Embedded Systems with Linux

Programming the Beaglebone Board

Manuel Domínguez-Pumar

Embedded Systems with Linux series

Volume 1: Programming the Beaglebone Board

Manuel Domínguez Pumar

Electronic Engineering Department

Technical University of Catalonia – BarcelonaTech

First Edition – July 2018

ISBN 978-84-09-03813-8

AMG Editions, Barcelona, Spain

Artwork: JP Productions

Index

1 The Environment

The lab set up used includes a computer (host PC), a board for embedded system development (Beaglebone), a board for FPGA digital system development (DE2), a logic analyzer and other peripheral components. Let us describe the basic components: the host PC and the Beaglebone.

1.1 The Host PC

Use your student username and password to log into the host PC. The corresponding user's home directory is in the lab server.

1. Power on the computer and select *Ubuntu* as the operating system (this must be done twice). The distribution installed in the host PC is 64-bit Ubuntu release 14.01 LTS, being gnome the graphical environment.

2. Once in Ubuntu, create a specific directory to store the files of this lab course. You can do it either using the gnome file manager or opening a terminal session (press Ctrl-Alt-T) and using shell commands as follows:

```
host-pc$  cd ~
host-pc$  mkdir dsx
host-pc$  cd dsx
host-pc$  mkdir linux
host-pc$  mkdir bin
```

The default workspace for your Linux programming projects will be `dsx/linux`. Some specific programs and utilities will be placed into `dsx/bin`. Directories for other purposes will be added below `dsx`.

3. In order to execute the files placed in `dsx/bin` from any location in the PC, let us add this directory to the `PATH` environment variable. To this purpose, use `gedit` to open the hidden file `.basrhc` as follows:

```
host-pc$  gedit ~/.bashrc
```

This file sets some shell options each time a new `bash` session is opened.

4. In the editor window that opens, add the line shown below to the end of the file. Then save it (Ctrl-S) and close `gedit`.

```
export PATH=$HOME/dsx/bin:$PATH
```

5. Finally, close the terminal (type `exit` or close the window). Then open a new terminal (Ctrl-Alt-T) and examine the value of `PATH`:

```
host-pc$  echo $PATH
```

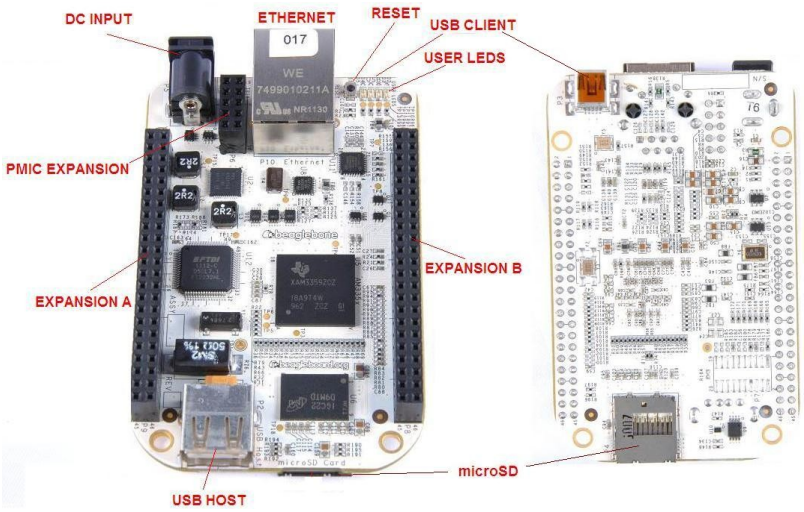
The `~/dsx/bin` directory, among others, should appear.

1.2 The Beaglebone board

The Beaglebone board was designed by a community of developers at Texas Instruments as an educational tool to teach open hardware and open source software capabilities for embedded systems.

Since Beaglebone is an open project, all hardware information is public domain; schematics, bill of components and materials and PCB (*printed circuit board*) layouts are available in the project's website www.beagleboard.org.

The figure below shows the main interfaces of the Beaglebone, while the table summarizes some key features.



Processor	AM3359 at 500 MHz / 720 MHz
Memory	256 MB DDR2 SDRAM SD/MMC micro SD connector
Power Options	USB connection 5V DC external jack
Power Management	TPS65217B Power Management IC (PMIC)
Board Features	Single cable integrated JTAG, serial port and USB 10/100 Ethernet Power expansion header
Processor Subsystems	176K ROM, 64K RAM 3D graphics engine LCD and touchscreen controller PRU-ICSS Real Time Clock (RTC) 2 USB ports, 1 Ethernet port Controlled Area Network (CAN) UART (2) McASPs (2) McSPI (2) I2C (3) Analog-to digital converter Enhanced Capture Module (3) Pulse width modulation (3)

The Beaglebone may be powered through:

- USB cable; then it operates at 500 MHz.
- 5V DC power jack; then it operates at 720 MHz.

The micro SD card works as hard disk for the board and hosts the operating system. The version used in this Laboratory is Ubuntu 14.04.4 LTS release, kernel 3.8.13.

The Beaglebone works as a standalone system, but it may be coupled with several daughter boards, called “capes”, and open source libraries to create a variety of custom embedded systems. The official cape catalogue can be seen at www.beaglebonecapes.com.

The picture below shows a cape board, the TT101, that has been specifically developed for this course. The TT101 adds a set of peripherals (LED matrix, position encoder, accelerometer, ADC, etc.) and connectivity resources (NFC, CAN and GPMC bus, etc.) to the Beaglebone system. This cape board will be used in next lab modules.

1.3 Information resources

The reference materials necessary for this Laboratory course can be found in atenea (http://atenea.upc.edu). This includes ...

- Documents, as this one, describing each lab module and compressed (tar) files that include programming projects.
- Data sheets of boards and components, such as the Beaglebone board, the AR-M335x processor, the TT101 cape, etc.

On the other hand, specific help about Linux commands, system calls, functions, etc. can be found here ...

- In the Linux Manual of the host PC.
- In the web page of the Open Group (which holds the Unix trademark):

  http://pubs.opengroup.org/onlinepubs/009604499/idx/index.html

2 Getting into the Beaglebone

In this Laboratory, we will use an Ethernet connection between the host PC and the Beaglebone. We will also use an USB cable to supply power to the board.

1. Connect the Beaglebone board to the host PC, both with the Ethernet and the USB cables and allow the board some time to boot.

In the Ethernet connection, the host PC has the static IP address 192.168.1.1 while the Beaglebone has 192.168.1.100. The host PC also works as gateway to other networks.

