Unit 5 – Real-Time Programming

Real-Time Systems

- References:
  - ✓ "Real-Time Embedded Systems", Xiacong Fan, 2015.
  - ✓ "Real-Time Systems", Jane W. Liu, 2000.

- A real-time system has to respond to externally generated input stimuli within a finite and specified period.
  - ✓ Correctness depends not only on the functional result, but also on the time it was delivered.
  - ✓ Failure to respond is as bad as the wrong response.
- Each incoming service request imposes a task typically associated with a real-time computing constraint (timing constraint).
- Real-time is not necessarily fast. Fast means low average latency. Real-time needs predictable worst-case performance.
- Timing constraint: specified in terms of its deadline (time instant by which its execution is required to be completed). Types:
  - ✓ Hard timing constraint: If the consequence of a missed deadline is fatal. A late response is useless and/or unacceptable.
  - ✓ Soft timing constraint: If the consequence of a missed deadline is undesirable but tolerable. A late response is still useful as long as it occurs occasionally (with low probability)
- As for real-time applications, they can be classified according to their timing attributes into four types:

TABLE I. CLASSIFICATION OF REAL-TIME APPLICATIONS

Purely cyclic	▪ Every task executes periodically. Even I/O operations are polled. ▪ Demands in resources (e.g., computing, communication, storage) do not vary significantly from period to period ▪ Example: most digital controllers and real-time monitors.
Mostly cyclic	▪ Most tasks execute periodically ▪ The system must also respond to some external events asynchronously (e.g., fault recovery and external commands). ▪ Example: modern avionics and process control systems
Asynchronous and somewhat predictable	▪ Most tasks are not periodic. ▪ The time between consecutive executions of a task may vary considerably, or the variations in resource utilization in different periods may be large. These variations have either bounded ranges or known statistics ▪ Examples: multimedia communication, radar signal processing and tracking.
Asynchronous and unpredictable	▪ These are applications that react to external asynchronous events and have tasks with high and variable run-time complexity. ▪ Example: intelligent real-time control systems.

Soft Real-Time Systems

- All tasks have soft timing constraints.
- It offers best-effort services, i.e., its service of a request is almost always completed within a known finite time. Occasionally, it may miss a deadline (which is usually tolerable).
- Soft timing constraints: expressed in probabilistic terms (e.g.: average performance, standard deviation).
- Example: Wireless router that provides a number of late/lost frames as 2/min. The consequence of a missed deadline is that the user has a bad web surfing experience.

Hard Real-Time Systems

- Key tasks have hard timing constraints. Missing some deadlines is completely unacceptable.
- It offers guaranteed services. It must have both functional correctness and timing correctness. In most cases timing correctness is more important than functional correctness.
- Hard timing constraints: expressed in deterministic terms.
- Example: cardiac pacemaker, antimissile systems, FTSE 100 index (calculated in real time and published every 15 s; the consequence of a missing deadline if financial catastrophe).

- There is no precise boundary between soft and hard time real-time systems: it depends on the application and context.
- Firm Real-Time Systems: Systems which are soft-real time but in which there is no benefit from late delivery of services. Another way to put it: systems that lie between hard and soft real-time systems.
- Real Real-Time Systems: These are hard systems in which the response times are very short (e.g.: missile guidance system).

Real-Time Embedded Systems

- A real-time embedded system is one that is designed to be embedded within some larger system.
- Timing correctness is equally important as functional correctness.

Reference Model of Real-Time Systems

- A reference model and consistent terminology let us reason about real-time systems. A model allows us to focus on the timing properties and resource requirements of system components and the way the operating system allocates the available system resources among them. By describing the algorithms used to schedule the applications and the methods for validating their timing constraints abstractly, rather than in implementation-specific terms, we can appreciate their general applicability better. The reference model is characterized by:
  - ✓ Workload model: It describes the applications supported by the system.
  - ✓ Resource model: It describes the system resources available to the applications.
  - ✓ Scheduling algorithms that define how the applications execute and use the resources at all times.
- Definitions:
  - ✓ Job: Unit of work scheduled and executed by the system.
    - Task: Set of related jobs that jointly provide some function.
    - If jobs occur on a regular cycle, the task is termed periodic.
    - If jobs are unpredictable, the task is termed aperiodic.
  - ✓ Processors: Active devices on which jobs are scheduled.
  - ✓ Resource: passive entity on which jobs might depend (e.g.: system memory, hardware device).
    - Resources are not consumed by usage and can be reused multiple times.
    - Jobs compete for access to resources.

