

LPKF-RTOS

Real time programming tools for DSPs
Version 1.1

User Manual

May 2003



© LPKF Motion & Control GmbH
Mittelbergstr. 17
98527 Suhl - Germany
Phone: +49 (0) 3681 - 8924 – 0
E-Mail: support@lpkf-mc.de
<http://www.lpkf-mc.de>

Introduction

LPKF-RTOS is a small real-time operating system with the following properties:

- minimal usage of memory (about 1kWords ROM, some 10-100 words of RAM depending upon the configuration)
- priority-controlled, preemptive execution of program threads
- modular, only the modules needed have to be included
- hardware independent, written in C
- statically configurable at compile time by the user.

About this Manual

It is assumed, that the reader is familiar with the concepts of real time programming, programming in C and with the concepts of hardware interrupts of the target processor.

When designing an RTOS, one has to make compromises between safety and execution speed. We decided to make LPKF-RTOS fast. That doesn't mean that the RTOS is not safe. However, we have decided that the user is responsible to call the RTOS routines in a right way. Please read the **CAUTIONs** carefully and follow the glue, and all will be ok.

Terms and Concepts

Program flow

After the program start, usually a reset of the processor, the hardware (IO-lines, timers, interfaces) must be initialized. The modules of the RTOS must be initialized too. This is usually done in the `main` () function. After initialization, the program enters an infinite loop (the IDLE-loop). Several events, triggered by hardware or software, interrupt this IDLE loop and start so called hardware or software interrupt routines (HWIs or SWIs).

Thread

A thread is a piece of program code that implements a certain task. 'thread' is a common term for HWIs, SWIs and the Idle-loop.

HWIs and SWIs must be programmed as finite threads, i.e., they have to execute a task which is finished within a more or less well-defined time. It is forbidden, to wait for events or for the release of resources within threads.

Priorities

Program threads can run as hardware or software interrupts with several priorities. Hardware interrupts have the highest priority, software interrupts have a higher priority than the programs executed in the Idle-loop. Threads of higher priority preempt the execution of threads with lower priority.

Deadlines

A real time computing system has to react to events in the real world. To ensure the proper operation of the whole system, the computer must have finished certain tasks at certain times. That means, to 'meet the deadlines'. Threads with a short deadline should have a high priority.

Scheduling

SWIs may be 'posted' using SWI system calls. If the SWI posted has a higher priority than the thread currently executed, the current thread is interrupted and the SWI with the higher priority is executed immediately. SWIs with lower priority than the current thread are notified for later execution. The execution of this SWI starts after all SWIs with higher priority have finished.

SWIs may be disabled by the user. While SWIs are disabled, higher priority SWIs do not interrupt the current thread until SWIs are enabled again. (The SWIs are scheduled with the re-enabling of SWIs.) A currently active (executing) SWI thread may be posted again. This does not preempt the thread. Instead, the thread is re-started after it has finished its execution.

Stack

All threads (HWIs, SWIs and the IDLE-loop) use the same stack. Since in the worst case, all the threads may be active, i.e. interrupted by threads with higher priority, stack size has to be dimensioned for this case. Also keep in mind the stack usage of the arguments and local variables of the C functions called.

Objects

Each module (SWI, PIP, PRD) may contain several objects. The number of objects is defined statically with the configuration of the system. The objects contain data necessary to control the program flow within RTOS. The user may access the objects via the API functions.

Note: SWIs and PRD objects are used to call functions. The function called is *not* the object. PIP objects handle a buffer. The buffer is *not* the PIP object.

User function

Several modules of the RTOS allow to start user functions that are connected to an object if a certain condition is met. This function is provided by the user, it must be programmed as a finite thread.

Pipes

Pipes are used as a means of a buffered message exchange between two processes, where one is the writer, the other is the reader of the pipe.

Structure of LPKF-RTOS

Target hardware dependencies

LPKF-RTOS is nearly hardware independent and intended to run on several target hardware configurations. Currently, there is a implementation for TMS320C24x (TI 16 bit DSP for motor controller applications) and a simulation (no HWIs) on a PC compiled with both MSVC and gcc. The target system is defined in the file `targets.h`. Several hardware dependencies are defined in the file `rtos.h`.

Modules

LPKF-RTOS provides the following modules:

HWI	-	routines for hardware-interrupts using other modules of the RTOS
SWI	-	software-interrupts for priority controlled, preemptive execution of programs
PRD	-	periodic and timer controlled execution of programs
MEM	-	dynamic memory allocation
PIP	-	pipes for interprocess communication

Files of a Module

There are the following files for each module (xxx stands for the module name):

<code>xxx.h</code>	- C include with the definitions of the API functions
<code>xxx.c</code>	- Implementation of the API functions
<code>xxxcfg.h</code>	- C include with the configuration defined by the user
<code>xxxcfg.c</code>	- Configuration data defined by the user.

