Real-Time Linux

Agenda

- **Real Time Linux**
- **Real Time Applications Interface (RTAI)**
  - Hardware Abstraction Layer
  - Real Time Schedulers
  - Real Time Services
  - Features and Performance
- **Programming in RTAI**
  - Application Development

Real Time Systems - Recap

- **A real-time system is a "System capable of guaranteeing logical correctness & timing requirements of the processes under its control"**
  - Fast – Low latency (quick response to external, asynchronous events and context switching)
  - Predictable – deterministic completion time of tasks with certainty

Real-Time Systems - Recap

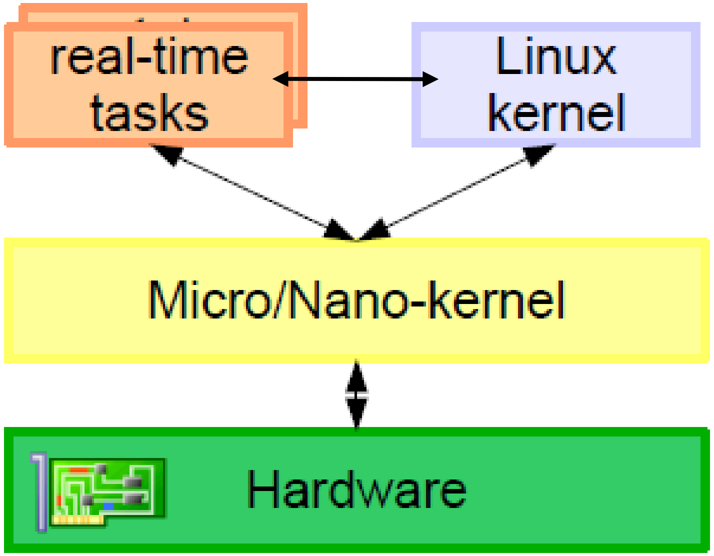
- **Real Time Operating Systems are typically used for control or communications applications in which untimely response can have catastrophic consequences**
  - Air traffic controllers
  - Missile guidance systems
  - Health monitoring systems
- **Scheduling guarantees are offered by real-time operating systems such as QNX, LynxOS or VxWorks**

Standard Linux OS

- Linux is a time-sharing OS
  - Good performance as a desktop/server OS with Sophisticated services
  - Designed to maximize throughput and give a fair share of the CPU
    - Scheduler avoids process starvation
  - User space preemptible
    - Not preemptible when executing system calls(in Kernel space)
  - Non deterministic timing behaviour of some kernel services: memory allocation, system calls
  - Changes in Interrupt latency
    - Disabling Interrupts : spinlocks, evil drivers masking out interrupts
    - Shared interrupts : all handlers are executed (don't use shared IRQ lines for critical events)

Real time Linux extensions

- **Goal of Real time extensions**
  - Add extra layer between hardware and the Linux kernel to manage real-time tasks separately
  - Changes in the kernel sources which ensures real time platform, with low latency and high predictability requirements
  - Strict separation of hard and non hard real time processes
  - Communication and synchronization services to allow an effective interaction between the two environments



Real-Time Linux – The Concept

- In a real-time Linux system
  - Real-time kernel has a higher priority than the Linux kernel
  - Real-time tasks are executed by the real-time kernel, and normal Linux programs are allowed to run when no real-time tasks have to be executed
  - Linux is treated as the idle task of the real-time scheduler
    - When this idle task runs, it executes its own scheduler and schedules the normal Linux processes.
    - A normal Linux process is preempted when a real-time task becomes ready to run

Real-Time Linux – The Concept

- Scheduler execution
  - Driven by timer interrupts of a clock to reschedule at certain times
  - An executing program can block or voluntary give up the CPU in which case the scheduler is informed by means of a software interrupt
  - Driven by hardware generated interrupts, interrupt the normal scheduled work
- RT Linux uses the flow of interrupts to give the real-time kernel a higher priority
  - When an interrupt arrives, it is first given to the real-time kernel. Interrupts are stored, for Linux to handle, when the real-time kernel is done
  - Real-time kernel can run its real-time tasks driven by these interrupts
- Each flavor of RT Linux does this is in its own way