Characterization of Application Systems

- Jobs: basic components of any real-time application system. The operating system treats each job as a unit of work and allocates processor and resources to it. For the purpose of scheduling and validation, it suffices to define each job by its temporal, resource, interconnection and functional parameters.
- Among the parameters of a job J_i, the most common are:
  - ✓ Release time (or arrival time) r_i : Instant of time at which the job becomes eligible for execution. The release time of the job may be jittery (sporadic), meaning that $r_i \in [r_i^-, r_i^+]$ and that only the range of r_i is known but not its actual value.
  - ✓ Relative deadline D_i : Maximum allowable response time of the job.
  - ✓ Absolute deadline d_i: Instant of time by which the job must complete. $d_i = D_i + r_i$
  - ✓ Laxity (slack) type: The deadline (or timing constraint in general) of the job can be hard or soft.
  - ✓ Execution time e_i: Amount of time required to complete the execution of a job J_i when it executes alone and has all the resources it requires. The execution time of J_i may vary, meaning that $e_i \in [e_i^-, e_i^+]$ and that this range is known but not the actual value of e_i.
  - ✓ Preemptivity: The job may be constrained to be non-preemptable.
  - ✓ Resource requirements: This parameter gives the processors and resources required by the job in order to execute and the time interval(s) during which each resource is required.

- r_i, d_i, and D_i, ei, are the temporal parameters of a job J_i. The time interval $(r_i, d_i]$ between the release time and absolute deadline of the job J_i is called its feasible interval. Fig. 1 depicts these parameters.

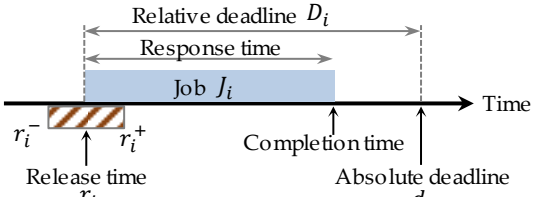

Figure 1. Temporal parameters of a job. J_i

- **Periodic Tasks**: A set of jobs that are executed at regular time intervals can be modeled as a periodic task. Many real-world systems fit this model.
- **Aperiodic and Sporadic Tasks**: Many real-time systems are required to respond to unpredictable events. When such events occur, the system executes a set of jobs in response. The release time of these jobs are not known until the event triggering them occurs. These jobs are modelled as sporadic or aperiodic jobs, because they are released at random times.
  - ✓ An aperiodic job has no deadline; a sporadic job has a deadline once released.
  - ✓ Each aperiodic or sporadic task is a stream of aperiodic or sporadic jobs, respectively.
  - ✓ Sporadic Task: it contains jobs that are released at random time instants and has hard deadlines.
    - Sporadic tasks make the design of a hard real-time system impossible, unless some bounds can be placed on release times and relative deadlines.
  - ✓ Aperiodic Task: If the jobs in it have either soft deadlines or no deadlines.

Real-Time Scheduling

- Jobs are scheduled and allocated access to resources according to a set of scheduling algorithms and some resource access-control protocol. The module that implements these algorithms is called the **scheduler**.
- A valid schedule satisfies the following conditions:
  - ✓ Every processor is assigned at most one job at any time; every job is assigned to at most one processor at once
  - ✓ No job is scheduled before its release time.

