LXRT services (soft-hard real time in user space).

* LXRT is a module that allows you to use all the services made available by RTAI and its schedulers in user space, both for soft and hard real time.

* Why is this useful?
  This way one can use a debugger like gdb while developing the code for a RT task.
There are two schedulers

* **rtai_lxrt**: this is a co-scheduler, it supports scheduling of all Linux/GNU related schedulable objects, threads and kthreads.

* **rtai_sched**: rtai_lxrt + scheduling of RTAI kernel tasks.

* For user space only applications the above schedulers are the same.

* **What is the difference?**
  With RTAI tasks one leaves the Linux environment at the benefits of cheap context switching. Linux kernel threads are commonly larger, hence context switching is more expensive. Co-operating with other Linux tasks is much simpler as everything stays inside the Linux environment. *Use rtai_lxt, unless you really need RTAI-kernel tasks!**
How do the scheduler work?

* Creating a new real time task
RT_TASK* rt_task_init(int name, int priority, int stack_size, int max_msg_size)

* rt_task_init provides a real time buddy, also called proxy, task to the Linux process that wants to access RTAI scheduler services. It needs no task function as none is used, but it does need to setup a task structure and initialize it appropriately as the provided services are carried out as if the Linux process has become an RTAI task. Because of that it requires less arguments and returns the pointer to the task that is to be used in related calls.

* Parameters:
  * name: is a unique identifier that is possibly used by easing referencing the buddy RTAI task, and thus its peer Linux process.
  * priority: is the priority of the buddy's priority.
  * stack_size: refers to the stack size used by the buddy.
  * max_msg_size: is a hint for the size of the most lengthy message most likely to be exchanged.

* If stack_size and max_msg_size are zero the default internal values are used. Usage of different values should be required only if you want to use task signal functions. In such a case note that these signal functions are intended to catch asynchronous events in kernel space and, as such, must be programmed into a companion module and interfaced to their parent Linux process through the available services.
void rt_make_hard_real_time ( void)
* Gives a Linux process, or thread, hard real time execution capabilities allowing full kernel preemption. i.e., it makes the soft Linux POSIX real time process, from which it is called, a hard real time LXRT process. --Linux will not know of it, while having it still in its process list.
* It is important to remark that this function must be used only with soft Linux POSIX processes having their memory locked in memory.
* Only the process itself can use this functions, it is not possible to impose the related transition from another process.
* Note that processes made hard real time should avoid making any Linux system call that can lead to a task switch as Linux cannot run anymore processes that are made hard real time. To interact with Linux you should couple the process that was made hard real time with a Linux buddy server, either standard or POSIX soft real time (rt_linux_syscall_server_create). To communicate and synchronize with the buddy you can use the wealth of available RTAI, and its schedulers, services, we come back to this.
* After all it is questionable to use a non hard real time Operating System, i.e. Linux, from within hard real time processes.
void rt_make_soft_real_time(void):
* turns a hard real time user space process to
  soft Linux POSIX real time;
* This needs to be followed by `rt_task_delete()`
in user space to dettach the RTAI process from
the LINUX kernel task structure. Solely do this
ti avoid side effects in Linux, RTAI will clean-up
anyway.
How does hard RT come for LINUX’s schedulable objects?
* With `rt_make_hard_real_time()` a task suspends itself.
* The Linux scheduler executed for invoking the next Linux object, i.e., process, pthread, or kthread is preempted and the RTAI tasks resumes in hard RT.
* Upon exit RTAI will execute `rt_schedule` which will give schedule a LINUX object if there is nothing more to do.
* Compatibility is assured by a common context-switching function from Linux `sched.c`
* Going back to Linux needs some attention, the hardened task needs to suspend itself and put itself on a circular buffer. The entries of which are waiting to be awakened by a Linux “wake_up_process”.
* RTAI must now remove the task properly from its list of schedulable objects
Kernel RT task

* A RT task running in the kernel mode consists of two main sections:
  * `init_module()`
    called upon insertion of a module into the running kernel, here one may allocate the resources used by the RT task and starts the RT task.
  * `cleanup_module()`
    called upon removal of the module from the running kernel. Right place to release resources
Example 1 rtai_kernel_mode_task.c:
#include <linux/module.h>
#include <asm/io.h>
#include <rtai.h>
#include <rtai_sched.h>

static RT_TASK Simple_Task;

MODULE_LICENSE("GPL");

static void Simple_Thread(int t) {
    //This is the real-time thread
}

int init_module(void) {
    printk("Init module function starting point\n");
    return 0;
}

void cleanup_module(void) {
    printk("Cleanup module function starting point\n");
}
User Space RT task

* Via the usage of some dedicated functions of the RTAI API one can turn a Linux function (with its main routine) into a RT task running in user space. --This is mainly what we have already seen before.