There are two template files for each module: `xxxcfg.ct1` (the template for the `xxxcfg.c` file) and `xxxcfg.ht1` (the template for the `xxxcfg.h` file). These templates have to be copied to the respective `xxxcfg.c` and `xxxcfg.h` files and then to be modified to configure the RTOS for the application.

The HWI module consists of the file `hwi.h` only.

Dependencies

The HWI module also needs the SWI module, it makes no sense to use HWI functions if there are no SWIs. If you don't need SWIs, you probably won't need the RTOS.

The SWI module needs the HWI module. The PRD module and the PIP module are independent from each other. If SWI calls are configured in the PRD module or in the PIP module, the SWI and HWI module are needed.

Version 1.1

With version 1.1, there is a new PIP module with extended functionality. This PIP module needs the MEM module. However, the MEM module may be used without the PIP module.

Module HWI – Hardware Interrupts

Hardware interrupt service routines (ISRs) are triggered by hardware events, its implementation depends on the target system. ISRs have to be programmed by the user, there is no support by the RTOS, and ISRs may be completely independent from RTOS. However, LPKF-RTOS provides some means necessary to call functions of the other RTOS modules out from hardware triggered ISRs.

API functions (overview)

disable & enable hardware interrupts	<code>void hwi_disable(void);</code> <code>void hwi_enable(void);</code>
routines for hardware triggered ISRs that use RTOS functions	<code>void hwi_enter(void);</code> <code>void hwi_exit(void);</code>

API functions (description)

```
void hwi_disable(void)
```

Globally disable hardware interrupts.

```
void hwi_enable(void)
```

Globally enable hardware interrupts

hwi_disable() and **hwi_enable()** are useful to surround statements that modify critical data, i.e. data, that are used by different threads. Note, that this affects the interrupt latency of the whole system, so you should surround only short sequences of statements. Several RTOS functions itself use **hwi_disable()** and **hwi_enable()**. Do not surround any RTOS calls with **hwi_disable**.

```
void hwi_enter(void)
```

To be called in hardware interrupt routines that use other modules of the RTOS. Must be called before any other calls to RTOS routines in the procedure. Disables software interrupts. The call enables hardware interrupts, so time critical processing within the HWI could be done before calling **hwi_enter()**.

```
void hwi_exit(void)
```

To be called in hardware interrupt routines that use other modules of the RTOS. Must be called after any other calls to RTOS routines in the procedure. Enables software interrupts and schedules SWIs triggered from the HWI.

hwi_enter() and **hwi_exit()** must be called in HWI routines that call SWI routines, the **prd_tick()** function or pipe functions, that indirectly call SWIs. SWIs posted within a HWI are not scheduled immediately, they are scheduled within the **hwi_exit()** call.

CAUTION: A hardware interrupt procedure that calls `hwi_enter()` and `hwi_exit()` must have saved all the registers used by the C Compiler (and restore them on exit). There should be no problem if the hardware interrupt is a C function declared as `interrupt`. If the hardware interrupt is written in assembler, the user is responsible for the saving and restoring of the registers.

CAUTION: If a hardware interrupt procedure re-enables interrupts, and it is interrupted by a HWI that calls SWIs (i.e., uses `hwi_enter` and `hwi_exit`), its code is executed **after** all SWIs posted in subsequence of the SWIs posted in the other interrupt routine. To avoid this, you may:

- let interrupts globally disabled (useful for fast HWI routines only)
- use itself `hwi_enter()` and `hwi_exit()`.

Note, that it is not allowed in HWIs to disable and re-enable SWIs using `swi_disable()` and `swi_enable()`.

Configuration

None.

Example

The example shows the usage of the `hwi` module functions in a simple application that maintains a system clock.

```
#include "hwi.h"
#include "prd.h"

#define TIMER_PERIOD 10

/* system time */
static volatile int time;

/* hardware interrupt routine triggered periodically by a timer */
interrupt void timer_isr()
{
    /* disable execution of SWIs */
    hwi_enter();
    /* trigger PRD module clock (may post SWIs) */
    prd_tick();
    /* increment time */
    time += TIMER_PERIOD;
    /* continue with isr */
    do_other_time_critical_processing();
    /* enable SWIs and schedule */
    hwi_exit();
}
```

```

/*
time correction routine that adjusts the time according to a
time difference provided by a comparison with a master clock
(running e.g. in a SWI)
*/

void
do_time_correction(int time_diff)
{
    /* disable HWIs to prevent modification of time in timer_isr */
    hwi_disable();
    time += time_diff;
    /* re-enable HWIs */
    hwi_enable();
}

/*
The local variable 'time' is modified both by timer_isr and
do_time_correction. If we don't disable HWIs when doing the
time correction, the following may happen: The variable time
is loaded in do_time_correction, the time_diff is added, and
do_time_correction may be interrupted by timer_isr. Here, the
time is incremented by TIMER_PERIOD and stored again. But,
after return of the HWI, the old - corrected - time is stored,
and the increment by timer_isr is lost. This will result in a
time error of TIMER_PERIOD.
Also, 'time' must be declared volatile to ensure the correct
access to this variable.
*/

```

Module SWI – Software Interrupts

Software interrupts (SWIs) allow a priority-controlled, preemptive scheduling of user defined functions. See the 'Terms and Concepts' section.