Variants of Real-Time Linux Extensions

- **Real Time Linux (RTLinux)**
  - Developed at the New Mexico Institute of Mining and Technology
  - RTOS Micro kernel running entire Linux in fully preemptive mode
  - Runs special real-time tasks and interrupt handlers
  - FiFo, Shared memory, Semaphores. POSIX mutexes and threads
  - Avg latency - 15us
  - Used to control robots, data acquisition systems, manufacturing plants, and other time-sensitive instruments and machines

- **Xenomai Framework**
  - Platforms - x86, ARM, POWER, IA-64, Blackfin, nios
  - Implementing and migrating real time applications, based on standard APIs or emulators of proprietary RTOS interfaces, such as VxWorks and pSOS
  - Linux-hosted dual kernel, with pure Adeos patch
  - User space and kernel space RT tasks
  - Avg Interrupt response time - 43us

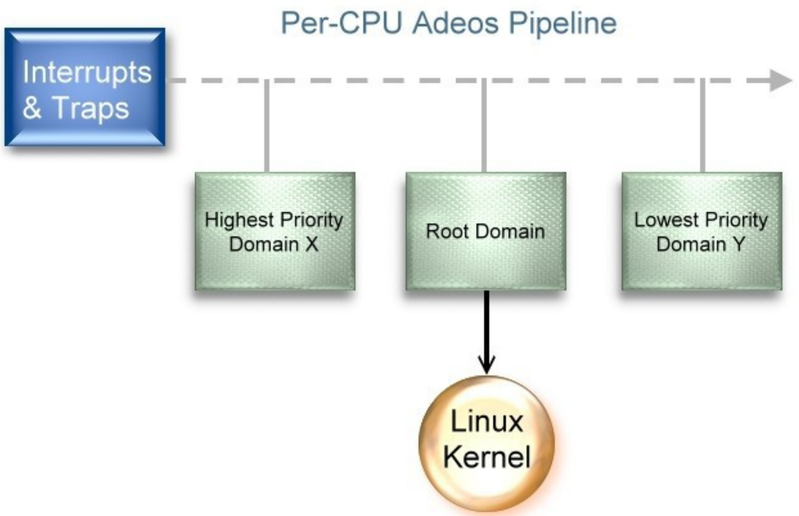
Real Time Application Interface (RTAI)

Real-Time Application Interface (RTAI)

- RTAI Core system
  - RTAI is integrated into Linux through a kernel patch and a series of add on programs (loadable modules) expanding the Linux kernel to hard real time
- Adaptive Domain Environment for operating Systems (ADEOS) patch
  - Nanokernel Hardware Abstraction Layer (HAL)
  - Provides flexible environment for sharing hardware resources among multiple operating systems
  - Implements a pipeline scheme into which every domain (OS) has an entry with a predefined priority.
  - For each event (interrupts, exceptions, syscalls, ...), the various domains may handle the event or pass it down the pipeline
  - RTAI is the highest priority domain which always processes interrupts before the Linux domain, thus serving any hard real time activity fully preempting anything that is not hard real time

Real-Time Application Interface (RTAI)