2. In the host PC, open a new terminal session (Ctrl-Alt-T) and check the configuration of the Ethernet interface as follows:

```
host-pc$  ifconfig eth1 | grep inet
```

3. Use `ping` to check if the Beaglebone is available:

```
host-pc$ ping 192.168.1.100
  PING 192.168.1.100 (192.168.1.100) 56(84) bytes of data.
  64 bytes from 192.168.1.100: icmp_req=1 ttl=64 time=0.432 ms
  64 bytes from 192.168.1.100: icmp_req=2 ttl=64 time=0.434 ms
  ...
  ^C
```

4. Now you can open remote sessions with the Beaglebone using `telnet`, `ssh`, etc. Let us use `ssh` to log into the Beaglebone as user `ubuntu`:

```
host-pc$  ssh ubuntu@192.168.1.100
  ubuntu@192.168.1.100's password:
```

5. Type `temppwd` as password, press return and then you're in & ready to work!

6. Note that Beaglebone Ubuntu looks like any other Linux environment. The default shell is `bash` and therefore most of the well-known Linux shell commands work here in the same familiar way. Try them!

3 Beaglebone Linux

3.1 First steps

As commented above, the Ubuntu version installed in the Beaglebone looks & works as any other Linux system, well almost! In order to reduce size, utilities such as the graphical environment or the Linux Manual are not included. Another difference is that a set of common Linux commands do not exist as separated binary files, but concentrated into a single multi-call file, named *busybox*. Other common Linux commands exist as standalone binary files, but being simplified versions of the usual ones.

1. Let us first check the file systems. In a terminal session with the Beaglebone, type the following:

```
beaglebone$  df
  Filesystem     1K-blocks   Used Available Use% Mounted on
  /dev/root        3686872 599292   2896964  18% /
  devtmpfs          124256      4    124252   1% /dev
  none                   4      0         4   0% /sys/fs/cgroup
  none               24880    212     24668   1% /run
  none                5120      0      5120   0% /run/lock
  none              124384      0    124384   0% /run/shm
  none              102400      0    102400   0% /run/user
```

This command shows the disk occupation and availability on all currently mounted file systems. In the example, all file systems reside in the SD card (a 4GB device). You can find help information about this command (and also about any other command) by typing the name of the command followed by `--help`.

2. Now examine the contents of the root directory:

```
beaglebone$ cd /
beaglebone$ ls -l
  total 68
  drwxr-xr-x  2 root   root    4096 Apr  2 17:50 bin
  drwxr-xr-x  4 root   root    4096 Jan  4  2017 boot
  drwxr-xr-x 12 root   root    3480 Apr  2 17:48 dev
  drwxr-xr-x 80 root   root    4096 Apr  2 17:48 etc
  drwxr-xr-x  3 root   root    4096 Apr  2 17:50 home
  drwxrwxr-x 15 ubuntu ubuntu  4096 Oct 18  2016 lib
  drwx------  2 root   root   16384 Jan  4  2017 lost+found
  drwxr-xr-x  2 root   root    4096 Apr  2 17:45 media
  ...
```

3. Take a look into some of these directories. Note that their contents are those typical of any Linux system: `/bin` contains system commands and programs, `/usr` contains libraries, programming tools and non-system applications, `/home` contains users' directories (for instance, `ubuntu`), `/dev` contains device files, `/etc` contains system configuration files, etc.

4. Note also that some useful shell features, such as the command history, available through the arrow keys, or the auto-fill property, available using the TAB key, also work in this environment. Try them!

3.2 Redirection and pipes

Redirection and pipes are key features of Linux systems. Both are extremely useful in Linux programming, as we will see in this course.

1. Go to your home directory `/home/ubuntu`, then ceate the directory `prova` and get into it:

```
beaglebone$  cd ~
beaglebone$  mkdir prova
beaglebone$  cd prova
```

2. Test the `echo` command with something like this:

```
beaglebone$  echo This works!
```

This command takes the string passed as parameter and writes it to the application's standard output file. When executing a program from the shell, the standard output is by default the terminal. Thus, you see the phrase repeated (echoed) on the screen.

3. Redirect the `echo` standard output to a file named *first*:

```
beaglebone$  echo This works! > first
```

The file *first* has been created; dump its contents into the terminal using the `cat` command.

4. Repeat the previous step with a different phrase and note that, if the file *first* already exists, it is overwritten.

5. Redirect other data into the file *first*, this time as follows:

```
beaglebone$  ls -l / >> first
```

This kind of redirection adds (appends) the output data to the end of *first*: now it additionally contains the list of directories at the root level. To see it, use the `more` command to dump the contents of *first* into the terminal.

6. It is also possible to redirect the contents of a file into the standard input of a program. Here we use the contents of *first* as input for the command `grep`:

```
beaglebone$  grep Apr < first
```

The `grep` (*general regular expression parser*) command searches inside the file *first* and displays all lines containing the string "Apr".

7. The following set of commands displays the total number of files (including directories) at the root directory level. The `wc` (*word count*) command counts the number of text lines contained in the file *second*.

```
beaglebone$  ls -l / > second
beaglebone$  wc -l < second
  19
beaglebone$  rm second
```

All this can be done in a single step using a pipe (|), as follows:

```
beaglebone$  ls -l / | wc -l
  19
```

The pipe (|) is a FIFO-like feature that redirects the standard output of the `ls` command into the standard input of the `wc` command, skipping the need for the explicit use of intermediate files such as *second*. In general, pipes are a common way to communicate data between processes.

8. Write a unique command line that counts the number of directories and files in the `/etc` directory with names starting by 'e'.

9. The `tee` command allows to both redirect the standard output of a program to a file and also pipe it to the standard input of another program, as shown in this example:

```
beaglebone$  ls -l | tee third | wc -l
```

Explain what is the purpose of this command line.

10. Finally, remove the directory `prova` and all its contents:

```
beaglebone$  rm -r ~/prova
```

3.3 Processes

A process is an instance of an executing program. When a program is executed, the kernel loads it into memory, allocates space for the variables and stores information about the process. This information is available in “file” format in subdirectories below `/proc`.

1. Using the `ps` command, list the (relatively) large set of processes currently running in the board:

```
beaglebone$  ps aux | more
```

The information displayed about each process (line) includes the identification number (PID), the CPU and memory occupation, the current status, etc.

2. Execute the `yes` command. Note that it does nothing but writing repeatedly the character 'y' on the standard output, that is, on the terminal screen. Press Ctrl-C to terminate the process:

```
beaglebone$  yes
  y
  y
  ...
  ^C
```

3. Now execute this “noisy” command as follows:

```
beaglebone$  yes > /dev/null &
  [1] 1133
```

The symbol '&' means that the process is to be executed in background mode, leaving the terminal free for other work. On the other hand, redirecting data to the device file `/dev/null` is a common way to discard such data.