SWI mailboxes

Each SWI routine has an `int` Variable called 'mailbox'. The initial value of the mailbox is set during initialization. The mailbox value can be modified by the posting calls to signalize several conditions when the SWI is posted.

When a SWI is scheduled for execution, the mailbox value is copied and the mailbox value is reset to its initial value for further postings. The copy of the mailbox value the SWI was posted with can be accessed by the user function using `swi_getmbox()`.

A SWI may be re-posted while it is running (it is the current thread). This does not interrupt this thread, and it does not affect the mailbox value of this thread (returned by `swi_getmbox()`). Instead, the current thread finishes execution and the SWI is scheduled again with the new mailbox value.

API functions (overview)

SWI module initialization	<code>void swi_init(void);</code>
disable / enable SWI scheduling	<code>void swi_disable(void);</code> <code>void swi_enable(void);</code>
SWI posting calls	<code>void swi_post(swi_obj *);</code> <code>void swi_inc(swi_obj *);</code> <code>void swi_dec(swi_obj *);</code> <code>void swi_or(swi_obj *, int);</code> <code>void swi_andn(swi_obj *, int);</code>
obtain information about current SWI	<code>int swi_getmbox(void);</code> <code>swi_obj *swi_self(void);</code>

API functions (description)

```
void swi_init(void)
```

Initializes the SWI module. Must be called before using any other SWI function.

```
void swi_disable(void)
```

Disables SWIs. This prevents the preemption of the running thread by higher prioritized SWIs until SWIs are enabled again. Does not affect HWIs.

```
void swi_enable(void)
```

Enables SWIs again.

`swi_disable()` and `swi_enable()` are useful to surround statements that modify critical data, i.e. data, that are used by different threads. Note, that this affects the latency of SWIs, so you should surround only short sequences of statements. Since HWIs remain enabled, `swi_disable()` and `swi_enable()` are less critical than `hwi_disable()` and `hwi_enable()`.

CAUTION: For each `swi_disable()` there must be exactly one `swi_enable()`.

```
void swi_post(swi_obj *)
```

Unconditionally posts a SWI object. Mailbox value is not affected.

```
void swi_inc(swi_obj *)
```

Increments the mailbox value and unconditionally posts the SWI. Useful to indicate multiple postings of the same SWI. The initial mailbox value should be 0 in this case.

```
void swi_dec(swi_obj *)
```

Decrements the mailbox value and posts the SWI if the resulting mailbox value is 0. Useful to post a SWI after a certain number of events. The initial mailbox value should be greater 0 in this case.

```
void swi_or(swi_obj *, int flag)
```

ORs the mailbox value with `flag` (`mailbox_value |= flag`) and unconditionally posts the SWI. Useful to indicate which event (out of several ones) is the reason for the posting of the SWI. The initial mailbox value should be 0 in this case.

```
void swi_andn(swi_obj *, int flag)
```

ANDs the mailbox value with the bitwise complement of `flag` (`mailbox_value &= ~flag`) and posts the SWI if the resulting mailbox value is 0. Useful if multiple conditions must be met before the SWI may be posted. The initial mailbox value should be the OR of all the conditions to meet in this case.

```
int swi_getmbox(void);
```

Returns the mailbox value of the currently executing SWI. Returns 0 if called from the IDLE loop. Calling from a HWI makes no sense (yields the mailbox value of the interrupted SWI or 0).

```
swi_obj *swi_self(void);
```

Returns the pointer to the SWI currently executed. Returns `NULL` if called from the IDLE loop. Useful to repost the currently executing SWI again. Calling from a HWI makes no sense (yields the interrupted SWI or `NULL`).

CAUTION: One may have the strange idea to call the user function connected to a SWI object directly (given, that it is reentrant). However, this is completely different to the posting mechanism provided by the SWI module. A direct call gives control immediately to the called function, a 'posting' of a user function via SWI schedules the function according to the priority of the SWI object the user function is connected to.

Configuration

The configuration data of a SWI object contains:

- the priority of the SWI object
- the initial mailbox value
- the user function to be called when the SWI is scheduled.