- ✓ The total amount of processor time assigned to each job is equal to its maximum or actual execution time
- ✓ All the precedence and resource usage constraints are satisfied

- A schedule is feasible if it's valid and every job meets its timing constraints
- A scheduling algorithm is optimal if it always produces a feasible schedule for a given set of jobs if a feasible schedule exists
  - ✓ There are some scheduling algorithms that will find some, but not all, feasible schedules, and so may fail to schedule a set of jobs that some other algorithm could schedule

DYNAMIC VS. STATIC SYSTEMS

- Dynamic multiprocessor system: Jobs are dynamically dispatched to processor, i.e., a job can migrate between processors.
- Static multiprocessor system: If jobs are assigned and bound to a single processor. Jobs are moved among processors only when the system must be reconfigured (i.e., when the operation mode of the system changes or some processor fails).
  - ✓ Dynamic systems may be more responsive on average, however their worst-case real-time performance may be poorer than static systems.
  - ✓ Static systems have inferior performance (in terms of overall response time) compared to dynamic system.
    - However, it is possible to validate static systems, whereas this is not always true for dynamic systems; hence, most hard real-time systems are static.
    - Results demonstrated for uniprocessor systems are applicable to each processor of a static multiprocessor system; they are not necessarily applicable to dynamic multiprocessor systems.

APPROACHES TO REAL-TIME SCHEDULING

- Two main classes of algorithms for scheduling real-time tasks:
  - ✓ Clock-driven (or time-driven) algorithms: used for mostly static systems, where all properties of all jobs are known at design time, such that offline scheduling techniques can be used.
  - ✓ Priority-driven algorithms: used for more dynamic real-time systems, with a mix of periodic tasks and event-driven (aperiodic and/or sporadic tasks), where the system must adapt to changing events and conditions.

CLOCK-DRIVEN SCHEDULING

- Decisions about what jobs to execute are made at specific time instants.
  - ✓ These instants are chosen before the system begins execution.
- Typically, in clock-driven systems:
  - ✓ All parameters of the hard real-time jobs are fixed and known.
  - ✓ A schedule of the jobs is computed off-line and it is stored for use at run-time; thus, scheduling overhead at run-time can be minimized.
  - ✓ Simple and straightforward, not flexible.
- Frequently adopted choice: make scheduling decisions at regularly spaced time instant.
  - ✓ Implementation: hardware timer that issues a ***periodic timer interrupt***. Scheduler awakes after each interrupt, schedules the job to execute for the next period, then blocks itself until the next interrupt.

PRIORITY-DRIVEN SCHEDULING

- It refers to a large class of scheduling algorithms that never leave any resource idle intentionally (as this is not optimal).
  - ✓ Other common names: greedy scheduling, list scheduling, work-conserving scheduling.
- Assign priorities to jobs, based on some algorithm.
- Make scheduling decisions based on the priorities, when events such as releases and job completions occur.
  - ✓ Priority scheduling algorithms are event-driven.
  - ✓ Jobs are placed in one or more queues; at each event, the ready job with the highest priority is executed.
  - ✓ The assignment of jobs to priority queues, along with rules such as whether preemption is allowed, completely define a priority scheduling algorithm.
- Most scheduling algorithms used in non real-time systems are priority-driven:
  - ✓ First-In-First-Out (FIFO), Last-In-First-Out (LIFO): Assign priority based on release time.
  - ✓ Shortest-Execution-Time-First (SETF), Longest-Execution-Time-First (LETF): Assign priority based on execution time.
- Real-time priority scheduling assigns priorities based on deadline of some other timing constraint:
  - ✓ Earliest deadline first (EDF): Earlier the deadline, the higher the priority. Simple, it just requires knowledge of deadlines.
  - ✓ Least slack time first (LST): The smaller the slack time, the higher the priority. $t_{slack} = d_i - t - t_{rem}$, $t_{rem} = e_i - (t - r_i)$. It is more complex as it requires knowledge of execution times and deadlines.