4. In this example, '1' is the job number and '1133' the PID of the `yes` process. You can use `ps` to check that the process is executing ...

```
beaglebone$ ps -aux | grep yes
  ubuntu   1133 98.1  0.1   2904  480 pts/0  R   19:35  0:42 yes
  ubuntu   1135  0.0  0.2   3296  704 pts/0  S+  19:36  0:00 grep yes
```

but also as follows ...

```
beaglebone$ jobs
  [1]+  Running         yes > /dev/null &
```

This command provides information about the status of all the user's processes executing in background.

5. You can terminate the process either with `kill -9 1133` or with `kill %1`.

```
beaglebone$  kill %1
beaglebone$  jobs
  [1]+  Terminated      yes > /dev/null
```

Important: From now on, we assume that:

- The Ethernet link between host PC and Beaglebone works OK.

- We have opened a terminal session with the host PC. Shell commands are denoted with the prompt `host-pc$`.

- We have opened a terminal session with the Beaglebone. Shell commands are denoted with the prompt `beaglebone$`.

4 Programming the Beaglebone

Up to now, we have managed to communicate with the Beaglebone and work with shell commands. Now it's time to set up the environment that will allow us to develop applications for this embedded system.

4.1 Cross-compilation

The objective is to obtain binary files to be executed on the Beaglebone system, based on an ARM335x processor, but it makes little sense to install all the compilation environment (which includes all tools and the disk space necessary to develop our programming projects: editors, compilers, libraries, etc) in the target system. It is more efficient and comfortable to work in the host PC.

Summarizing, we will develop our projects in the host PC and use a cross-compiler to generate the executable files for the target embedded system. This cross-compiler executes on Intel 386x based systems but generates binary code for ARM 335x based systems.

The cross-compiler is already installed on the host PC. Retrieve the value of the PATH variable and, among others, obtain the paths for the compiler executable and library files:

```
/usr/local/gcc-linaro-arm-linux-gnueabihf-4.9-2014.09_linux/bin
/usr/local/gcc-linaro-arm-linux-gnueabihf-4.9-2014.09_linux/arm-
   linux-gnueabihf/lib
```

4.2 Mounting file systems

At this point a problem arises: our programs are to be compiled on the host PC, but they must be executed on the Beaglebone. Sending the binary files through the Ethernet connection is not an efficient solution.

To this purpose, let us use the NFS (*network file system*) services to mount a Beaglebone directory into the file system of the PC; the directory is physically in the Beaglebone, but it is “seen” by the PC as one of its own file systems.

Although Linux has a standard procedure to mount filesystems, let us use here a specific utility named `sshfs`.

1. In the Beaglebone terminal, create the following directories:

```
beaglebone$  cd ~
beaglebone$  mkdir mnt
beaglebone$  mkdir mnt/bin
```

2. In the host PC terminal, the mounting point-directory is `~/dsx/mnt`. Create it and take in mind that this directory must be always empty before mounting.

3. In the host PC terminal, execute the following command:

```
host-pc$  sshfs ubuntu@192.168.1.100:/home/ubuntu/mnt ~/dsx/mnt
```

As in all ssh-related utilities, you will be asked to type the password for the remote user `ubuntu`.

4. Once the password is provided, the Beaglebone directory `/home/ubuntu/mnt` is seen as `~/dsx/mnt` in the host PC. Use the command `df` to check it:

```
host-pc$  df
  Filesystem        1K-blocks     Used Available Use%  Mounted on
  /dev/sda3         236435184  6417464 217984440   3%  /
  none                      4        0         4   0%  /sys/fs/cgroup
  udev                1008740        4   1008736   1%  /dev
  tmpfs                203908     1100    202808   1%  /run
  none                   5120        0      5120   0%  /run/lock
  none                1019532      152   1019380   1%  /run/shm
  none                 102400       48    102352   1%  /run/user
  /dev/sda2         240178696 29862612 198092648  14%  /home
  ubuntu@192.168.1.100:/home/ubuntu/mnt
                      3686872   610800   2885456  18%  /home/joan/dsx/mnt
```

5. Now the contents of the mounted directory are the same than in the original Beaglebone file system. Moreover, if we change such contents on the host PC, they are automatically seen in the Beaglebone and vice-versa. Try it!

In fact, all changes in the contents of the mounted directory are automatically updated on the other side through the Ethernet connection. This process is transparent to the user.

4.3 A first program: Hello World!

Let us now compile & execute a program, the traditional “Hello World”.

1. Download the file `lm01.tar` from atenea and store it in the directory `~/dsx/linux` of the host PC. This file contains the project.

2. If not already done, open terminal sessions both in the host PC and in the Beaglebone and mount the directory `/home/ubuntu/mnt`.

3. In the host PC, uncompress `lm01-hello.tar`:

```
host-pc$  cd ~/dsx/linux
host-pc$  tar xvf lm01.tar
  ...
host-pc$  rm lm01.tar
```

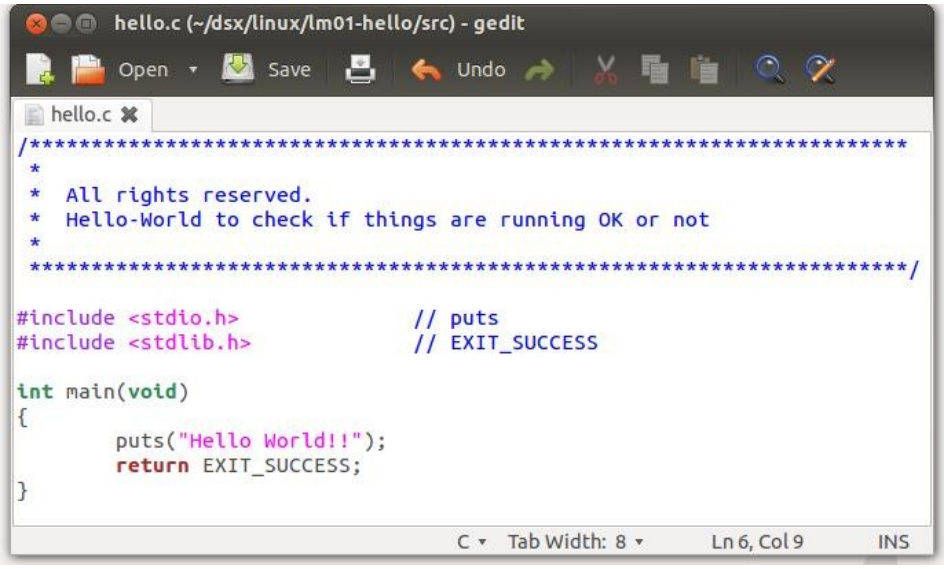
4. The directory `~/dsx/linux/lm01-hello` has been created. It contains the file `Makefile` and the directories `src` and `include`. Source files (only `hello.c` in this case) are in `src`. User's include files for compilation purposes (none in

this case) must be placed in `include`. Finally, `Makefile` manages the compilation calls and sets the appropriate options for the current project.

