Analysis of Techniques for Linux Kernel Device Driver Programming

by

ANTONI MARTÍ COLL

directed by

MANUEL M. DOMINGUEZ PUMAR

FINAL WORK, 36 CREDITS

OCTOBER 2016

Escola Tècnica Superior d'Enginyeria de Telecomunicació de Barcelona

UNIVERSITAT POLITÈCNICA DE CATALUNYA

i. About this work and organization 7
ii. Tools used for the creation of this work 9
1. Introduction 11
1.1. Motivation for this work 11
1.2. About GNU/Linux 12
1.3. GNU/Linux: current usage statistics 13
1.4. Relevance of the GNU project in modern IT 14
1.5. Devices powered by Linux 16
2. Mounting a bootable Ubuntu/Linux image 19
2.1. Description of needed components 20
2.2. Working environment set-up 22
2.3. Configuring VM workspace 24
2.3.1. Tuning VM 24
2.3.2. Needed tools description 26
2.3.3. Install system utilities 26
2.3.4. Install Device Tree Compiler 26
2.3.5. Configure git command line tools 27
2.3.6. Creating work directory structure 27
2.3.7. Download cross-compiler 27
2.4. Bootable image preparation: download and compile 28
2.4.1. Download and compile boot-loader 28
2.4.2. Download, configure and compile Kernel 29
2.4.3. Download prepared Ubuntu 14.04 image for ARM 30
2.5. Bootable image preparation: image assembly 32
2.5.1. Formatting, installing the bootloader and partitioning SD-card 33
2.5.2. Mount SD-card and prepare image 34
2.5.3. Image configuration 34
2.6. Bootable image testing 36
2.6.1. Connection to BeagleBone 36
2.6.2. Mount BeagleBone file system as a local folder 36
3. Kernel Programming Techniques 37
3.1. Reference material used .... **Error! Bookmark not defined.**
3.2. Guides organization and common treats 38
3.1.1. Assumed directory structure 39
3.1.2. Boilerplate code for all developed drivers 39
3.1.3. All global variables declared static 40

3.1.4. Trace mechanism 40
3.3. GPIO And Interrupts Guide 42
3.3.1. Interrupts..... 42
3.3.2. Who provides each ISR code? 43
3.3.3. Basic interrupt management in Linux 43
3.3.4. Basic API for interrupt management from drivers 44
3.3.5. Notes on IRQ management from a driver 45
3.3.6. Mapping between interrupt number and IRQ number 47
3.3.7. GIPO usage from drivers 47
3.3.8. Event aware drivers..... 49
3.3.9. GPIO and interrupts 51
3.4. File Drivers Guide 54
3.4.1. Introduction..... 54
3.4.2. Miscellaneous driver overview 55
3.4.3. Simple implementation of a miscellaneous driver..... 57
3.5. Blocking I/O Guide..... 61
3.5.1. Introduction..... 61
3.5.2. Considerations for blocking I/O from a driver point of view 62
3.5.3. Keeping low resource usage during blocking I/O..... 62
3.5.4. Wait queues 66
3.5.5. Blocking I/O with polling 67
3.5.6. Blocking I/O with interrupts and wait queues 68
3.5.7. Interruptible wait queues..... 69
3.6. Kernel Probes Guide..... 72
3.6.1. Introduction..... 72
3.6.2. General mechanism 72
3.6.3. kprobes..... 74
3.6.4. jprobes..... 75
3.6.5. Return probes..... 75
3.6.6. No need for instrumented functions' source code 76
3.6.7. kprobes API 76
3.6.8. About commented usage examples..... 80
3.6.9. Commented kprobes usage example..... 81
3.6.10. Commented jprobes usage example..... 83
3.6.11. Commented return probes usage example 85
3.7. Thread Local Storage (TLS) Guide..... 89