BASIC MECHANISMS

- In an ideal world, the occurrence of events and the actions that need to be taken can be planned for and scheduled into a clean multi-tasking multi-use OS.
- However, in the real-world, external events can, and will occur whenever they want to, and the Real Time System must react to them within a specified time. For example, we may have an electronic control unit (ECU) for a car, that measures road speed and updates the driver's speedometer. To achieve this, the time of arrival of pulses from the road speed sensor must be logged almost immediately (depending upon the accuracy required) whatever the processor may be doing. So, the processor must suspend its current operation, and pass control immediately over to the road speed sensor function. The processor is ***interrupted*** by the external event, which takes over control of the processor and all its resources.
- When designing a Real Time System, consideration must be given to handling the system interrupts, and how data is passed from the interrupt functions to the main body of the programmed for processing. Therefore, it is necessary to analyze both the timing implications of the interrupts, and the data flow requirements. In addition, the implementation must take account of the disruption an unscheduled interrupt can cause to the smooth operation of a structured programmed.
- Timing interrupts are generated internally by the processor itself. An independent digital divider circuit uses the processor clock as timing source. This divider is can be set up by the programmer to divide the main clock source down to give a slower time base.
- Two basic examples are shown here:
  - ✓ Software Interrupt: using keyboard and alarm.
  - ✓ Real-time clock (Timer)

SOFTWARE INTERRUPT VIA SIGNAL

- A signal is a software interrupt delivered to a process. The OS uses signals to report exceptional situations to an executing program, e.g.: references to invalid memory addresses, report asynchronous events.
- If you anticipate an event that causes signals, you can define a ***handler function*** and tell the OS to run it when that particular type of signal arrives.

SIGNAL HANDLING

- Signal: Event generated to notify a process or thread that some important situation has arrived. When a process or thread receives a signal, it will stop what it is doing and take some action. Signals might be useful for inter-process communication.
  - ✓ **Standard signals**: Defined in the `signal.h` header file. Each signal name is a macro (positive integer). Every signal has a unique numeric value, but you should always use the name of the signals, not the number (they can differ based on the system but meaning of the name is standard). For a comprehensive documentation on signals, go here.
    - `NSIG`: Total number of signals defined.
    - Program Error Signals: `SIGFPE, SIGILL, SIGSEGV, SIGBUS, SIGABRT, SIGIOT, SIGTRAP, SIGEMT, SIGSYS.`
    - Termination Signals: `SIGTERM, SIGINT, SIGQUIT, SIGKILL, SIGHUP`
    - Alarm Signals: `SIGALRM, SIGVTALRM, SIGPROF`
    - Asynchronous I/O Signals: `SIGIO, SIGURG, SIGPOLL`
    - Job Control Signals: `SIGCHLD, SIGCLD, SIGCONT, SIGSTOP, SIGTSTP, SIGTTIN, SIGTTOU`
    - Operation Error Signals: `SIGPIPE, SIGLOST, SIGXCPU, SIGXFSZ`
    - Miscellaneous Signals: `SIGUSR1, SIGUSR2, SIGWINCH, SIGINFO`

- If a process receives a signal, the process can choose to: ignore signal, specify a handler function, or accept default action.
  - ✓ If the signal is neither handled nor ignored, its default action takes place. Each signal has a default action, which may be changed using a handler function (`SIGKILL` and `SIGABRT` signal's default action cannot be changed or ignored).
    - Term: Process will terminate.
    - Core: Process will terminate and produce a core dump file.
    - Ign: The process will ignore the signal.
    - Stop: Process will stop.
    - Cont: Process will continue from being stopped.
  - ✓ Ignoring the specified action for the signal: The signal is then discarded immediately.
  - ✓ The process specifies a ***handler function*** (via `signal` function). We say that we catch and handle the signal.
    - `int signal () (int signum, void (*func)(int))`
      - Operation: if the process receives a signal (defined by `signum`), this function will call the `func` ***handler function***. It returns a pointer to function `func` if successful (or -1 otherwise).