5. Use the `more` command to take a close look into the contents of `Makefile`. Note that, depending on an environment variable named `LOCAL`, it performs compilation for the ARM335x (Beaglebone) or for the 386x system (host PC).

6. Open the (unique) source file of the project, `hello.c`. Since it contains plain text, you are free to use any text editor. In this example we use `gedit`:

```
host-pc$  gedit ~/dsx/linux/lm01-hello/src/hello.c &
```



The file is automatically recognized as C source code and color formatting is used to display it. Anyway, this time no changes must be done, so close *gedit*.

7. To compile the project for the Beaglebone, go to the directory `lm01-hello` and proceed as follows:

```
host-pc$  make clean
  rm -rf /home/joan/dsx/linux/lm01-hello/src/*.o
      /home/joan/dsx/linux/lm01-hello/hello
  ...
  Project clean is done

host-pc$  make
  rm-linux-gnueabihf-gcc -c -Wall -O2 -I. -Iinclude src/hello.c -o
  src/hello.o
  arm-linux-gnueabihf-gcc -lpthread  src/hello.o -o hello
  cp hello /home/joan/dsx/mnt/bin

  Cross Compilation is done
  export LOCAL=yes to do local compilation
```

The `make clean` command deletes unnecessary files of the current project (i.e. those from a previous compilation) and updates the remaining ones. It is highly recommended to execute it prior to each new compilation.

8. If no compilation errors arise, the `make` command creates the executable file (*hello*) for the ARM-based system and writes it both in `~/dsx/linux/lm01-hello` (the project directory) and `~/mnt/bin` (mounted from the Beaglebone).

```
host-pc$  ls -l ~/dsx/linux/lm01-hello
  total 20
  -rwxrwxr-x 1 joan joan 7503 feb 11 16:11 hello
  drwxr-xr-x 2 joan joan 4096 feb 11 16:11 include
  -rwxr-xr-x 1 joan joan 2714 feb 11 16:11 Makefile
  drwxr-xr-x 2 joan joan 4096 feb 11 16:11 src

host-pc$  ls -l ~/dsx/mnt/bin
  total 4
  -rwxrwxr-x 1 joan joan 7503 feb 11 16:11 hello
```

9. To execute the program, go to the terminal open in the Beaglebone and simply proceed as follows:

```
beaglebone$  cd ~/mnt/bin
beaglebone$  ./hello
  Hello-world!!
```

10. Let us now create a simple script file allowing to clean the contents of the directory `/home/ubuntu/mnt/bin` from anywhere in the Beaglebone. To this purpose, follow these steps:

a. Create the text file `clearbin` with this shell command as content:

```
rm -r /home/ubuntu/mnt/bin/*
```

b. Using `chmod`, set `clearbin` as an executable file for all system users (owner, group and others).

c. Move `clearbin` to the target directory `/usr/bin`; which is in the default search path.

d. Check that `clearbin` works as expected.

11. Create, compile and execute a new version of the “Hello-World” project. When executing the program, it must ask the user to select a language to display the message (i.e. English, Catalan, Spanish ...). Create a new specific directory tree for this new project, i.e. `lm01-hello2`.

5 Leaving the Beaglebone

1. Once finished working with the Beaglebone, you must halt the system by typing ...

```
beaglebone$  sudo init 0
```

Provide the password for the user `ubuntu`. Then wait until all activity on the board ceases (lights off) and finally unplug USB and Ethernet cables.

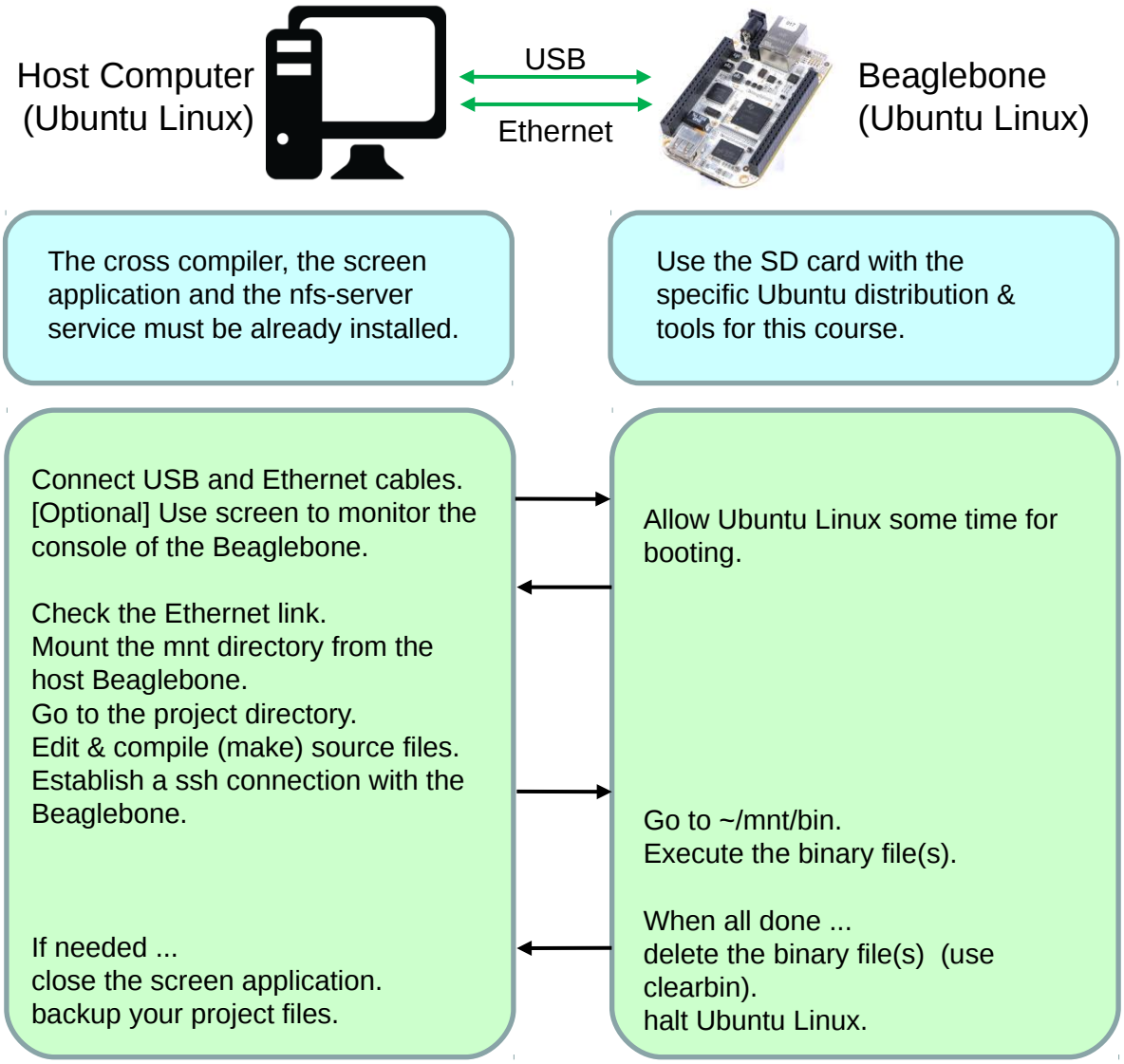
2. The `tar` command can be used to store complete directory structures in a unique file while preserving file metadata such as owners, permissions, etc. As an example, let us do it for the directory `lm01-hello2`:

```
host-pc$  cd ~/dsx/linux
host-pc$  ls -l
  ...
  drwxrwxrwx 4 joan joan 4096 feb  4 10:45 lm01-hello2/
  ...
host-pc$  tar cvf lm01-hello2.tar lm01-hello2
  lm01-hello2/
  lm01-hello2/hello
  lm01-hello2/Makefile
  lm01-hello2/src/
  lm01-hello2/src/hello.c
  lm01-hello2/src/hello.o
  lm01-hello2/include/
host-pc$  ls -l
  ...
  drwxrwxrwx 4 joan joan  4096 feb  4 10:45 lm01-hello2
  -rw-rw-r-- 1 joan joan 20480 feb  6 12:03 lm01-hello2.tar
  ...
```