Configuration must be defined by the user in the files `swicfg.c` and `swicfg.h`. One may copy the template files `swicfg.ct1` and `swicfg.ht1` to `swicfg.c` and `swicfg.h`, respectively, and modify them according to the configuration wanted.

CAUTION: Exactly follow the glue given in the templates. There is no check if your configuration is correct, and an incorrect configuration may lead to unpredictably wrong program execution.

Example

The example shows some simple SWI postings.

```
#define THIS_SWI_FLAG 0x0001
#define THAT_SWI_FLAG 0x0002
```

```
a_function()
{
    swi_or(swi_this_or_that, THIS_SWI_FLAG);
    /* swi is posted, mailbox value = 1 */
    swi_or(swi_this_or_that, THAT_SWI_FLAG);
    /* swi is posted again */
    swi_andn(swi_this_and_that, THIS_SWI_FLAG);
    /* mailbox value = 2, swi not posted */
    swi_andn(swi_this_and_that, THAT_SWI_FLAG);
    /* mailbox value = 0, swi is posted */
}
```

```
/*
```

```
If swi_this_or_that has a higher priority than the thread executing
a_function, the user function of this SWI is executed twice, first
with mailbox = 1, then with mailbox = 2.
```

```
If swi_this_or_that has a lower priority than the thread executing
a_function, the user function of this SWI is executed once with a
mailbox value = 3 (= THIS_SWI_FLAG | THAT_SWI_FLAG).
```

```
The initial mailbox value of the swi_this_and_that must be set to
(THIS_SWI_FLAG | THAT_SWI_FLAG) to get the SWI posted after THIS and
THAT has happened.
```

```
*/
```

Module MEM – dynamic memory management

The MEM module allows the dynamic allocation of memory. It is in some aspects similar to the `malloc()`, `calloc()`, and `free()` functions provided by the C standard library, but there are some major differences. The MEM module manages several memory pools (memory objects). Each memory object manages a number of memory frames with a fixed size. Each `mem_alloc()` call allocates a frame with a fixed size. To allocate frames with different sizes, one has to use several memory objects. This has the disadvantage of wasting some memory cells, however, it prevents the fragmentation of the dynamic memory. (Fragmentation may cause `malloc()` to fail, even if there are some free memory cells. This is not acceptable for a real time system.) Also, the management of frames with a fixed size is simpler and faster.

The MEM module was mainly intended to support the new PIP module in version 1.1. However, you may use a memory object not used by the PIP module for your own purposes.

API functions (overview)

initialization	<code>void mem_init(void);</code>
allocation / free functions	<code>void *mem_alloc(mem_obj *);</code> <code>void mem_free(mem_obj *, void *);</code>
information functions	<code>int mem_get_framesize(mem_obj *);</code> <code>int mem_get_numframes(mem_obj *);</code>

API functions (description)

```
void mem_init(void)
```

Initializes the memory module. Must be called before using any other functions of the memory module.

```
int mem_get_numframes(mem_obj *);
```

Returns the total number of frames in the memory object.

```
int mem_get_framesize(mem_obj *);
```

Returns the size of a frame of the memory object in MAU units. A MAU is the minimum addressable unit, e.g. a byte or a word, depending upon the architecture of the processor.

```
void *mem_alloc(mem_obj *)
```

Allocates a frame from the memory pool of this memory object and returns a pointer to the allocated frame or NULL if no space is available. The returned pointer may be casted to any type. `mem_get_framesize(mem_obj *)` MAUs may be written to this frame.

```
void mem_free(mem_obj *mem, void *p);
```

Returns a frame `p` back to the memory pool of that `mem` object.

CAUTION: Do not write more than `framesize` MAUs to a frame reserved with `mem_alloc()`. Otherwise, the memory management will be corrupted, the system may crash.

CAUTION: Do not call `mem_free()` with a pointer `p` not obtained from that memory object. Do not call `mem_free()` more than once with the same pointer `p`. Otherwise, the memory management will be corrupted, the system may crash.

CAUTION: Do not use a memory object that is used by pipes. Otherwise, the memory management will be corrupted, the system may crash.

For each successful `mem_alloc()`, there should be a `mem_free()`. Free the frames no longer used as soon as possible to allow a subsequent allocation of that frame.

Configuration

The configuration data of a MEM object contains:

- the total number of frames in the memory pool
- the size of a frame in the memory pool
- a pointer to number of frames * size of a frame MAUs (the memory pool).

Configuration must be defined by the user in the files `memcfg.c` and `memcfg.h`. One may copy the template files `memcfg.ct1` and `memcfg.ht1` to `memcfg.c` and `memcfg.h`, respectively, and modify them according to the configuration wanted.

CAUTION: Exactly follow the glue given in the templates. There is no check if your configuration is correct, and an incorrect configuration may lead to unpredictably wrong program execution.