BASIC SIGNAL HANDLER

- We specify the use of the signal `SIGINT` when invoking the function `signal()`.
  - ✓ The `SIGINT` ("program interrupt") signal is sent when the user types the INTR character (normally ***Ctrl-c***).
- The program below is in an infinite loop. When interrupted by the user typing ***Ctrl-c***, it issues a `SIGINT` signal and executes the handler function (`sig_handler`).
  - ✓ `signal (SIGINT, sig_handler)`: It registers the signal handler, i.e., when a process receives a `SIGINT` signal (***Ctrl-c***), it will execute the ***handler function*** `sig_handler`.

- If the user wants to exit the program, it can do so via the `SIGQUIT` signal that is sent when the user types the QUIT character (usually ***Ctrl-***). This is like a program error condition "detected" by the user.

```
#include<stdio.h>
#include<signal.h>
#include<unistd.h>
void sig_handler(int signum){ //Return type of the handler function should be void
  printf("\nInside handler function\n");
}

int main(){
  int i;
  signal(SIGINT,sig_handler); // Register signal handler
  for(i=1;;i++){    //Infinite loop
    printf("%d : Inside main function\n",i);
    sleep(1);  // Delay for 1 second
  }
  return 0;
}
```

  - ✓ Program Output: When the user enters ***Ctrl-***, we quit the process.

```
1 : Inside main function
2 : Inside main function
^C
Inside handler function
3 : Inside main function
4 : Inside main function
^\Quit (core dumped)
```

SIGNAL HANDLER WITH ALARM

- We can use the `alarm()` function to generate a `SIGALRM` signal after a specified amount of time has elapsed.
  - ✓ `SIGALRM` signal: Typically indicates expiration of a timer that measures real or clock time. It is used by the `alarm()` function for example.
- Syntax: `unsigned int alarm (unsigned int seconds) // defined in unistd.h header file`
  - ✓ After `seconds` seconds has elapsed since requested the `alarm()` function, the `SIGALRM` signal is generated.
  - ✓ Return value: number of seconds remaining for a previous scheduled alarm due to be delivered.

First Example

- In the program below, when the process receives the `SIGALRM` signal, it executes the ***handler function*** (`sig_handler`).
  - ✓ `signal (SIGALRM, sig_handler)`: Set the ***handler function*** to `sig_handler` when the `SIGALRM` signal arrives.
- We then schedule the alarm for 2 seconds via the `alarm()` function. Then, during the for loop execution, the `SIGALRM` signal is issued after 2 seconds and `sig_handler` function is called. After the execution of `sig_handler` function, the for loop execution is resumed (`sleep(1)` is used for delaying so we can see the flow of execution).
  - ✓ We can stop the execution using a `SIGINT` signal (***Ctrl-c***) or a `SIGQUIT` signal (***Ctrl-***)

```
#include<stdio.h>
#include<unistd.h>
#include<signal.h>

void sig_handler(int signum){
  printf("Inside handler function\n");
}

int main(){
  int i;
  signal(SIGALRM,sig_handler); // Register signal handler
  alarm(2);  // Scheduled alarm after 2 seconds
  for(i=1;;i++){ // infinite loop
    printf("%d : Inside main function\n",i);
    sleep(1);  // Delay for 1 second
  }
  return 0;
}
```

  - ✓ Program Output: When the user types ***Ctrl-c***, we quit the process.

```
1 : Inside main function
2 : Inside main function
Inside handler function
3 : Inside main function
4 : Inside main function
5 : Inside main function
6 : Inside main function
7 : Inside main function
^C
```

Second Example

- Only one `SIGALRM` generation can be scheduled with `alarm()` at a time. Successive `signal()` calls reset the alarm clock of the calling process.
- In the code snippet below, we can see that the program executed for more than 7 seconds but the first alarm which was scheduled after 4 seconds is not calling the ***handler function***. Instead, the second alarm which was scheduled after 1 second did reset the alarm, and issues `SIGALRM` signal after 1 second.

```
#include<stdio.h>
#include<unistd.h>
#include<signal.h>

void sig_handler(int signum){
  printf("Inside handler function\n");
}

int main(){
  signal(SIGALRM,sig_handler); // Register signal handler
  alarm(4);  // Scheduled alarm after 4 seconds
  alarm(1);  // Scheduled alarm after 1 seconds
  for(int i=1;;i++){
    printf("%d : Inside main function\n",i);
    sleep(1); }  // Delay for 1 second
  return 0;
}
```