Now you can copy the file lm01-hello2.tar to its target destination: a backup directory, an USB pen drive, etc.

6 How-to summary

The figure shown below summarizes the setup, the tools needed and the standard procedures we are going to use in this course in order to compile our projects for the Beaglebone board.



7 Handling files in Linux

This laboratory module focus on the system calls used to perform file I/O, a key topic in Linux, which follows the “everything is a file” philosophy: regular files, directories, FIFOs, pipes, devices, sockets, etc. are seen as files by users and applications.

Such philosophy leads to universal I/O: the same system calls, namely *open, read, write,* and *close,* can be used to work with all types of files. In practice, the kernel translates such calls into filesystem or driver operations that perform appropriate I/O on the target device.

7.1 Files and file descriptors

In Linux, a regular file is organized as a linear sequence of bytes, called a byte stream. No further organization or format is expected.

Files may be read or written byte-to-byte, starting at a specific position, the *file offset*. When a file is first opened, the *file offset* is zero (first byte), and it varies sequentially according to the number of bytes read or written. In some cases (i.e. disks or tapes), files can also be randomly accessed. Writing a byte to any position within a file overwrites the byte previously located there.

A file can be opened more than once by multiple processes running simultaneously or even by the same process. This implies that user programs must ensure that concurrent file access is properly synchronized.

Each open instance of a file is given a *file descriptor*, a positive integer. The *file descriptor* is the unique cookie used by most I/O system calls to refer to open files. Normally, a process inherits three open file descriptors when it is started by its parent:

- Descriptor 0: standard input (file from which the process reads input data).
- Descriptor 1: standard output (file to which the process writes output data).
- Descriptor 2: standard error (file to which the process writes error messages and other notifications).

In interactive shells or programs, these file descriptors are normally connected to the controlling terminal, so that by default data input is read from the keyboard and data output is printed on the screen.

7.2 Performance

Using file I/O system calls can be inefficient, essentially for two reasons:

- System calls are time-expensive because they imply switching from running your program to executing kernel code and going back to your program again. Reducing the number of system calls used in a program and doing as much work as possible on each call are good ideas.
- The hardware imposes restrictions on the size of data blocks that can be trasferred at one time. This is the case of the block size in filesystems, which can be typically 1024, 2048 or 4096 bytes. Using multiples of the block size in data transfers is also a good idea.

Linux provides standard libraries that allow higher-level interfacing to devices and disk files than system calls. An example is the standard I/O C library, *stdio*, that provides buffered output: programs can write data in blocks of arbitrary sizes, and the library functions provide the system calls with full blocks as the data becomes available. This strongly increases the system call efficiency.

7.3 Handling errors

Checking and handling errors is a must. System calls and functions notify the occurrence of an error by returning a specific value (usually -1) and setting the global variable *errno*, of type integer. The value of *errno* maps to the text description of a specific error.

The C library provides functions to translate *errno* to its corresponding textual description, being *perror()* the most common:

```
#include <stdio.h>
#include <fcntl.h>

void perror(const char *str);
```

This function prints to standard error output the string pointed at by *str*, followed by a colon and the string representation of the error described by *errno*. It is useful to include in *str* the name of the function that failed.

A common mistake when checking *errno* is to forget that any library or system call can modify it. For example, this code is buggy because the value of errno can be modified by *printf*:

```
if (close (fd) == −1) {
    printf ("Ooops! Something failed! See below...\n");
    perror ("close");
    exit(EXIT_FAILURE);
}
```

Other C library functions available for error reporting purposes are *strerror()* and *strerror_r()*.

In the next laboratory modules, we will introduce & include in the programming projects an specific utility (*fatal.c)* to more comfortably handle fatal errors. This becomes more useful as complexity of such projects increases.

8 Basic File I/O System Calls

8.1 Open, read, write & close

Let us briefly describe the four key system calls for performing file I/O: *open(), read(), write()* and *close()*. Programming languages employ these calls only indirectly, via their own I/O libraries.

open()

Purpose: open an existing file or create and open a new file.

```
#include <sys/stat.h>
#include <fcntl.h>

int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
```

This system call maps the file given by *pathname* to a file descriptor. On success, *open()* returns a file descriptor, an integer used to refer to the open file in subsequent calls. On error, returns –1 and sets *errno*.

The *flags* argument is mandatory. It specifies the file access mode, as one of the following values:

Flag	Access mode
O_RDONLY	Open file only for reading
O_WRONLY	Open file only for writing
O_RDWR	Open file for reading and writing

If the file does not exist, *open()* may create it accordingly to additional values of the *flags* argument (see appendix). When *open()* creates a new file, *mode* specifies the access permissions (see appendix). However, *mode* should be omitted when a file is not being created.