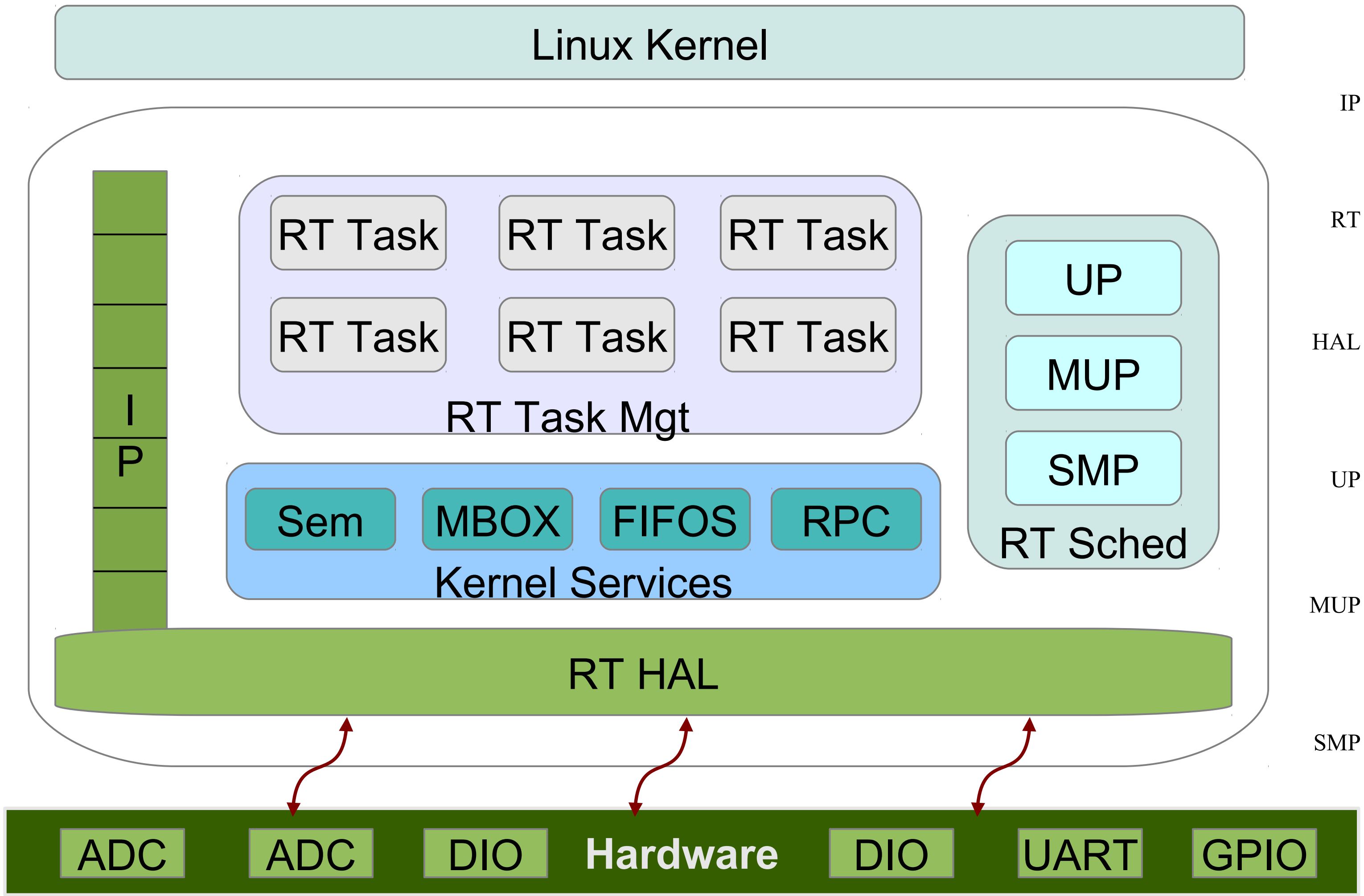
- Interrupt pipeline(I-pipe)
  - The high priority domain is at the beginning of the pipeline, so events are delivered first to it
  - There is a pipeline for each CPU
  - The Linux domain is always the root domain. Other domains are started by the root domain
  - Linux starts and loads the kernel modules that implement other domains



Real-Time Application Interface (RTAI)

- Virtualized interrupts disabling
  - Each domain may be “stalled”, meaning that it does not accept interrupts
  - Hardware interrupts are not disabled(except for the leading domain in pipeline, i.e RTAI) however, instead the interrupts received during that time are logged and replayed when the domain is unstalled
  - Thus Linux is not permitted to disable hardware interrupts, and hence, cannot add latency to the interrupt response time of the real time system

RTAI Architecture



IP	:	Interrupt Pipeline
RT	:	Real Time
HAL	:	Hardware Abstraction Layer
UP	:	Uni Processor
MUP	:	Multi Uni Processor
SMP	:	Symmetric Multi Processor