  - ✓ Program Output: When the user types ***Ctrl-c***, we quit the process.

```
1 : Inside main function
Inside handler function
2 : Inside main function
3 : Inside main function
4 : Inside main function
5 : Inside main function
6 : Inside main function
7 : Inside main function
^C
```

REAL TIME CLOCK

- Most computers have two types of clocks that record the current wall clock time (elapsed real time)
  - ✓ Hardware Clock: Real Time Clock (RTC). It is battery backed and it records time even if the system is shut down.
  - ✓ System clock (or kernel clock): Maintained by the OS (you can use `gettimeofday()`, `time()`).

REAL TIME CLOCK (RTC) CONFIGURATION

- Read/written with `hwclock()` or with `ioctl()` requests. See here for more information on RTC configuration.
- Besides tracking date and time, RTCs can also generate (maskable) interrupts:
  - ✓ On every clock update (i.e., once per second). These are called ***Update Interrupts***.
  - ✓ At periodic intervals (frequencies from 2 to 8192 Hz). These are called ***Periodic Interrupts***.
  - ✓ Upon reaching a previously specified alarm time. These are called ***Alarm Interrupts***.
- Each of the interrupt sources can be enabled/disabled separately. We can monitor (and wait for) the interrupts using `read()` or `select()`. Note that a `read` on RTC reads interrupt-related data, not configuration stuff (which `ioctl` does).

- Device: `/dev/rtc` (or `/dev/rtc0`, `/dev/rtc1`, …). It can only be opened once (until it is closed) and it is read-only.
  - ✓ We require to be root user when attempting to open the RTC.
  - ✓ To open a file (or device), use the `open()` system call: `int open(pathname, flags)`
    - `pathname`: This specified the file (or device).
    - `flags`: this argument must include one of the following access modes: `O_RDONLY`, `O_WRONLY`, or `O_RDWR`. These request opening the file read-only, write-only, or read/write, respectively.
    - Return value: file descriptor (a nonnegative integer) used in subsequent system calls: `read`, `write`, `select`, `iotctl`, etc. It returns a -1 if an error occurred.
  - ✓ Example: `fd = open("/dev/rtc", O_RDONLY)`: It opens the file specified by "`/dev/rtc`". Access mode: `O_RDONLY`.

- A user process can monitor the interrupts via a `read()` or `select()` on "`/dev/rtc`". Both will block a user process until the next interrupt from RTC is received.
  - ✓ `size_t read (int fd, void *buf, size_t count)`: attempts to read up to `count` bytes from file descriptor `fd` into the buffer pointed by `buf`. When using `read()` on an RTC, we only read interrupt-related data, not the actual RTC data.
  - ✓ `int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout)`: It monitors multiple file descriptors, waiting until one or more of the file descriptors become 'ready' for an I/O operation. A file descriptor is 'ready' if it is possible to perform a corresponding I/O operation. Thus, invoking `read()` right after `select()` with a `readfds` file descriptor is non-blocking.

- File descriptor sets: They allow the caller to wait for 3 classes of events. Specify one or more as NULL if no file descriptors are to be watched for the corresponding class of events:
  - `readfds`: File descriptors in this set are watched to see if they are ready for reading.
  - `writefds`: File descriptors in this set are watched to see if they are ready for writing.
  - `exceptfds`: File descriptors in this set are watched for 'exceptional conditions'.
  - Upon return, each file descriptor set is cleared except for i) those that are ready for reading (`readfds`), ii) those thar are ready for writing (`writefds`), iii) those for which an exceptional condition has occurred. When using `select()` within a loop, the sets must be reinitialized before each call.
  - The contents of a file descriptor set can be manipulated using macros (`FD_ZERO, FD_SET, FD_CLR, FD_ISSET`). `FD_ZERO` removes all file descriptors from set. `FD_SET` adds a file descriptor to a specified file descriptor set.
- `nfds`: It is the highest-numbered file descriptor in any of the three sets + 1.
- `timeout`: It specifies the interval that `select()` should block waiting for a file descriptor to become ready. It is a ***timeval*** structure:

```
struct timeval { time_t tv_sec; // seconds
                 suseconds_t tv_usec; } // microseconds
```