3.7.1. Introduction ... 89
3.7.2. Outline of TLS' inner working ... 92
3.7.3. Restrictions to the usage of TLS ... 94
3.7.4. The API for TLS ... 94
3.7.5. GCC altrerntive to TLS API ... 96
3.7.6. Example code 1: usage of a global variable ... 96
3.7.7. Example code 2: usage of TLS ... 98
3.7.8. Exercises ... 101
3.8. DMA Transfers Guide ... 103
4. Conclusions ... 128
4.1. The path to the guides ... 128
4.2. Retrospect about Kernel code digging ... 128
5. Bibliography ... 130
5.1. Introduction bibliography ... 130
5.2. Mounting a bootable Ubuntu/Linux image bibliography ... 131
5.3. Guides bibliography ... 131

i. About this work and organization

Having completed the U.P.C. course called ***D****igital* ***S****ystems for Embedded Linu****x*** (DSX), an interest about Linux and device drivers has been growing for me.

This interest finally materialized as the desire to start and develop the PFC project as a personal research project to gain further knowledge on Linux programming, covering some kernel-space concepts. So in some sense this work is kind of a continuation to the work done during DSX course using the BeagleBone board.

First part of this work is a small overview of what is GNU/Linux, along with some context including statistics and some of its selected applications, which show the diversity of purposes for which Linux is being used nowadays apart from classical server/desktop computing.

Then, a description of the working environment is given, including the steps needed to set up a virtualized development machine.

Next, programming guides (prepared specifically for this work) with proposed exercises (anyone who is reading this work and willing to put in practice the concepts exposed here) related to various aspects of kernel programming are included.

The guides cover advances programming aspects related to kernel-side such as:

- GPIO programming
- interrupt programming
- blocking I/O programming
- file drivers
- kernel probes

Also, and specifically related to the BeagleBone, a guide which covers programming of the DMA controller found at the Board is included.

As a *bonus track*, a guide (not related to kernel programming) which covers a topic not seen during the DSX course is included: TLS.

Finally, a conclusion to the presented work is exposed, encouraging readers to explore other advanced topics.

ii. Tools used for the creation of this work

Tools are an essential part of any digital work, and following have been the ones used:

Following hardware has been used:

- old PC: laptop HP Pavilion DV3, Intel Core i5 with (8 GB RAM, 500 GB HDD) with Windows 7 Home Premium
- new PC: laptop Dell XPS13 (9350), Intel Core i7 (16 GB RAM, 500 GB HDD) with Windows 10 Home
- both with external mouse (Logitech) and keyboard (Dell)

Following software has been used:

- Oracle Virtual Box with Ubuntu 14.04 guest as development machine (both freely available for non-commercial usage)
- Vim for edition of all code presented in this document (freely available)
- Microsoft Word for the edition of this document (work license)
- yED for the creation of vector diagrams embedded into this document (freely available for non-commercial usage)
- Firefox as the internet browser (freely available)
- Notepad++ for taking quick notes (freely available)
- and the set of command-line utilities bundled with Ubuntu 14.04 and some of its packages (freely available)

Following third party services have been used:

- Drop Box for regular backups
- Google Drive for sporadic backups

Apart from the links explicitly referenced though this work, following sources have been consulted to obtain information:

- Linux Kernel 3.8 source code
- man pages
- Texas Instruments Forums
- The AMM335x processor programming manual
- Material from the course on Digital Systems for Embedded Linux (DSX)
- And finally google to search for generic topics

1. Introduction

1.1. Motivation for this work

It all begin at a course given at U.P.C. called Digital Systems for Embedded Linux (DSX), where some programs were developed for the BeagleBone board, illustrating that this embedded device can effectively power a Linux distribution, with less than 1 GB RAM and a permanent storage of about 4 GB (using a SD-card), featuring Ethernet connectivity.