The examples below illustrate some uses of *open()*:

```
//  Open the existing file “myfile” only for reading
fd = open("myfile", O_RDONLY);

// Open new or existing file “myfile” for reading and writing
// If if already exists, truncates to zero bytes. Permissions are rw- --- --- .
fd = open("myfile", O_RDWR | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR);

// Open new or existing file “myfile” only for writing in append mode.
// If if already exists, truncates to zero bytes. Permissions are rw- --- --- .
fd = open("myfile", O_WRONLY | O_CREAT | O_TRUNC | O_APPEND,
                    S_IRUSR | S_IWUSR);
```

read()

Purpose: read data from the open file with descriptor *fd*.

```
#include <unistd.h>

ssize_t read(int fd, void *buff, size_t len);
```

This call reads up to *len* bytes from the open file referenced by *fd* and stores them in the memory pointed at by *buff*. On success, it returns the number of bytes actually read; this number is 0 when no further bytes could be read (EOF reached). On error, the call returns -1 and sets *errno*.

A call to *read()* may read less than the requested number of bytes. This is not an error. For a regular file, this means that we are close to the end of the file.

For other types of files—pipes, FIFOs, sockets or terminals—there are various circumstances where fewer bytes than requested may be read. For example, a *read()* from a terminal reads text only up to the next new line (\n) character.

write()

Purpose: write data to the open file with descriptor *fd*.

```
#include <unistd.h>

ssize_t write(int fd, void *buff, size_t len);
```

A call to *write()* writes up to *len* bytes from the memory pointed at by *buff* to the open file referenced by the file descriptor *fd*. On success, the call returns the number of bytes actually written, which may be less than *len*. On error, the call returns -1 and sets *errno*.

When performing I/O on disk files, a successful return from *write()* does not mean that the data has been effectively transferred to disk, because the kernel performs buffering in order to reduce disk activity and expedite *write()* calls.

close()

Purpose: close the open file with descriptor *fd*.

```
#include <unistd.h>

int close(int fd);
```

A call to *close()* must be done after all I/O has been completed. This call unmaps the open file descriptor *fd* and disassociates the file from it.

When the execution of a process ends, all of its open file descriptors are closed automatically. Anyway, to close explicitly unneeded file descriptors and handle potential errors is recommended practice.

8.2 Laboratory work

Download the file *lm02.tar* from atenea. Expand it in the directory *~/dsx/linux*. The project directories *lm02-copy*, *lm02-fileio* and *lm02-rconds* are created.

Let us illustrate the basic use of file I/O system calls. Go to the *lm02-copy* directory. The program *copy.c* is a simplified version of the shell command *cp*: it copies the contents of an existing file, *oldfile*, to a new one, *newfile*, according to the following syntax:

```
./copy oldfile newfile
```

```
// ------------------------------------------------------------------------------------ copy.c starts
#include <sys/stat.h>    // open
#include <fcntl.h>       // open
#include <unistd.h>      // read, write
#include <stdio.h>       // printf
#include <stdlib.h>      // exit
#include <string.h>      // strcmp
#define BUF_SIZE 1024

int koerror(char *message) {
      perror(message);
      exit(EXIT_FAILURE);
}

int main(int argc, char *argv[])
{
      int infd, outfd, openFlags;
      mode_t filePerms;
      ssize_t numRead;
      char buf[BUF_SIZE];

      if (argc != 3 || strcmp(argv[1], "--help") == 0) {
                        printf("command usage: ./copy oldfile newfile \n");
                        exit(EXIT_FAILURE);
      }

      /* Open input and output files */
      infd = open(argv[1], O_RDONLY);
      if (infd == -1)
                        koerror("opening infile");

      openFlags = O_CREAT | O_WRONLY | O_TRUNC;
      filePerms = S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH;
      outfd = open(argv[2], openFlags, filePerms);
      if (outfd == -1)
                        koerror("opening outfile");
```

```
        /* Transfer data until EOF or error */
        while ((numRead = read(infd, buf, BUF_SIZE)) > 0)
                        if (write(outfd, buf, numRead) != numRead)
                                koerror("couldn't write whole buffer");
        if (numRead == -1)
                        koerror("read");

        /* Close input and output files */
        if (close(infd) == -1)
                        koerror("close input");
        if (close(outfd) == -1)
                        koerror("close output");

        exit(EXIT_SUCCESS);
}
// ------------------------------------------------------------------------------ copy.c ends
```

This program performs some error handling after each system call: if it returns -1, an error message is displayed from *perror()* (accordingly to a string provided to such funcion and the actual value of *errno*) and the execution ends.

Note that the syntax of the command line is also checked. If it is wrong, or help was requested (./copy --help), the program displays the right syntax.

If *newfile* does not exist, it is created; if it already exists, it is truncated to zero length. Access permissions are “rw- rw- rw-”, thus everybody can read from or write to the file.

8.2.1 Task: How it works

1. Connect the Beaglebone and the host PC, open a terminal session on each side and mount the /home*/ubuntu/mnt* directory of the Beaglebone in the *~/dsx/mnt* directory of the host PC.

2. In the host PC, compile *copy.c* and copy the files *txtfile* and *largefile* to the same target directory as the executable file.

```
host-pc$ cd ~/dsx/linux/lm02-copy
host-pc$ make
  …
host-pc$ cp largefile txtfile ~/dsx/mnt/bin
```

3. In the Beaglebone, go to the directory */home/ubuntu/mnt/bin*. Examine *txtfile* using the shell command *cat*. Then execute *copy*:

```
beaglebone$ cd /home/ubuntu/mnt/bin
beaglebone$ ./copy txtfile txtfile2
beaglebone$ ls -l txt*
-rw-r--r-- 1 ubuntu ubuntu 1126 Apr  2 17:55 txtfile
-rw-r--r-- 1 ubuntu ubuntu 1126 Apr  2 17:56 txtfile2
```

4. Check if the file created, *txtfile2*, is exactly as expected. If not, explain why.

5. Execute the following command, then explain the results:

```
beaglebone$ ./copy txtfile2 txtfile2
```

6. Modify *copy.c* to work in append mode: if *newfile* already exists, the contents of *oldfile* should be written at the end of *newfile*.

8.2.2 Task: Time performance

Let us now test, in a simple way, the time performance of the program. To this effect, we will copy a large file and use the *time* shell facility to measure how long the program takes to execute.