✓ Sample operation:
- Initialize a file descriptor set (remove all file descriptors from it) via `FD_ZERO(fd, *fds)`.
- Use `select()` to wait until one of the file descriptors (from one or more file descriptor sets) is 'ready'.
- Use `read()` to read data from one or more file descriptors. This will be a non-blocking read. If `select()` had not been used immediately before, this would be a blocking read.
  - After the interrupt, the process can read a long integer, of which the LSByte contains a bit mask encoding the types of interrupt that occurred, while the remaining bytes contain the number of interrupts since the last read.

- To write/read on RTCs, we use the `ioctl()`. The `ioctl` (control device) system call manipulates the underlying device parameters of special files (including RTC).

✓ Syntax (for RTC requests): `int ioctl (fd, RTC_request, param)`.
- `fd`: open file descriptor. The device (a Linux file) needs to be open.
- `RTC_request`: Device-depending request code. On RTC devices there are different types of request. For example (see complete list here):
  - `RTC_RD_TIME, RTC_SET_TIME`: Read time/set time.
  - `RTC_UIE_ON, RTC_UIE_OFF`: enable/disable interrupt on every clock update.
  - `RTC_ALM_READ, RTC_ALM_SET`: Read and set the alarm time (for RTCs that support alarms). The alarm interrupt must separately enabled/disabled using the `RTC_AIE_ON, RTC_AIE_OFF` requests.
  - `RTC_IRQP_READ, RTC_IRQP_SET`: Read and set the frequency for periodic interrupts (for RTCs that support them). This interrupt must be separately enabled/disable using the `RTC_PIE_ON, RTC_PIE_OFF` requests. The set of allowable frequencies is in the range 2 to 8192.
- `param`: parameter associated with the request (specified as a pointer).
- Return value: Usually, 0 is returned on success (except for a few requests that use it as a non-negative output parameter). On error, -1 is returned.

Basic Example

- The following code snippet provides a simple way to open the RTC (when logged in as root) and to set the RTC via `ioctl()`.

```
#include <linux/rtc.h>
#include <sys/ioctl.h>

int fd;
struct rtc_time rt;
/* set your values here */
fd = open("/dev/rtc", O_RDONLY);
ioctl(fd, RTC_SET_TIME, &rt);
close(fd);
```

✓ The RTC's time is organized in a structure of type `rtc_time`.

```
struct rtc_time {
    int tm_sec;
    int tm_min;
    int tm_hour;
    int tm_mday;
    int tm_mon;
    int tm_year;
    int tm_wday; /* unused */
    int tm_yday; /* unused */
    int tm_isdst;/* unused */
};
```

✓ `ioctl(fd, RTC_SET_TIME, &rt)`: Request: `RTC_SET_TIME`
- This instruction sets the RTC's time to the time specified by the `rtc_time` structure pointed to by the third `ioctl` argument (`&rt`). You need to initialize the elements of the structure `rt` before you execute this instruction.

RTC Configuration

- Go to Tutorial #8 for a more complete example on how to deal with:
  - ✓ Read data from RTC device
  - ✓ Set and wait for ***Update Interrupts, Alarm Interrupts***, and ***Periodic Interrupts.***

INTERRUPTS (refer to here for more information)

- In the previous examples (including the complete one in Tutorial #8), we explain how to configure, enable, and monitor an interrupt. We use blocking `read()` or `select()` to wait for the Interrupt and then perform and action. However, we do not handle the interrupt (as we did with signals).