Its size makes it fit in a hand:

In addition to *playing* with the BeagleBone, some practices included developing programs for the TT01-v1 Cape (which is a board add-on) developed by a previous student featuring as fancy gadgets as a push button, some individual LEDs, a LED matrix, an accelerometer and an external 16-bit memory bus:

Approximately in the middle-end of the DSX course, a driver had to be programmed. It was a simple file driver for Linux, meant to interface a user-space program with external VHDL-programmed board through the BeagleBone (using its integrated General Purpose Memory Controller (GPMC) via the TT011-v1 Cape connector).

Apart from the interest of programming the VHDL code, one aspect of those exercises was fascinating: programing a driver.

In that course, some techniques were learnt, including mapping physical memory addresses to virtual processor addresses and a slight introduction to file drivers. But the most important lesson was it was not hard to write your own driver for Linux.

So many possibilities opened… lots of techniques to explore and an open-source Kernel to research how all these techniques do work.

Then, the idea for this work was born: extend the knowledge (related to drivers) acquired during the DSX course, and extend it in such a way new students to that course would obtain a benefit for it. This led to think that the guide format used at this work was the most suitable for the goals of the work.

This work originally started with the desire to be able to perform a DMA transfer from the external memory (via GPMC bus) into some kernel-allocated-memory and expose it in such a way that a person who does not know what does *DMA* mean could follow the reasoning and why thinks are done how they are.

This is just the same process that happens to people when they learn some new concept:

- first, there is the will to learn something
- then, this will to learn shows you that there are some other concepts which are pre-requisites for the original wanted-to-learn

As it happens usually, the will to learn about DMA transfers led to first learn some other aspects related to driver development which are essential to program a decent DMA transfer: extend knowledge about file drivers, learn what an interrupt is, how blocking I/O is done…

And as a side-effect, some other concepts entered scene just to be able to test the acquired knowledge: interrupts triggered by GIPO to be able to alter behavior of the tested programs, kprobes check code flow…

And slowly, this work started to grow, to become what is now.

All explanations done in this work could not have been done without the effort and uncountable hours spent by a huge community to give form to what is one of the biggest distributed projects nowadays.

Welcome to Linux.

1.2. About GNU/Linux

About 1984, GNU Project had a goal: create a free Unix-type operating system (OS). So, the GNU Project created some guidelines which all GNU-like OS should follow, known as *Free System Distribution Guidelines*.

Basically, it states about those OS distributions that '*They will reject nonfree applications, nonfree programming platforms, nonfree drivers, nonfree firmware "blobs", nonfree games, and any other nonfree software, as well as nonfree manuals or documentation.*'

Nowadays, there are a bunch of OS which follow the GNU Principles of digital freedom.

Linux itself officially started on August 1991, with a message from Linus Torvalds on a computing Usenet board (*comp.os.minix*), announcing an early development stage of its Kernel.

Although the GNU Project had its own planned replacement for its operating system Kernel called HURD, nowadays Linux is usually bundled as the core for the operating systems endorsed by the GNU Project. This way:

- the GNU Project provides an ecosystem of helper utilities (command line utilities, compilers, editors…) needed to interact with the operating system
- the Linux Project provides the GNU operating systems with a modern Kernel

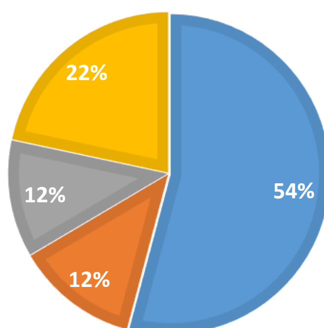
It's not uncommon to call the entire operating system a *Linux OS*, but it should be called a *GNU/Linux OS*.

Of course, there are many distributions of GNU/Linux, each of them having its own particularities and advantages for a specific purpose, but the GNU Project only endorses a small fraction of them (only those which strictly follow the principles of free software promoted by the GNU Project). Other distributions (although they may have more relevance on Google queries) are not endorsed by the GNU Project.