Example

The (very simple) example allocates two memory frames and frees them in reversed order.

```
#include "memcfg.h"

/* it is assumed that the configuration defines a memory object 'mem'
with at least two frames that may contain an integer variable */

main()
{
    int *p1, *p2;

    /* initialize memory module */
    mem_init();
    /* try to allocate a first frame */
    if ( (p1 = (int *)mem_alloc(mem)) != NULL ) {
        /* OK, write to allocated memory */
        *p1 = 1;
    }
    /* try to allocate a second frame */
    if ( (p2 = (int *)mem_alloc(mem)) != NULL ) {
        /* OK, write to allocated memory */
        *p2 = 2;
    }
    /* free the allocated frames in reversed order */
    if ( p2 != NULL ) {
        mem_free(mem, p2);
    }
    if ( p1 != NULL ) {
        mem_free(mem, p1);
    }
}
```

Module PIP – Pipes

Pipes may be used to exchange messages between several threads. A pipe is an unidirectional communication channel with a writer (generator of the messages) at the one end and a reader (consumer of the messages) at the other end. A pipe contains a number of frames that may contain messages that are sent from one process (writer) to another process (reader). The buffering of data may be used for synchronisation of the program execution with other events.

Pipes are linked to a memory object that contains the memory for the messages. Several pipes may share the same memory object. Each pipe can have a number of reserved memory frames, i.e., these frames of the memory object are reserved for the exclusive use by that pipe. The sum of all reserved frames of all pipes that share a memory object must be less or equal than the total number of frames in that memory object. If the sum of the reserved frames is less than the total number of frames, the rest of the frames are so called 'shared' frames, that are allocated to the pipe whichever comes first.

From the point of view of a message in a frame it's life cycle is as follows:

- some memory gets allocated for it using `pip_alloc()`
- the message is created writing some data to the allocated frame
- the message is sent using a `pip_put()`
- the message becomes received by a `pip_get()`
- the message is read and processed
- the message may be moved to another pipe using a `pip_move()`, this means a `pip_free()` in the sending pipe and a `pip_alloc()` / `pip_put()` in the receiving pipe, where it again will be received by a `pip_get()`, etc.
- the message dies after the final receiver has processed it, the used memory is returned back using `pip_free()`.

API functions (overview)

initialization and start	<code>void pip_init(void);</code> <code>void pip_start(void);</code>
writer functions	<code>void *pip_alloc(pip_obj *);</code> <code>void pip_put(pip_obj *, void *);</code>
reader functions	<code>void *pip_get(pip_obj *);</code> <code>void *pip_move(pip_obj *, pip_obj *, void *);</code> <code>void pip_free(pip_obj *, void *);</code>
information functions	<code>int pip_get_numframes(pip_obj *);</code> <code>int pip_get_resframes(pip_obj *);</code> <code>int pip_get_framesize(pip_obj *);</code> <code>int pip_get_reader_frames(pip_obj *);</code> <code>int pip_get_writer_frames(pip_obj *);</code> <code>int pip_get_used_frames(pip_obj *);</code>

API functions (description)

```
void pip_init(void)
```

Initializes the Pipe module. Must be called before using any other functions of the pipe module. The MEM module must be initialized before the PIP module is initialized.

```
void pip_start(void);
```

Starts the pipe module by calling the `notify_writer` function of each pipe object.

CAUTION: If SWIs or hardware devices are involved in the message exchange mechanism, they must be initialized before `pip_start()` is called.

```
int    pip_get_framesize(pip_obj *);
```

Returns the size of a frame of the pipe object in MAU units. A MAU is the minimum addressable unit, e.g. a byte or a word, depending upon the architecture of the processor. (The framesize is inherited from the memory object used by the pipe.)

```
int    pip_get_numframes(pip_obj *);
```

Returns the total number of frames the pipe can manage. Since memory frames may be shared by several pipes, it is not guaranteed that this number of frames can be used by the pipe.

```
int    pip_get_resframes(pip_obj *);
```

Returns the number of memory frames that are reserved for exclusive use by the pipe.

```
int    pip_get_writer_frames(pip_obj *);
```

Returns the number of frames in the pipe that are available for the writer and may be obtained by the `pip_alloc` function call. If `pip_get_writer_frames` returns a value $n > 0$, n subsequent `pip_alloc` calls will be successful. Actually, `pip_get_writer_frames` returns the difference `reserved_frames - used_frames`. A return value less than 1 does not mean, that there are no frames available, however, it is not guaranteed that a `pip_alloc` will succeed (this depends on how many shared memory frames and free frames in that pipe are available).

```
int    pip_get_reader_frames(pip_obj *);
```

