while** (1) {  printf("Task T%d is running\n", ptask\_get\_index());  **ptask\_wait\_for\_period**();  }  }  **int** main()  {  **rtmode\_t** mymodes;  **tpars** param;  **int** res;  **ptask\_init**(SCHED\_FIFO, GLOBAL, PRIO\_INHERITANCE);  res = **rtmode\_init**(&mymodes, 3); // System with 3 modes  **ptask\_param\_init**(param);  **ptask\_param\_period**(param, 1, SEC);  **ptask\_param\_priority**(param, 4);  // this task is present in two modes  **ptask\_param\_modes**(param, &mymodes, 2);  // the task is present in mode MODE\_ON  **ptask\_param\_mode\_add**(param, MODE\_ON);  // The task is present in mode MODE\_FAIL  **ptask\_param\_mode\_add**(param, MODE\_FAIL);  // The task is NOT present in MODE\_OFF  res = **ptask\_create\_param**(taskbody, &param);  // create the other tasks in a similar way  **rtmode\_changemode**(&mymodes, MODE\_OFF); // set initial mode  **if** (condition)  // activates MODE\_ON; all tasks not in this mode  // are suspended; after that, the tasks in MODE\_ON  // not already active, are activated  **rtmode\_changemode**(&mymodes, MODE\_ON);  **else if** (error\_condition)  // activates MODE\_FAIL; all tasks not in this mode  // are suspended; after that, the tasks in MODE\_FAIL  // not yet active, are activated  **rtmode\_changemode**(&mymodes, MODE\_FAIL);  ...  } |

Figure 4: example of mode change.

1. Handling exception on a deadline

The Ptask library allows the user to add a timer to each task which makes it a lot easier to stop a task that has overcome its deadline.

int dle\_timer\_start();

Arms the timer which will expire at the same time as the current task’s deadline.

int dle\_timer\_stop();

Disarms the current task’s timer and makes sure the timer won’t expire when there’s no need for it.

int dle\_chkpoint();

This function sets up a checkpoint which will be used if the task’s timer expires. This checkpoint is an underlying **sigsetjmp**. Upon timer expiration, the timer will send a signal to the task manager. Once the task manager receives this signal, it sends another signal to the corresponding task. This signal will be handled by the task. The default handler is an underlying **siglongjmp**. Therefore, it is important to make sure the **chkpoint** is set before any job starts.

int dle\_manager\_init();

This must be called right after **ptask\_init()** in the main. This makes sure the task manager which will receive the signals from the tasks’ timers is properly initialized.

|  |
| --- |
| **ptask** taskbody(int time\_to\_work, int unit)  {  **int** job = 0, idx = **ptask\_get\_index**();  **ptask\_wait\_for\_activation**();  **for** (;;job++) {  **if**(**dle\_chkpoint**() != 0) {  **ptask\_wait\_for\_period**();  **continue**;  }  **printf**("Task T%d starting: %d time\n", idx, job);  **dle\_timer\_start**();  **work\_for**(time\_to\_work, unit);  **dle\_timer\_stop**();  **printf**("Task T%d completed: %d time\n", idx, job);  **ptas****k\_wait\_for\_period**();  }  }  **ptask** task1()  {  taskbody(500, MILLI);  }  **ptask** task2()  {  taskbody(800, MILLI);  }  **static int** start\_task(**int** unit, **int** period, **int** deadline, **int** priority, **void** (\*task)(**void**))  {  **tpars** param;  **ptask\_param\_init**(param);  **ptask\_param\_period**(param, period, unit);  **ptask\_param\_deadline**(param, deadline, unit);  **ptask\_param\_priority**(param, priority);  **return ptask\_create\_param**(task, &param);  }  **int** main()  {  **int** res;  **ptask\_init**(SCHED\_FIFO, GLOBAL, PRIO\_INHERITANCE);  **dle\_manager\_init**();  res = **start\_task**(MILLI, 3000, 1500, 10, task1);  **printf**(“Created new task: %d\n”, res);  res = start\_task(MILLI, 2000, 2000, 5, task2);  printf(“Created new task: %d\n”, res);  **for**(;;);  **return** 0;  } |

Figure 5: example of a deadline exception handling.

In this section, two new structures were defined. One structure, is useful to keep information about tasks’ timer, this one is called **struct dle\_timer\_s.** Each task that needs to have a timer will be assigned one. The other structure is named **struct dle\_manager\_s** holds the task manager’s information and is meant to be instantiated only once.

A little more detailed description on the first structure :

|  |
| --- |
| struct dle\_timer\_s{  int dle\_timer\_signo; //defines the sigmask that the timer will have to correspond to, in order to throw an exception  pthread\_t dle\_timer\_threadid; //The timer is aimed to one specific task (thread)  int dle\_timer\_timerid; //This will be useful to arm / disarm the task’s timer  void (\*dle\_timer\_handler)(void); //handler executed upon timer expiration  }; |

A little more detailed description on the first structure :

|  |
| --- |
| struct dle\_manager\_s{  int dle\_manager\_tid //this will be used to feed a structure needed to create the tasks’ timers  pthread\_t dle\_manager\_threadid; //this is used by the function which arms the task’s timers  }; |