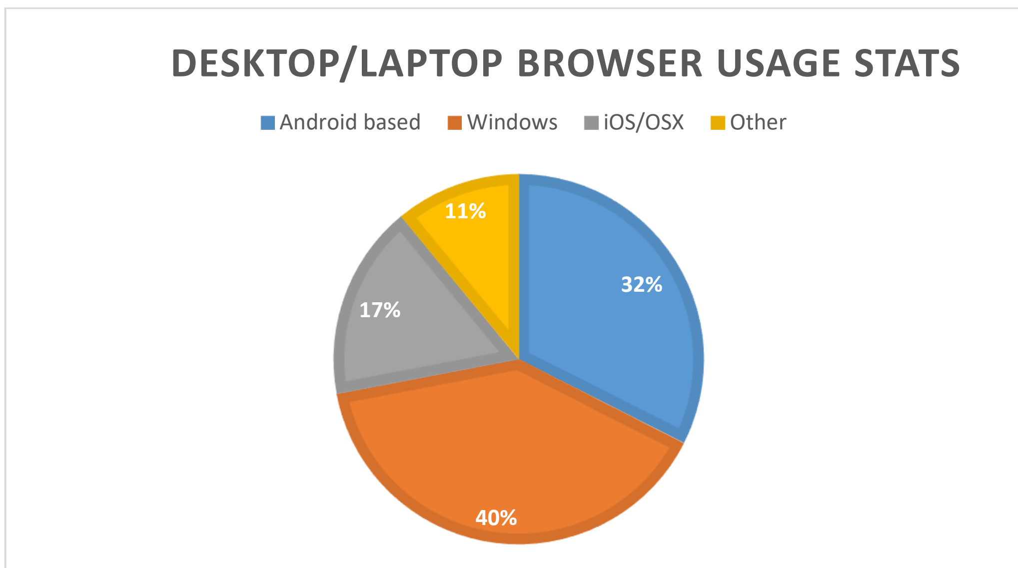
1.3. GNU/Linux: current usage statistics

Since its introduction in 1991, Linux has slowly found its way among all areas related to computing.

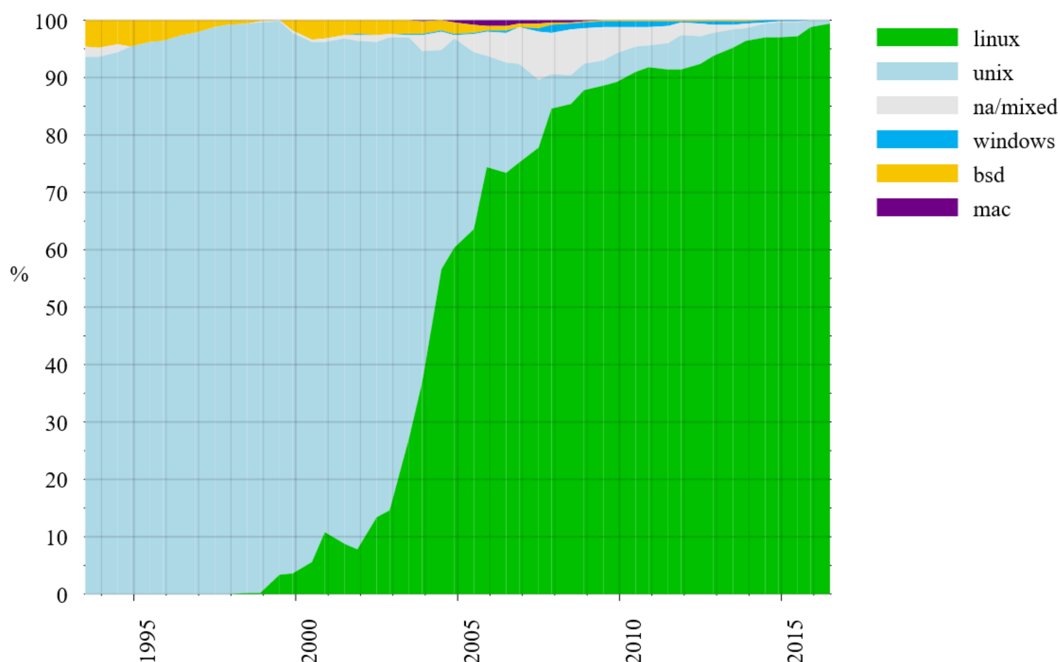
One very successful introduction at the everyday users' life from Linux is to be the powering kernel for all Android devices on the market. Considering all personal computers, Gartner gives following market share for consumer electronic devices (corresponding to PCs + tablets + smartphones) up to 2015, where it can be appreciated the Linux relevance.