1. In the Beaglebone, go to the directory */mnt/nfs_pc/bin*, execute the original version of *copy* and check the results as follows:

```
beaglebone$ time ./copy largefile largefile2
  real      0m1.011s
  user      0m0.000s
  sys       0m0.200s
beaglebone$ ls -l large*
  -rw-r--r-- 1 ubuntu ubuntu 7633039 Apr  2 17:25 largefile
  -rw-r--r-- 1 ubuntu ubuntu 7633039 Apr  2 17:28 largefile2
```

Note that *largefile* was successfully copied to *largefile2*, and that it took the times displayed.

2. Change the BUF_SIZE parameter from 1014 to 16 and compile *copy.c* again. Then perform the same operations as in the previous step. Do you see any difference? If yes, explain what could be the cause(s).

9 More about File I/O

9.1 Changing the File Offset

For each open file, the kernel records a *file offset*: the file location at which the next *read()* or *write()* operation will occur. When a file is first opened, the *file offset* points to the start (first byte) of the file.

Successive *read()* and *write()* calls progress sequentially through the file and set the *file offset* to the next byte after the ones last read/written. Such sequential behavior can be modified using *lseek()*.

lseek()

Purpose: set the *file offset* of an open file with descriptor *fd*.

```
#include <unistd.h>

off_t lseek(int fd, off_t offset, int whence);
```

This call sets the position in the file where the next read or write will occur. The *offset* parameter is an integer and *whence* specifies how *offset* is to be used.

On success, *lseek()* returns the new value of the *file offset,* measured in bytes from the beginning of the file. On error, it returns -1 and sets *errno*.

The parameter *whence* must be one of the following:

whence	***offset***
SEEK_SET	Offset is an absolute position
SEEK_CUR	Offset is relative to the current position
SEEK_END	Offset is relative to the end of the file

Note that *offset* must be non-negative when SEEK_SET is choosen. Otherwise, *offset* can be either positive or negative.

9.2 Laboratory work

Let us show an example of using *lseek()*. Consider the project stored in the *lm02-fileio* directory. There, the program *file_sr.c* opens the file *filename* and performs the sequence of operations specified by the command-line arguments, according to the following syntax:

```
./file_sr filename {r|s} …
```

The meaning of each r|s argument is:

- **r** : Read & display bytes from current file offset.
- **s** : Set a new file offset, referred to the first byte of *filename*.

```
// ------------------------------------------------------------------------------------ file_sr.c starts
#include <sys/stat.h>    // open
#include <fcntl.h>       // open
#include <unistd.h>      // read, lseek
#include <stdio.h>       // printf, scanf, argc, argv
#include <stdlib.h>      // exit, free, atol, malloc
#include <string.h>      // strcmp

int koerror(char *message) {
    perror(message);
```

```
        exit(EXIT_FAILURE);
}

int main(int argc, char *argv[]) {
        size_t len;
        off_t offset;
        int fd, ap, j;
        char *buf, lenin[16];
        ssize_t numRD;

        if (argc < 3 || strcmp(argv[1], "--help") == 0) {
                        printf("command usage: ./file_sr filename {r|s} {r|s} ...\n");
                        exit(EXIT_FAILURE);
        }

        fd = open(argv[1], O_RDONLY);
        if (fd == -1)
                koerror("open");

        for (ap = 2; ap < argc; ap++) {
                switch (argv[ap][0]) {

                    case 'r':   // Read & display text from current file offset
                        printf("Number of bytes to read?\n");
                        scanf("%s", lenin);
                        while ( (len = atol((const char *) &lenin)) == 0 ) {
                                printf("Illegal value. Try again!\n");
                                scanf("%s", lenin);
                        }

                        buf = malloc(len);
                        if (buf == NULL)
                                koerror("malloc");

                        numRD = read(fd, buf, len);
                        if (numRD == -1)
                                koerror("read");
                        else if (numRD == 0)
                                printf("%s: end-of-file\n", argv[ap]);
                        else {
                                printf("%s:\n", argv[ap]);
                                for (j = 0; j < numRD; j++) {
                                        printf("%c", buf[j]);
                                }
                                printf("\n");
                        }

                        free(buf);
                        break;

                    case 's':    // Change file offset
                                printf("Seek offset?\n");
                                scanf("%ld", (long *) &offset);
                                if (lseek(fd, offset, SEEK_SET) == -1)
                                        koerror("lseek");
                                printf("%s: seek succeeded\n", argv[ap]);
                                break;
```

```
                    default:    // Argument is not r, nor s
                                printf("Argument must be {r|s}, not %s\n", argv[ap]);
                                break;
                }       // end switch
        }       // end for
        exit(EXIT_SUCCESS);
}
// ------------------------------------------------------------------------------------- file_sr.c ends
```

9.2.1 Task: How it works

1. If not already done, connect the Beaglebone and the host PC, open a terminal session on each side and mount the directories as in previous works.

2. In the host PC, compile *file_sr.c*.

```
host-pc$ cd ~/dsx/linux/lm02-fileio
host-pc$ make
```

3. In the Beaglebone, execute the program on the file *txtfile* applying any sequence of s|r arguments separated by blank spaces:

```
beaglebone$ cd /ubuntu/mnt/bin
beaglebone$ ./file_sr txtfile s r r ...
```

For each s|r argument, the program asks the user to introduce a new *file offset* (when s) or the number of text bytes to read (when r) at the current *file offset*. Try it at your own!

4. Answer & justify the following questions: What happens if a wrong syntax is introduced? Is the *file offset* modified by consecutive reads? How can we read the entire file? Is it possible to read beyond the last byte of the file?

9.2.2 Task: Improvements

Warning: Create a new project directory for each improvement.

1. Modify *file_sr* to obtain the *file_srw* utility, a program that opens *filename* and performs on it the following operations:

- **r** : Read & display bytes from current file offset.
- **s** : Set a new file offset, referred to the first byte of *filename*.
- **w**: Write a text string to the current file offset.

At each write (**w**) operation, the program must prompt the user to type the corresponding text string. The maximum size of such string is 128 characters.

2. [Optional] Modify *file_srw.c* to obtain the *file_srwi* utility, a program that opens *filename* and performs on it the following operations:

- **r** : Read & display bytes from current file offset.
- **s** : Set a new file offset, referred to the first byte of *filename*.
- **w**: Write a text string to the current file offset.
- **i**: Insert a text string to the current file offset.

At each insert (**i**) operation, the program must prompt the user to type the corresponding text string. The maximum size of such string is 128 characters.

9.3 Race Conditions

To avoid race conditions is essential in Linux programming. Those are situations where the result produced by two processes operating on a shared resource depends on the relative order in which they gain access to the CPU.

