SCHOOL OF ELECTRICAL ENGINEERING AND TELECOMMUNICATIONS
ELEC4633: Real Time Engineering
Laboratory Experiment 1 - Multitasking

AIMS OF THE LABORATORY EXERCISE

This laboratory exercise is primarily concerned with various ways of creating tasks and threads within the Linux operating system and RTAI kernel. It is intended that it will provided you with the multitasking and task synchronisation tools necessary to assist in completing the aim of the lab program, that is, to design and implement an embedded real time control system. When you have completed this laboratory exercise, you should have learned the following:

- How to boot and use the Linux operating system and RTAI, and utilities such as the compiler and editors.
- How to write simple programs, compile and execute them from within Linux/RTAI.
- How to create separate processes using Linux/RTAI functions.
- How to run/install real-time tasks.
- How to provide communication/synchronisation between separately running real-time tasks.
- How to implement moderately complex multitasking applications.

LABORATORY CHECKPOINTS

Please make yourself aware of the checkpoints throughout this lab exercise. You will need to demonstrate your work to one of the demonstrators in order to get marked off for each checkpoint. Each checkpoint may attract a different mark. There are three (3) checkpoints in this lab exercise.

DEADLINE

The deadline for having the checkpoints in this experiment marked off will be your third lab period (Week 6 for even weeks and Week 7 for odd weeks). This lab will not be marked after this time.

PREPARATION

The lab program will require that you become familiar with the use and importance of RTAI quite quickly. Hence, this first lab exercise is very important. To help you to move through this exercise comfortably and in reasonable time, it is advised that you read the *RTAI Programming Guide*, available online. In particular, pay attention to the first three (3) Chapters (and even more specifically, Chapters 2 and 3). There will be some material or terminology in this document that you may not understand, however the concepts, and the fundamentals of building a simple real-time application should be reasonably clear. In addition, a tutorial will be given on the basics of real-time Linux versions.

Please ensure that you come to the labs with a laboratory notebook. This notebook will be used to document your work in the lab. It is an important skill for an engineer to be able to accurately, clearly, and succinctly document their work as they undertake it. This is what the notebook is for.

LABORATORY EXERCISES

Getting Started

When starting the computer, you will be given a several boot-up options. Choose *RTAI* for all work using Linux and real-time.

An *elec4633* account has been created on the machines which can be accessed by logging in as user: elec4633 with NO password. RTAI/Linux will boot up in graphical mode, so that logging in should put you into the *X Windows* environment. Further command line processing can be undertaken by starting a *terminal console*. The home directory is /home/elec4633. This is where you will do all of your work during the laboratory program.

Each lab exercise is a small programming task. All of the exercises can be undertaken at home if Linux/RTAI is installed, although you are encouraged to first confirm in the laboratory that you know how things work. It is a good idea to treat the programming as incremental. Whenever you write a program, start with something that is really simple - just a few lines - and test it. If necessary, you may have to create some simple code (like a printf() statement) to aid the test. Then add the next few lines, and test that they work ... as so on. You should never test more than a handful of lines at a time.

The laboratory exercises will help with this process - all of the exercises are small, but they build on each other.

Terminal Window

Most of the work you do in the lab will require you to use the terminal, or console, window. This provides you with command line processing. You can start a terminal window by either clicking on the terminal icon on the top menu bar, or by going to Applications, and then Accessories in the menu system. Once started you are free to used text based command line processing.

Editors

A range of editors are at your disposal including *gedit* and *vi*. For those not experienced with *vi*, *gedit* is a wise (graphical and more user friendly) choice. It may be invoked from the menu system under Applications, or by running gedit from the command line.

A First Program

```
#include<stdio.h>

int main(void)
{
    printf("Hello world\n");
    return 0;
}
```

Exercise

For your first Linux program (revision really!), make up a variant of this one. Have it display an identifiable message, or increment a counter, wait for a few seconds, and then repeat ... indefinitely. If not known, you will need to figure out how to "wait" inside the code. (See *Online manual* below)

Online Manual

In Linux, information on various commands as well as C programming code can be obtained through the online manual, by entering man command, eg. man printf, man gcc, man make. Use the online manual to look up the following commands:

ls	cd	cp	rm	mv
mkdir	mcopy	mdir	gcc	make
chmod	su	tar	lpr	lpq
mount	umount	more	less	grep
sleep	usleep	msleep		

Finding Functions

In Linux, information on various commands can be obtained through the online manual, by entering man command, eg. man printf, man gcc, man make. You can also use some of the other standard tools for finding things quite quickly and effectively. Try the following:

```
grep sleep /usr/include/*.h
less /usr/include/time.h
```

The first command will search for the character string *sleep* in all .h files in the include directory. The second command will allow you to peruse the file time.h. Once inside less try typing /sleep to find the string you are interested in.