If only desktop and laptop computers are considered for personal computing, an estimate based on web browser statistics (this is, based on the user agent perceived by web traffic analyzers) gives the following chart, where it can be seen that Linux is not so spread at home computers.



Another area of interest is supercomputing. Evolution of the systems which power supercomputers through the last two decades shows how Linux has implacably taken over and displaced the rest of the systems:



1.4. *Relevance of the GNU project in modern IT*

One of the main achievements of the GNU project was to (thanks to its continuous striving) be able to assemble an entire family of operating systems and make them available not only to the general public but also to developers and researchers.

Another great achievement accomplished by the GNU Project is that the term free stated in the *Free System Distribution Guidelines* includes not only free availability and

usage for the computer programs created using these guidelines, but also free availability of the source code which compiles into those programs' binary files, with licenses like the Gnu Public License (GPL).

It has been of tremendous importance the GPL, which states that if a binary/executable file (in such a way that all or parts of its source code contains any piece of code licensed as GPL (including static and dynamic linking depending the GPL license version) or derived from GPL-licensed code) is distributed to the public, this public also has the right to obtain the source code that generates the binary file (thus enforcing open-source politics).

This may seem very like a very aggressive licensing option from an IT company point of view, because code open-sourceness may represent a strategic weakness for the companies.

Because of this, some less restrictive (not requiring open-sourceness for derivative works derived of licensed code) aroused, such as the MIT License, Apache2 License, etc.

Slowly people from all around the world started joining the Linux project (imagine it, to be able to inspect the source code of a modern OS Kernel and tweak it just for study purposes), and today more than 1000 developers contribute to the release of each version of the Linux Kernel.

GPL-like licenses contributed not only to accelerate knowledge sharing between programmers all around the world, they also contributed to grow the *sharing feeling* between programmers (of course GPL license is not the reason that led programmers to share source code, but Linux is a powerful demonstration of what open-source along with collaborative programming can achieve, and impressed many people who were later open-source coders).

It's enough to take at websites like SourceForge or GitHub to be amazed by the activity of open-source projects related to Linux available out there, including Linux.

So nowadays a combination of open/closed source policies is found between IT companies. Some selected examples are:

- traditional companies which produce closed source products

  It's not uncommon that these companies use open-source auxiliary components like web servers, Linux development tools, GNU Toolchains and so on.

- small companies which produce open source programs or libraries and sell supporting/branding/consulting/customization services related to their product

- companies which do not produce software but offer consulting services on some software products (for either or both open-source / closed-source products)

- companies not directly related to IT with very small operational budgets which can't afford buying proprietary licenses, so they use free IT products in conjunction with some customization and/or tailoring

1.5. Devices powered by Linux

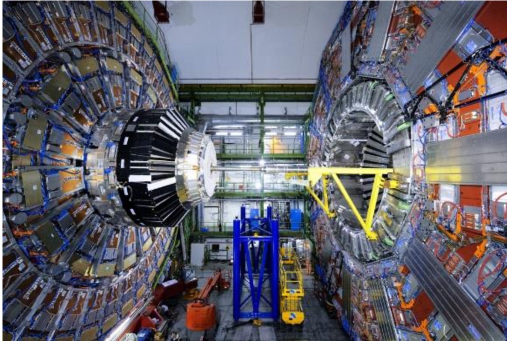
In contrast with proprietary OS components (such those from the wide spread Microsoft and Apple), the kernel offered by Linux is so customizable that it has been tailored to work in devices which traditionally had custom-made solutions.