Returns the number of frames in the pipe that are available for the reader and may be obtained by the `pip_get` function call. If `pip_get_reader_frames` returns a value $n > 0$, n subsequent `pip_get` calls will be successful.

```
int    pip_get_used_frames(pip_obj *);
```

Returns the total number of frames that are in use by the pipe (i.e., the number of frames that are allocated, but not freed again).

```
void *pip_alloc(pip_obj *)
```

Allocates a frame from the pipe and returns a pointer to the allocated frame or `NULL` if no space is available. Usually called by the writer of the pipe if it wants to put a message into the pipe. The returned pointer may be casted to the type of the message. A `NULL` return value indicates either a full pipe or a full memory.

```
void pip_put(pip_obj *, void *p);
```

Puts a message into the pipe and calls the `notify_reader()` function of the pipe object. Called by the writer of the pipe after it has allocated a frame (using `p = pip_alloc()`) and put some data into the frame.

```
void *pip_get(pip_obj *);
```

Get a frame from a pipe. Returns a pointer to the frame or NULL if no frame is available to read. Usually called by the reader of the pipe. The returned pointer may be casted to the type of the message.

```
void pip_free(pip_obj *, void *p);
```

Returns a frame obtained using pip_get() back to the pipe and calls the notify_writer() function of the pipe object. Called by the reader of the pipe to signalize that the data of a frame have been read and are no longer used.

```
void *pip_move(pip_obj *to, pip_obj *from, void *p);
```

Moves a frame *p* with a message (obtained using pip_get()) from the *from* - pipe to the *to* - pipe. Actually, a pip_move() does a pip_alloc() and a pip_put() in the *to* - pipe, and a pip_free() for the *from* - pipe. However, there is no memory allocated and freed like in pip_alloc() and pip_free(), only a pointer to the message is moved. The notify_reader() function of the *to* - pipe and the notify_writer() function of the *from* - pipe are called. The pip_move() function is useful to redirect or distribute messages to several pipes that share the same memory pool.

pip_move() returns the pointer to the frame *p* on success or NULL if there is no space in the *to* - pipe, if it is impossible to get a shared memory frame (if a reserved frame would become a shared one), or, if the pipes do not share the same memory pool (this is a programming error).

CAUTION: Writer has to take care, that the messages written to the pipe are not longer than the framesize value. Otherwise, subsequent messages and, even worse, the memory management will be corrupted. The writer may send shorter messages anyway, if this is suitable to the reader. Implementation is left to the user, e.g. one may write the length of the message into the message itself.

CAUTION: For each successful pip_alloc(), there must be a pip_put(). And, for each successful pip_get(), there must be a pip_free(). Otherwise, the pipe may be locked, illegal data may be read or other terrible things may happen. (For reasons of execution speed, there is no protection implemented.)

It is not necessary, to call pip_put() after each pip_alloc(), or, to call pip_free() after each pip_get(). (Although, this is the most simple case which will be used in most of the cases.) Writer or reader may have more than one frame 'in work'. However, they must know (e.g. by counting the calls), how many frames they occupy and they have to maintain the pointers to the data frames itself.

Note, that the order the messages are sent is determined by the order of the pip_put() calls. (E.g., several frames may be allocated, and the frame last allocated may be sent first. In this case, the user function has to manage all the pointers of the allocated and not yet put messages.) Also, it is possible to free frames in a different order that they were get.

Configuration

The configuration data of a PIP object contains:

- the total number of frames in the pipe
- the number of reserved memory frames for that pipe (at least one, but less than or equal to the total number of frames), the sum of all reserved frames must be less or equal to the total number of frames of the used memory object
- the memory object
- a pointer to a management vector
- the notify_reader() function and the two arguments it is called with
- the notify_writer() function and the two arguments it is called with.

Configuration must be defined by the user in the files `pipcfg.c` and `pipcfg.h`. One may copy the template files `pipcfg.ctl` and `pipcfg.htl` to `pipcfg.c` and `pipcfg.h`, respectively, and modify them according to the configuration wanted.

CAUTION: Exactly follow the glue given in the templates. There is no check if your configuration is correct, and an incorrect configuration may lead to unpredictably wrong program execution.

Reserved / Shared frames

The total number of messages that are 'alive' in a pipe system is limited by the number of frames in the memory object shared by the pipes.

Every pipe must have a number of reserved frames. Otherwise, since there is no prioritisation when shared frames are allocated, a deadlock may occur.

If all the memory frames are reserved by a pipe, a lot of memory is used. Having shared frames gives a better flexibility. Actually, it means something like a variable length of the pipes – a 'busy' pipe can allocate some more frames, if the other pipes are 'not so busy'.