Focus on the project stored in the *lm02-rconds* directory. The code *open_ko.c* generates a situation where such indesired effect occurs. The program tries to ensure that it is the only one that creates a new file. It does it by trying to *open()* the file once without *O_CREAT;* if the call succeeds then the file already exists and the program ends. If not, the program assumes that it can create the file, pauses for N seconds (i.e it does other useful work) and calls *open()* again, now to create the file using *O_CREAT*.

However this mechanism fails if another process creates the same file between the two calls to *open()*. This can happen if the kernel decides that the process time slice has expired and gives control to another process. The result depends on the order of scheduling of the two processes, so this is a race condition.

```
// ------------------------------------------------------------------------------ open_ko.c starts
#include <sys/stat.h>    // open
#include <fcntl.h>       // open
#include <stdio.h>       // printf
#include <stdlib.h>      // exit, atol
#include <unistd.h>      // close, sleep, getpid

int main(int argc, char *argv[])
{
    int fd;  long N;

    if (argc != 3) {
        printf("command usage: ./open_ko filename N\n");
        exit(EXIT_FAILURE);
    }

    fd = open(argv[1], O_WRONLY);     // Open 1: check if file exists
    if (fd != -1) {
        printf("[PID %ld] File %s already exists\n", (long) getpid(), argv[1]);
        close(fd);
    } else {
        printf("[PID %ld] File %s doesn't exist\n", (long) getpid(), argv[1]);
        if ((N= atol(argv[2])) > 0) {
            sleep(N);                 // Suspend execution for N seconds
            printf("[PID %ld] Done sleeping\n", (long) getpid());
```

```
                }
                // Open 2: create the file
                fd = open(argv[1], O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);
                if (fd == -1) {
                        perror("open2");
                        exit(EXIT_FAILURE);
                }
                printf("[PID %ld] Created File "%s"\n", (long) getpid(), argv[1]);
        }
        exit(EXIT_SUCCESS);
}
// ------------------------------------------------------------------------------ open_ko.c ends
```

9.3.1 Task: How it works

1. If not already done, connect the Beaglebone and the host PC, open a terminal session on each side and use sshfs to mount the directory /home/ubuntu/mnt of the Beaglebone in the directory ~/dsx/mnt of the host PC.

2. In the host PC, compile *open_ko.c*.

```
host-pc$ cd ~/dsx/linux/lm02-rconds
host-pc$ make
```

3. In the Beaglebone, run two instances of the program trying to open the same file, but with different sleep times:

```
beaglebone$ cd /home/ubuntu/mnt/bin
beaglebone$ ./open_ko tatata 10 &
  [1] 386
  [PID 386] File tatata doesn't exist
beaglebone$ ./open_ko tatata 0
  [PID 387] File tatata doesn't exist
  [PID 387] Created File tatata
beaglebone$ [PID 386] Done sleeping
  [PID 386] Created File tatata
  [1]+  Done
beaglebone$
```

Note that both processes claim that they have created the file. This is obviously wrong!

4. Modifiy *open_ko.c* so that, although still using the naïve “two calls to *open()*” method, the race condition seen in the example above disappears.

10 Appendix. Flags and modes for *open()*

This table summarizes the flag values that can be used in the *open()* system call (more details about some of them on next page):

flag	Purpose
O_RDONLY	Open file only for reading
O_WRONLY	Open file only for writing
O_RDWR	Open file for reading and writing
O_CLOEXEC	Set the close-on-exec flag
O_CREAT	Create file if it doesn't already exist
O_DIRECT	File I/O bypasses buffer cache
O_DIRECTORY	Fail if pathname is not a directory
O_EXCL	With O_CREAT: create file exclusively
O_LARGEFILE	Used on 32-bit systems to open large files
O_NOATIME	Don't update file last access time on read()
O_NOCTTY	Don't let pathname become the controlling terminal
O_NOFOLLOW	Don't dereference symbolic links
O_TRUNC	Truncate existing file to zero length
O_APPEND	Writes are always appended to end of file
O_ASYNC	Generate a signal when I/O is possible
O_DSYNC	Provide synchronized I/O data integrity
O_NONBLOCK	Open in nonblocking mode
O_SYNC	Make file writes synchronous

These flag values are divided into three groups:

- **Access mode flags**: O_RDONLY, O_WRONLY, and O_RDWR. They can be retrieved using the *fcntl()* F_GETFL operation.
- **Creation flags**: From O_CLOEXEC to O_TRUNC. They control various aspects of the behavior of *open()*, as well as options for subsequent I/O operations. These flags can't be retrieved or changed.
- **Status flags**: From O_APPEND to O_SYNC. They can be retrieved and modified using the *fcntl()* F_GETFL and F_SETFL operations.

When a file is created using O_CREAT, the *mode* parameter of the *open()* call sets the file permissions. The possible values for *mode* are OR (|) combinations of the following:

	Read	Write	Execute
Owner	S_IRUSR	S_IWUSR	S_IXUSR
Group	S_IRGRP	S_IWGRP	S_IXGRP
Others	S_IROTH	S_IWOTH	S_IXOTH

For example,

```
open("myfile", O_CREAT, S_IRUSR | S_IWUSR | S_IXOTH);
```

creates the file *myfile*, with read and write permission for the owner and execute permission for others (rw- --- --x).

The user mask variable (*umask*) also affects the permissions of created files in such a way that the flags of the *open()* call should be understood as requests to set permissions. Whether those permissions are set or not depends on the current value of *umask*.

Finally, descriptions of some flags follow:

O_APPEND
: The file will be opened in append mode: before each write, the file position will point to the end of the file.

O_ASYNC
: A signal will be generated when the file becomes readable or writable. Available only for FIFOs, pipes, sockets, and terminals.

O_CLOEXEC
: Upon executing a new process, the file will automatically be closed.

O_CREAT
: If the file does not exist, it will be created. If the file already exists, this flag has no effect unless O_EXCL is also given.

O_DIRECT
: The file will be opened for direct I/O.

O_EXCL
: When given with O_CREAT, this flag will cause the call to *open()* to fail if the file already exists. This is used to prevent race conditions on file creation. If O_CREAT is not also provided, this flag has no meaning.

O_NONBLOCK

If possible, the file will be opened in nonblocking mode. No operation will cause the process to block (sleep) on the I/O. This behavior may be defined only for FIFOs.

O_SYNC

The file will be opened for synchronous write: no write operation will be complete until the data has been physically written to disk. Read operations are already synchronous, so this flag has no effect on reads.

O_TRUNC

If the file exists and the given flags allow for writing, the file will be truncated to zero length. This flag is only for regular files.