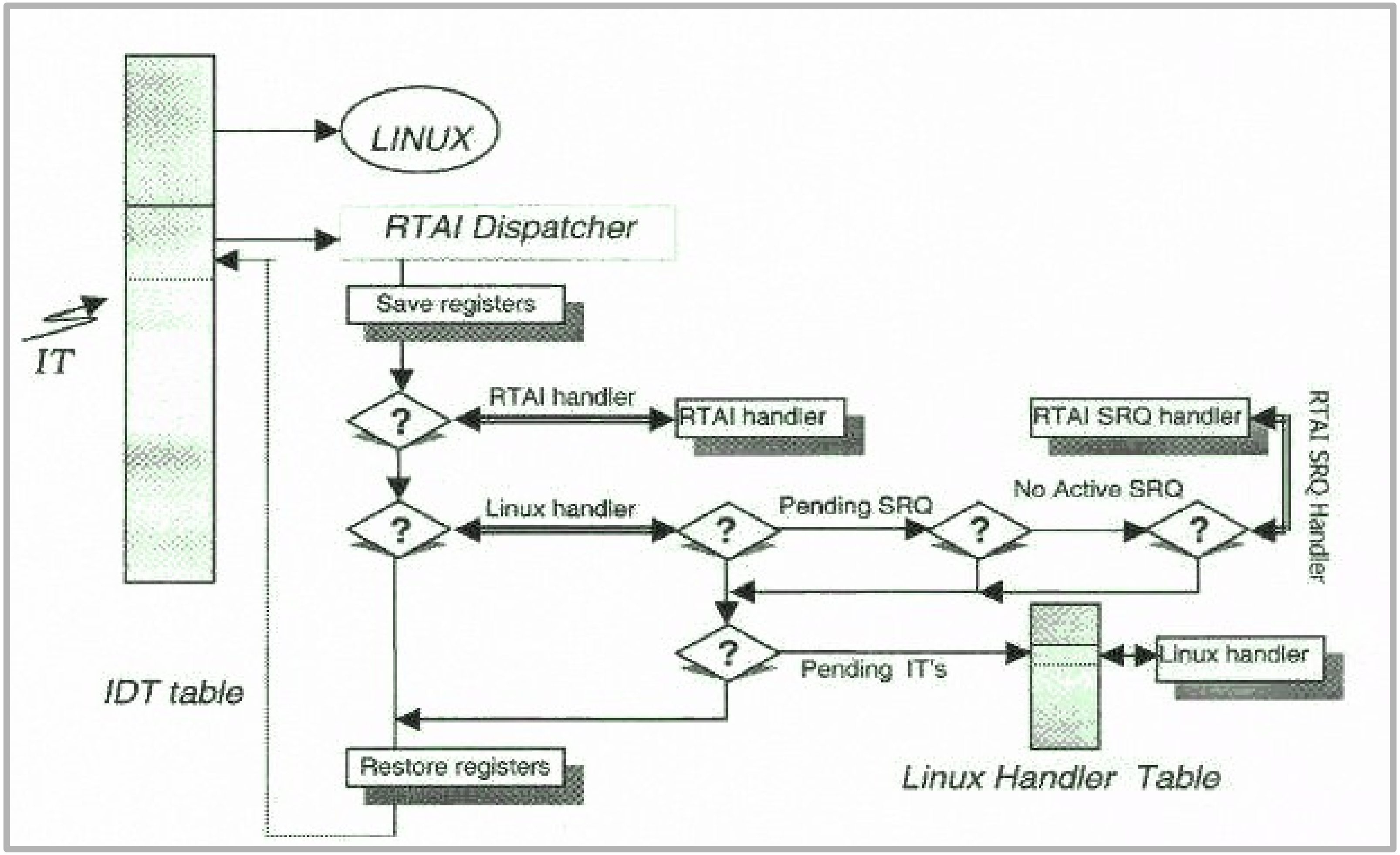
RTHAL

- RTHAL contains
  - Pointer to Interrupt Descriptor Table (IDT)
    - A key component which provides a set of pointers, which defines to which processes each of the interrupts should be routed
  - Functions to enable/disable x86 CPU interrupts (cli, sti)
  - Functions to mask/unmask interrupt controller
- Upon activation of RTHAL
  - A new IDT, which is a duplicate copy of the standard Linux interrupt descriptor table becomes the valid table
  - Handler functions in the new IDT table are changed to the RTAI interrupt dispatcher function so that it takes control of the system hardware and its interrupts