Example

The example shows the usage of pipes and SWIs to receive command messages via a serial interface and to execute the command if it is received.

```
#include "pipcfg.h"
#include "swicfg.h"

/* receiver hardware interrupt routine (writes to the pipe) */
void interrupt rx_isr(void)
{
    MSG *msg;

    hwi_enter();
    receive_some_bytes();
    if ( pip_get_writer_frames(a_pipe) > 0 ) {
        /* allocate a frame */
        msg = (MSG *)pip_alloc(a_pipe);
        /* write a message into the frame */
        write_bytes_to_msg(msg);
        /* make frame available to the reader */
        pip_put(a_pipe, msg);
    }
    else
        /* no frames available in the a_pipe */
        do_error_handling();
    hwi_exit();
}

void a_pipe_reader()
{
    MSG *msg;

    while ( pip_get_reader_frames(a_pipe) > 0 ) {
        /* get message from the pipe */
        msg = (MSG *)pip_get(a_pipe);
        /* react to the message */
        do_something_with(msg);
        /* message is no longer needed, return frame to pipe */
        pip_free(a_pipe, msg);
    }
}
```

Module PRD – Timer controlled activation of program threads

The period module maintains an internal clock. This clock must be triggered by a function, usually a hardware timer interrupt, which must be provided by the user.

Period objects start its user function at a certain clock tick. Period objects may be periodic or one-shot. Periodic objects, if once started, remain active until they are stopped explicitly. One-shot period objects are executed once, then they deactivate itself and must be restarted.

The period clock is a long variable that overflows periodically every $2^{(\text{bits_of_a_long_var})}$. If, e.g. the period clock is triggered every 200 μs , the overflow occurs every $200 \times 10^{-6} \times 2^{32}$ seconds, i.e. 238.6 hours or about 9 days & 10 hours.

Therefore, any time tick is in the future. It is not possible, to trigger events after the overflow period. This is similar to an 12-hour alarm clock, that won't awake you in 13 hours, but in one hour.

API functions (overview)

initialization	<code>void prd_init(void);</code>
clock tick and triggering of functions	<code>void prd_tick(void);</code>
get / set clock time	<code>long prd_get_time(void);</code> <code>void prd_set_time(long time);</code> <code>void prd_adjust_time(long time_diff);</code>
get / set period of an object	<code>long prd_get_period(prd_obj *obj)</code> <code>void prd_set_period(prd_obj *obj, long period);</code>
start / stop objects	<code>void prd_start_at(prd_obj *obj, long time)</code> <code>void prd_start_now(prd_obj *obj)</code> <code>void prd_stop(prd_obj *obj)</code>

API functions (description)

```
void prd_init(void)
```

Initializes the period module. The period clock is set to -1 , so that the next call of `prd_tick()` will yield a period clock time of 0.

```
void prd_tick(void)
```

Increments the period clock by 1. Checks if there are any period objects to trigger at this clock tick. If so, executes the user function of the period objects that have to be triggered and, if the period object is periodic, computes the next trigger clock time, otherwise deactivates the (one-shot) object.

```
long prd_get_time(void)
```

Returns the actual value of the period clock.

```
void prd_set_time(long time)
```

Sets the period clock to `time` with the next call of `prd_tick()`.

CAUTION: Instead of changing the period clock immediately, the clock value is updated with the next call of `prd_tick()`. To ensure that no activations of period objects get lost, the start times of all objects are changed, too.

```
void prd_adjust_time(long time_diff)
```

Adjusts the period clock by `timediff` (`prd_time += time_diff`) with the next call of `prd_tick()`. To ensure that no activations of period objects get lost, the start times of all objects are changed, too.

```
long prd_get_period(prd_obj *obj)
```

Returns the period in clock ticks of the period object. Returns 0 for a one-shot period object.

```
void prd_set_period(prd_obj *obj, long period)
```

Sets the period of a period object to `period`.

```
void prd_start_at(prd_obj *obj, long start_time)
```

Sets the next trigger point of the period object to `start_time`.

CAUTION: Do not use this function to start a function at the next clock tick. Enclose the call in `swi_disable()` / `swi_enable()` calls if you are in a low prioritized SWI. Otherwise, the clock tick may have gone, and the object will be triggered after a clock overflow.

```
void prd_start_now(prd_obj *obj)
```

Activates a period object and sets the trigger of it to the next clock tick. Note, that the user function is not executed in the thread calling this function, but with the next tick of the timer.

```
void prd_stop(prd_obj *obj)
```

Deactivates a period object. The object remains inactive until it is restarted.

CAUTION: Active period objects may be postponed or preponed by calling `prd_start_at()` or `prd_start_now()` again. When calling `prd_start_at()`, the user has to ensure that the start time is in the (near!) future. Otherwise, the function will be called a *little* bit later.