* But how do we return the focus to a LINUX entity to be executed while being in the Kernel mode?

  * `rt_task_wait_period` suspends the execution of the currently running RT task until the next period is reached. The task must have been previously marked for a periodic execution by calling `rt_task_make_periodic()` or `rt_task_make_periodic_relative_ns()`.

  * This way we give the focus back to the Linux co-scheduler which can resume with the execution of the process, or thread.

**RT Tasks in User space**
In kernel mode the initially created tasks are in the suspended state. The function `rt_task_make_periodic()` marks the task, previously created with `rt_task_init()`, as suitable for a periodic execution.

But before using them it's mandatory to start the timer. The timer is the main step to allow having deterministic timing constraint inside the RTAI created task. The timer can be started or stopped with the API defined in “rta_sched.h”.

There are 2 modes for timers periodic and oneshot. Need to be set before starting the timer.
* **rt_set_oneshot_mode** sets the oneshot mode for the timer. It consists in a variable timing based on the CPU clock frequency. This allows tasks to be timed arbitrarily. It must be called before using any time related function, including time conversions. Note that on i386s, i486s and earlier Pentiums, and compatibles, there is no CPU Time Stamp Clock (TSC) to be used as a continuously running time base for oneshot timings. For such machines a continuously running counter 2 of the 8254 timer is used to emulate the TSC. No wrap around danger exists because of the need of keeping Linux jiffies at HZ hz (HZ is a macros found in Linux param.h and is usually set to 100). Note however that reading an 8254 counter takes a lot of time. So on such machines the oneshot mode should be used only if strictly needed and for not too high frequencies. Moreover, for such a case, the timer resolution is clearly that of the 8254, i.e. 1193180 Hz.

* **rt_set_periodic_mode** sets the periodic mode for the timer. It consists of a fixed frequency timing of the tasks in multiple of the period set with a call to start_rt_timer. The resolution is that of the 8254 (1193180 Hz), or timed by the local APIC timer if present and enabled with a timer resolution of the local APIC timer frequency, generally the bus frequency divided 16. Any timing request not being an integer multiple of the set timer period is satisfied at the closest period tick. It is the default mode when no call is made to set the oneshot mode. Oneshot mode can be set initially also with the OneShot command line parameter of the rtai_sched module.

* If your project runs multiple tasks pay attention to **start the timer only once** and to avoid **stopping it when your task exit**.

* Otherwise you might have side effects, i.e. wrong wake ups or no wake ups at all.

* **TIMERS**
#define TICK_TIME 1000000

if ((hard_timer_running = rt_is_hard_timer_running())) {
    printf("Skip hard real_timer setting...\n");
    sampling_interval = nano2count(TICK_TIME);
}
else {
    printf("Starting real time timer...\n");
    rt_set_oneshot_mode();
    start_rt_timer(0);
}

sampling_interval = nano2count(TICK_TIME);
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <rtai_lxrt.h>
#include <sys/io.h>

#define TICK_TIME 1000000
#define CPUMAP 0x1

static RT_TASK *main_Task;
static RT_TASK *loop_Task;
int keep_on_running = 1;
static pthread_t main_thread;
static RTIME expected;
static RTIME sampling_interval;

*Example: a periodic task*
static void *main_loop() {
  if (!(loop_Task =
       rt_task_init_schmod(nam2num("RTAI01"), 2, 0, 0, SCHED_FIFO, CPUMAP))) {
    printf("CANNOT INIT PERIODIC TASK\n");
    exit(1);
  }
  expected = rt_get_time() + 100*sampling_interval;
  rt_task_make_periodic(loop_Task, expected, sampling_interval);
  rt_make_hard_real_time();

  while (keep_on_running) {
    //insert your main periodic loop here
    rt_task_wait_period();
    //set keep_on_running to 0 if you want to exit
  }
  rt_task_delete(loop_Task);
  return 0;
}

*Example: a periodic task*
int main(void) {
    RT_TASK *Main_Task;
    if (!(Main_Task = rt_task_init_schmod(nam2num("MNTSK"), 0, 0, 0, SCHED_FIFO, 0xF))) {
        printf("CANNOT INIT MAIN TASK\n");
        exit(1);
    }

    if ((hard_timer_running = rt_is_hard_timer_running())){
        printf("Skip hard real_timer setting...\n");
        sampling_interval = nano2count(TICK_TIME);
    } else {
        printf("Starting real time timer...\n");
        rt_set_oneshot_mode();
        start_rt_timer(0);
    }
    sampling_interval = nano2count(TICK_TIME);
    pthread_create(&main_thread, NULL, main_loop, NULL);
    while (keep_on_running) sampling_interval = sampling_interval; //do nothing!
    rt_task_delete(Main_Task);
    return 0;
}

*Example: a periodic task*