RTAI Interrupt Dispatcher

- The Interrupt Dispatcher called by the IDT when interrupts occur
  - Activates the right handler depending on the owner of the driver, either RTAI / Linux / both (RTAI first then Linux)
  - The addresses of the Linux handlers are saved before changing the IDT in the HAL
  - Takes into account the “SRQ(service request) handlers”
    - SRQ is a handler used for a system call implementation. SRQ can be either RTAI handler (e.g.: fifo) or User handler
    - RTAI manages up to 31 SRQ's. SRQ handler addresses are stored in IDT

RTAI Interrupt Dispatcher



RTAI Scheduler

- The RTAI multitasking scheduler, uses Interrupt-driven, Priority-based task scheduling
- Scheduler elects the highest priority task in READY state
  - Priority 0 is the highest priority
  - 0x3fffFfff is the lowest for rtai tasks
  - Linux is given priority 0x7fffFfff
- Pure Periodic Mode (Default)
  - Program the timer only once with a frequency
  - Task activations are only possible at multiples of the timer period
- One-shot Mode
  - Program the timer once for every iteration
  - Arbitrary timings for task activations

RTAI Scheduler Timing

- RTAI schedulers, timed by three type of hard timers
  - 8254 Timer IC
    - Allows one shot and periodic modes
    - Special care is taken in maintaining Linux timing
    - One-Shot mode is provided by Time Stamp Clock (TSC) or Counter2
  - APIC
    - Allows one shot and periodic modes
    - One-Shot mode is provided Time Stamp Clock (TSC)

RTAI Schedulers

- **The RTAI distribution includes three different priority based, pre-emptive real time schedulers**
  - Uni-Processor (UP) scheduler
  - Multi Uni-Processor (MUP) scheduler
  - Symmetric Multi-Processor (SMP) scheduler
- **During the installation process, a scheduler is determined based on the hardware configuration of the target machine**
  - It is then copied and linked so that it is called by the generic rtai_sched.ko reference

RTAI Schedulers

- **UP scheduler - For uni-processor platforms**
  - Based on the 8254 based timer
  - Supports either one-shot or periodic scheduling but not both simultaneously
- **SMP scheduler - For multi-processor machines**
  - Based on either 8254 timer or APIC timers
  - Supports either one-shot or periodic scheduling but not both simultaneously
  - Tasks can run symmetrically on any or a cluster of CPUs, or be bound to a single CPU
  - By default all tasks are defined to run on any of the CPUs and are automatically moved between CPUs as the system's processing and load requirements change.

RTAI Schedulers

- **Multi-Uniprocessor scheduler - For multiprocessor platforms only**
  - Supports both one-shot and periodic scheduling simultaneously.
  - The main advantage is the ability to be able to use mixed timers simultaneously, i.e. periodic and one-shot timers.
  - Like the SMP schedulers, the MUP can use inter-CPU services related to semaphores, messages and mailboxes.

RTAI Schedulers

- **All RTAI schedulers incorporate standard RTOS scheduling services like**
  - Resume
  - Yield
  - Suspend
  - Make Periodic
  - Wait Until

RTAI Task States

- The scheduler looks for ready tasks or tasks with expired period.
- A task may be in the following states
    - READY
    - SUSPENDED
    - DELAYED
    - SEMAPHORE
    - SEND
    - RECEIVE
    - RPC
    - RETURN
    - RUNNING

RTAI Features

- Features
  - Supports UniProcessor, Multi-UniProcessor and Symmetric Multi- Processor (SMP)
  - One-shot and Periodic schedulers
  - IPC : Semaphores, mailboxes, FIFOs, shared memory, and RPCs
  - POSIX 1003.1c (Pthreads, mutexes and condition variables) & POSIX 1003.1b (Pqueues only) compatibility
  - LXRT – which allows the use of the RTAI system calls from within standard user space
  - /proc interface – which provides information on the real-time tasks, modules, services and processes extending the standard Linux /proc file system support