Configuration

The configuration data of a PRD object contains:

- the initial state of the active flag
- the period of the object (0 if the object is one-shot)
- the first activation time
- the user function and the two arguments it is called with

Further, there is a variable `prd_us_per_tick` that may be set.

CAUTION: The user functions of period objects should be very short ones. All these functions should be executed until the next tick occurs. If there are lengthy computations to execute, the period object should post a software interrupt to execute them. On the other hand, it is more effective to start short user functions directly by the period module than to post a SWI.

Example

The example shows the usage of a period object `prd_timeout` to do a timeout handling. The sender of a message awaits an acknowledge within a certain time specified by `TIMEOUT_DELAY`. The sender starts a period object, which is stopped if the acknowledge was received. If no acknowledge was received within the specified time, the period objects starts the `timeout()` function to do some error handling. The object must be configured as inactive and one-shot.

```
#include "prd.h"

#define TIMEOUT_DELAY 10

send_a_message()
{
    send_the_message();
    /* activate the timeout period object */
    prd_start_at(prd_timeout, prd_get_time() + TIMEOUT_DELAY);
}

receive_acknowledge()
{
    receive_the_acknowledge();
    /* deactivate the timeout */
    prd_stop(prd_timeout);
}

timeout()
{
    /* this is the user function of the prd_timeout object */

    /*
    no acknowledge was received within the TIMEOUT_DELAY,
    the period object has 'fired' (and is inactive now, since
    it is a one-shot).
    */
    do_some_error_handling();
}
```

Appendix A - LPKF RTOS on TI TMS320C24xx DSPs

How to get Projects running

The distribution comes with several example projects for the TMS320F2407 Evaluation Module by Spectrum Digital Inc. To compile and run them open the projects in the TI Code Composer. There are both .mak and .pjt files in the directories rtos/examples/*.

To include LPKF RTOS in your own projects, the following must be done:

- generate the configuration files XXXcfg.c and XXXcfg.h for each module XXX of the LPKF RTOS you use. You may copy the templates in rtos/templates in the source directory of your project and rename them.
- add the rtos include directory rtos/include to the include path of the compiler
- include the library rtos/lib/rtos2xx.lib into your project
- the run time library rts2xx.lib must be adapted as described below.

Adapt the run time library (rts2xx.lib)

The LPKF RTOS expects that the .const section in data ram is initialized correctly. If C autoinitialization at run time is used (-c linker option), the boot routine in the run time library rts2xx.lib must be changed to copy the .const section from ROM to RAM.

Change to the directory where rts2xx.lib resides:

```
cd C:\tic2xx\c2000\cgttools\lib
```

Save original run time library to rts2xx.orig (or any other file):

```
copy rts2xx.lib rts2xx.lib.orig
```

extract boot.asm from the library source file:

```
dspar -x rts.src boot.asm
```

edit boot.asm using any text editor, change the CONST_COPY option to 1:

```
CONST_COPY .set 1
```

reassemble boot.asm:

```
dspar -v2xx boot.asm
```

insert the object file boot.asm in the run time library:

```
dspar -r rts2xx.lib boot.obj
```

Alternatively, you may replace boot.obj in a copy of the original rts2xx.lib, replace boot.obj in this file and link this copy instead of the original rts2xx.lib.

Use a section definition like

```
.const : load = PROG PAGE 0, run = DATA PAGE 1
{
    __const_run = .;
    *(.c_mark)
    *(.const)
    __const_length = . - __const_run;
}
```

in your linker command file *.cmd.

Appendix B - porting LPKF RTOS to other processors / known problems

LPKF RTOS is nearly completely written in C. It was developed for TI TMS 320C24xx DSPs, however, it may be ported to other DSPs or microprocessors, too. The target processor is selected by a C compiler option (-dTMS320C2xx for C2xx DSPs). All target processor dependent definitions are in the file rtos.h. However, there may be some problems:

Passing arguments by RTOS to user functions

The user functions called by the PRD and the PIP module are of the type `void function(arg, arg)`, where `arg` may be a pointer of any type (`void *`) or an integer (`int`). This works fine if the C compiler passes these two types of arguments in the same manner, e.g. via the stack. Problems will arise if pointer and ints are passed in a different way. E.g., this is the case for the TI C28xx compiler.

Memory alignment

The MEM module uses (implicitly) a construction like

```
struct {
    mau_t    *ptr;                /* pointer to a (next) frame */
    mau_t    frame[FRAMESIZE];   /* user memory frame */
};
```

to do the management of the memory frames. This works fine if the alignment of pointer variables is equal to the maximum alignment required for all data types. Problems will arise if some types (e.g. `long` or `double`) require a higher alignment than pointer (`void *`) or the minimum addressable unit (`mau_t` defined in rtos.h). (As far as I remember, there is a nice example how to solve the problem in [the C book](#) by Kernighan & Ritchi.)