Compiling Programs

You will soon want to create *Makefiles* and use the *make* facility to control compilation of programs. However, for the simple ones you are compiling now, you can do it from the command line by entering gcc -o junk junk.c where junk.c is the name of your code file, and junk will be the name of the executable. If you do not specify an executable file, then the default output file name a.out is created for you.

The GNU compiler *gcc* is a C/C++ compiler.

Consoles/Terminal Emulation

While in text mode, Linux/RTAI creates 6 log-in consoles when it boots. At any time, only one of them is in charge of the screen and keyboard. However, you can switch to any one of the four by pressing Alt-Fn where Fn is function key n and $0 \leq n \leq 6$. You need to login to each console that you wish to use.

Within X-Windows, consoles can be created using *Terminal Emulation*. This can be done using the menu system, or by 'clicking' the *Terminal emulation* button at the bottom of the screen.

To switch from the graphical X-Windows mode to text mode, simply type Ctrl-Alt-Fn, once again with Fn one of the first 6 function keys. To switch back from text mode to X-Windows mode, type Ctrl-Alt-F7.

Processes

Exercise

Programs within the present working directory (see man pwd), can be run by entering ./ followed by the program name (without spaces), followed by enter (what is the ./ for before the program name?). Once running, you can check on the processes which are executing in the machine, using the ps command. You will probably need to do this from a console other than that used to start your program.

Try running the hello world program that runs indefinitely, and then checking its status using ps.

Killing Processes

Exercise

Killing processes can be achieved through the command kill n, which terminates process number n (you can find the process number from the ps command). Alternatively, ctrl-c can also be used to terminate executing process. Make sure you can start and kill processes.

Task Creation

There are several ways to create new tasks.

- You can specify which tasks RTAI starts up during the boot process - look at some of the files in the /etc and /etc/init.d directories.
- You have multiple consoles, each of which support a login process, or several consoles within X-Windows.
- You can start a task (or multiple tasks) from the console by specifying an executable file.

When a file is executed from a console, it inherits the console, and further console operations are suspended until the task is terminated. However, you can ask the console to split itself in two, start the executable process and still provide normal console operations at the same time. You do this by entering filename & to start the new process, for example, emacs &.

Exercise

Start your program by entering filename &. You will see that you can use the console, but that the screen output is shared by the console and your program. Start the program again to see that there are two instances (processes) of the program running. Use ps and kill to control your program's execution - this time you do not have to change to another console.

This is an example of the fork() operation, and a good example of simple multitasking. fork() is a library function which requests the operating system to split the current process into two to create a new process.

Read the manual entry for the fork() operation. Create a simple program which uses *fork* to spawn a *child* process (there should be a *parent* and *child* process). Check there are two processes executing using ps. Can these processes share global data??

RTAI Tasks

Exercise

Now you want to create a program which creates two real-time tasks, each of which performs different things. **Again, refer to the *RTAI Programming Guide* and *RTAI Manual* online for further reference. In addition to these resources, a template is provided online for use in constructing a real-time task.**

This exercise requires some background in **RTAI**, **tasks/threads**, and **modules**. If you are not clear, the demonstrators will be available to provide you with some guidance.

Briefly, in RTAI, *tasks* can be created to run at precisely specified moments of time. The real-time programs created get loaded into the kernel as *loadable kernel modules*. Kernel modules are object files that can be dynamically loaded into the kernel address space and linked with the kernel code. That is, they exist as a part of the kernel (hence, special care must be taken). Modules are simply compiled (not linked) C programs which instead of having a main() function, have initialisation and cleanup functions, for example the init_module() and a cleanup_module() function. The *init_module* function is the function called when a module is *inserted* into the kernel, while the *cleanup_function* is the function called when *removing* modules from the kernel.

Real-time tasks or threads are like independently running programs and can be created within the initialisation function using the function rt_task_init() (or pthread_create() in RT-Linux). Use the online manual to see what this function does. Tasks can also be deleted inside the cleanup function using rt_task_delete() (or pthread_delete_np() in RT-Linux).

Real-time modules are handled by system utilities:

insmod	Install module
rmmod	Unloads module
lsmod	Lists loaded modules

To create and set up real-time tasks in RTAI, there are several RTAI function calls that are needed. They include:

rt_task_init	create a new task
rt_task_delete	delete a real-time task
rt_make_task_periodic	mark a task as periodic with a specified start time
rt_task_wait_period	suspend task execution until next period
rt_task_suspend	suspend a tasks's execution
rt_task_resume	resume execution of a suspended task

Read the manual entries in the RTAI manual on these function calls to see how they are used. Also, have a look to find some examples files to see how some of these functions are used. Ask the demonstrators where such examples may exist.