Real-Time Application Interface (RTAI)

- The Real Time kernel, all its component parts, and the real time applications are all run in Linux kernel address space as kernel modules
  - These modules can be removed from the kernel on completion of the real time system operation. Advantage of this is, it aids system modularity
  - If the scheduler is unsuitable for a particular application, then the scheduler module can be replaced by one that meets the needs of the application
  - One of the main disadvantages of running in Linux kernel address space is that a bug in a real time task can crash the whole system

Real-Time Application Interface (RTAI)

- Provides
  - Deterministic and pre-emptive performance
  - Allows the use of all standard Linux drivers, applications and functions
  - Highly Modularised
- Uses Linux
  - System and device initialisation
  - Blocking dynamic resource allocation
  - Loadable module mech: to install components of the Real-Time kernel
- Decoupled from the Linux kernel
  - Real-time kernel can be kept small and simple
  - Kernels can be optimized independently

RTAI Performance

- RTAI's performance is very competitive with the commercial Real Time Operating Systems such as VxWorks, QNX etc
- RTAI Offers
  - Context switch times of 4 uSec
  - Interrupt Response of 20 uSec
  - Periodic Tasks with periodicity 100 KHz

RTAI Programming

Topics

- Starting the Timer and Real-Time Scheduler
- Task Initialization
- Scheduling One-shot and Periodic Tasks
- Task Function
- Task APIs

Starting the Timer and Realtime Scheduler

- The timer can be started in either
  - Pure Periodic Mode (Default)
    - Program the timer only once with a frequency
    - Task activations are only possible at multiples of the timer period
    - **rt_set_periodic_mode()**
  - One-shot Mode
    - Program the timer once for every iteration
    - Arbitrary timings for task activations
    - **rt_set_oneshot_mode()**

Note: Mode must be set before using any time related function

Starting the Timer and Realtime Scheduler

- Start the real-time timer and scheduler using
  - **RTIME start_rt_timer(RTIME period)**
  - period is required only for the periodic mode
    - Given in internal count units (clock ticks)
    - RTIME nano2counts(int nanoseconds)
  - Scheduler is activated in the timer interrupt handler
- Stop the timer and scheduler
  - **void stop_rt_timer(void)**

Task Initialization

- RT_TASK data structure holds the information about a task
  - Task function, any initial argument passed to it
  - Size of the stack allocated for its variables
  - Task priority
  - Whether or not task uses floating-point math
  - "signal handler" that will be called when the task becomes active

Task Initialization

- Initializing the Task Structure(RT_TASK)
  - **rt_task_init (RT_TASK *task, void *rt_thread, int data, int stack_size, int priority, int uses_fp, void *sig_handler)**
  - The newly created real time task is initially in a suspended state
- To start the task immediately or change a task state from suspended to ready
  - **int rt_task_resume(RT_TASK *task)**

Scheduling Tasks

- Start after some delay, as a periodic task
  - **int rt_task_make_periodic(RT_TASK *task, RTIME start_time, RTIME period);**
    - start_time - absolute time to begin execution in clock ticks. Typically "now"(i.e., calling rt_get_time())
    - Period - task's period in clock ticks
  - **int rt_task_make_periodic_relative_ns (RT_TASK *task, RTIME start_delay, RTIME period);**
    - start_delay - Delay after which task starts execution, relative to the current time in nanoseconds
    - period - dummy value for a oneshot task, task's period for a periodic task

Task Function

- Periodic task function
  - Should enter an infinite loop, in which it does its job
  - On completion of task's activity wait for its next scheduled cycle
    - **void rt_task_wait_period(void);**
- Periodic task function code snippet

```
void task_function(int arg) {
    while (1) {
        /* Do your thing here */

        rt_task_wait_period();

    }

    return;

}
```

Task APIs

- Delete a task
  - **int rt_task_delete (RT_TASK *task)**
  - One-shot tasks delete themselves
- Yield current task
  - **void rt_task_yield (void)**
  - Stops and puts the task at the end of the ready list
- Suspend a task
  - **int rt_task_suspend (RT_TASK *task)**
  - Task will not execute until it is resumed again

THANK U