- To handle interrupts (like the ones issued by RTC), you need to register the Interrupt Handler.
  - ✓ Interrupt Handler (or ISR): Function that the kernel uses in response to a specific interrupt
    - Each device that generates interrupts has an associated interrupt handler
    - The interrupt handler for a device is part of the device's driver (kernel code that manages the device)

  - ✓ Registering an Interrupt Handler:
    - Each device has an associated driver. If that device uses interrupts (like most do), the driver must register one Interrupt Handler.
    - `request_irq()`: Register a given interrupt handler on a given interrupt line. Declared in `<linux/interrupt.h>`.
    - `free_irq()`: Unregister a given interrupt handler; if no handlers remain on the line, the interrupt line is disabled.

Function `request_irq()`

- Syntax:
```
int request_irq ( unsigned int irq, irq_handler_t handler, unsigned long flags,
                  const char *name, void *dev)
```
  - ✓ `irq`: interrupt number of the line. In some devices (e.g. system timer or keyboard), this value is typically hard-coded. Most other devices, it is probed or determined programmatically.
  - ✓ `handler`: function pointer to the actual interrupt handler that services this interrupt. This function is invoked whenever the OS receives the interrupt.
    - Prototype of the actual Interrupt Handler: `typedef irqreturn_t (*irq_handler_t) (int, void *)`
      - Type `irqreturn_t`: defined as a pointer to a function that returns a value of type `irqreturn_t` and takes two parameters `(int, void *)`.
    - Declaration of the actual interrupt handler: `static irqreturn_t intr_handler(int irq, void *dev)`
      - `irq`: numeric value of the interrupt line the handler is servicing.
      - `dev`: generic pointer to the same `dev` given to `request_irq()` when the interrupt handler was registered.
      - Returns `irqreturn_t` value: `IRQ_HANDLED` (if interrupt handler correctly invoked and its device is causing an interrupt), `IRQ_NONE` (if interrupt handler detects an interrupt for which its device was not the originator).
  - ✓ `flags`: 0 or bit mask of one or more flags defined in `<linux/interrupt.h>`. Example: `IRQF_DISABLED`, `IRQF_SAMPLE_RANDOM`, `IRQF_TIMER`, `IRQF_SHARED`.
  - ✓ `name`: device name associated with interrupt, e.g.: keyboard interrupt on PC: "keyboard". These text names are used by `/proc/irq` and `/proc/interrupts`.
  - ✓ `dev`: Pointer used for shared interrupt lines. Unique identifier used when the interrupt line is freed and that may also be used by the driver to point to its own private data (to identify which device is interrupting). If the interrupt is not shared, `dev` can be set to `NULL` if the line is not shared. A common practice is to pass the *driver's device structure*.
  - ✓ Return value: 0 on sucdess, otherwise an error.

- Example (RTC):
```
if (request_irq(rtc_irq, rtc_interrupt, IRQF_SHARED, "my_device", (void *)&rtc_port)) {
    printk(KERN_ERR "my_device: cannot register IRQ %d\n", irqn);
    return -EIO; }
```
  - ✓ `rtc_irq`: requested interrupt line (e.g., 8 for RTC, as RTC usually uses IRQ8 line).
  - ✓ `rtc_interrupt`: handler interrupt function.
  - ✓ `IRQF_SHARED`: flag that specifies that the line can be shared.
  - ✓ `"my_device"`: device name (e.g.: `/dev/rtc` for RTC)
  - ✓ `rtc_port`: pointer to *driver's device structure*.
  - ✓ If the handler returns 0, the handler has been successfully installed.

- Freeing an Interrupt Handler: To unregister an interrupt handler and disable the interrupt line, you can use:
```
void free_irq (unsigned int irq, void *dev)
```
  - ✓ If the specified interrupt line is not share, this function removes the handler and disables the line.
  - ✓ If the interrupt line is shared, the handler identified via `dev` is removed. However, the interrupt line is disables only when the last handler is removed.

- You can use `m /proc/interrupt` to see statistics related to interrupt on the system (devices , associated IRQ line numbers)