The specification for the first part of the exercise is:

- The program should create two *real-time* tasks.
- An integer variable will be created in the global data area. It will be initialized to 0.
- One of the tasks will increment the integer every n seconds (choose a suitable value for n).
- One of the tasks will display a message, which contains the integer value, every m seconds ($m \neq n$).

A template exists on Moodle (template.c) used for the creation of a single RT task that does nothing.

Note that printf will not work inside a real-time tasks. It can however be replaced by an equivalent function printk. printk works with exactly the same syntax as printf. However, it does not print to the terminal window, but rather to a log file. The log file can be viewed by typing dmesg, or you can see the output of the print statement by switching to console mode.

Compilation is carried out using the gcc compiler, however for your labs you will also use the *Make* facility. Provided a *Makefile* exists in the directory you are working in, and it is directed to compile your code, then you can compile by simply typing make in the command line. Note that an example template makefile for this lab is located on Moodle also. This should be downloaded and modified for each code you compile. Please ask the demonstrator if you require more assistance in using the Makefile.

Verify that the real-time program executes as expected.

Checkpoint #1 (2 marks): Obtain a signature and mark from your demonstrator.

Real-Time Task Communication

The use of the global memory variable is a sufficient means of communication between two real-time processes. However for communication between a real-time task and a *user space* or Linux (non real-time) task, global memory is not viable. RTAI provides several mechanisms for interprocess communication between real-time and non real-time processes. One of these is through FIFO (First In First Out) buffers, and another common method is the use of *shared memory*. The main advantage of shared memory is that it is not restricted to point-to-point communication. Shared memory can be written or read by any number of user or kernel processes.

To setup and use shared memory, the following commands are used:

rtai_kmalloc	allocate memory in real time task
rtai_kfree	free memory in real time task
rtai_malloc	allocate memory in user task
rtai_free	free memory in user task

Keep in mind the allocation needs to be consistent within both the real-time task and user task for communication to be successful. Using the RTAI manual or other RTAI reference documents, find out how to use the shared memory functions appropriately.

An alternative form of communication between tasks is via RT-FIFO's (First-In-First-Out data buffers). These are essentially linear buffers of which you can specify the size. As the name suggests you operate them on a first-in-first-out basis, so oldest data written to the buffer is read first. Data of different sizes are allowed to be placed on the FIFO. The functions that are needed to operate the RT-FIFO's include:

rtf_create	cretaes a RT-FIFO of specified size
rtf_destroy	removes an RT-FIFO
rtf_put	places data on the "end" of the FIFO buffer
rtf_get	retrieves data from the "front" of the FIFO buffer
open/read/write	function to operate on a FIFO within user space program

Again, refer to the online manual or RTAI reference documents to see how to use RT-FIFO's appropriately.

Exercise

From the previous exercise, create the two real-time tasks separately by constructing two independent modules, however using the same global counter variable. Compile and run each module to see the results.

Exercise

Now, retain the first real-time task, however implement the second task (which displays the counter) as a non real-time Linux task (containing *main*). An example template can be obtained in Moodle called user.c. Once again, you will need to compile them separately, run the real-time task by inserting it into the kernel, and *run* the non real-time task.

Exercise

You should find that the global memory counter variable does not work as a means of communicating between the real-time and non real-time tasks. Why is this the case? Alter your program so that the required counter is passed to the display task via *shared memory*. Confirm that the processes are running successfully.

Exercise

Repeat the above exercise, but now using RT-FIFO's instead of shared memory. Confirm that the processes are running successfully.

Checkpoint #2 (3 marks): Obtain a signature and mark from your demonstrator.

Real-Time Process Synchronisation

Process synchronisation is an important part of many, if not all, real-time systems. The ability to execute processes such that they are synchronised with others is crucial.

Exercise

Create three real-time tasks which are synchronised to execute one after the other. That is, the first task will be set up to run periodically. When this task is finished its code, it *wakes up* the next task, which in turn *wakes up* the last task when it has finished. The second and third task shouldn't have to be set up periodically. To do this have a look at the functions: rt_task_resume, and rt_task_suspend. Some thought may have to be given to the order in which tasks are created and deleted.

Test the code by printing a simple identification message in each tasks.

Checkpoint #3 (2 marks): Obtain a signature and mark from your demonstrator.

Exercise

As an additional exercise, try executing different delays in the "woken-up" tasks, as well as different priorities to see how it may effect the execution of all tasks together.