To name a few, here follow some applications powered by Linux:

- Routers like Linksys WRT54G, powered by the DD-WRT distribution: this is a classic among enthusiasts of home networking, and was very popular about a decade ago at early stages of networking communities such as guifi.net (they even created a distribution base on DD-WRT) and were commonly used as access network nodes

- airplane entertainment systems, which pursue low costs as well as high degrees of customization. Some companies which use Linux for this goal are Virgin America, Emirates, United and Air New Zealand

- related to scientific application, some example facilities which at some degree use Linux (formerly the Scientific Linux Distribution) are Large Hadron collider and the International Space Station



- Toyota Prius (a self-driving car by google), is powered by an OS based on Ubuntu/Linux

- BeagleBone.org Foundation, which is a US-based non-profit corporation devoted to promote open-source software and hardware in embedded computing, offers some embedded boards on top of which Linux can be run:

2. Mounting a bootable Ubuntu/Linux image

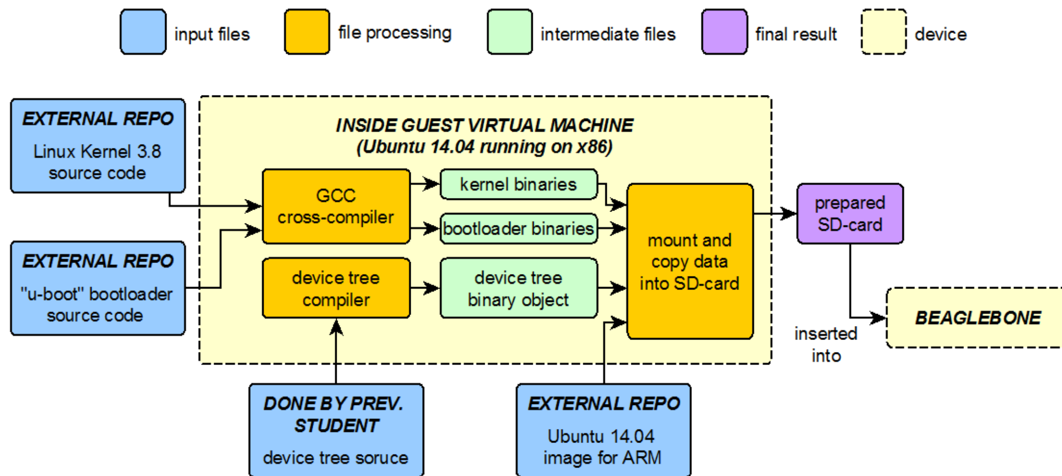
This section describes the steps needed to prepare a development machine aimed at:

- preparing a bootable of Ubuntu/Linux for the BeagleBone
- being able to compile the code dedicated to kernel-side driver programming

The steps are described in a detailed way, and include:

- how to download the packages needed to install and set up the machine
- description of desired networking layout
- description and installation of the tools needed to compile binaries for the target platform (the BeagleBone)
- description and installation of other tools which are needed to prepare and mount the bootable image

2.1. Description of needed components



Previous diagram is based on work done by R. González Muñoz [1], which may be consulted for reference. Nevertheless, the image build process will be described here for completeness.

Blue boxes on the diagram correspond to various pieces of software which are already prepared and serve as input to the image build process:

- Linux Kernel 3.8 source code, where primary reason for choosing this kernel version is that, as it is stated in [1] (section 4.2.2), "*Nowadays, if it is required to use capes in the BeagleBone, the best option is to use the 3.8 branch*". As a previous step to its compilation, some parameters will be tweaked.

- Bootloader source code, which is a bootloader targeting embedded boards based (among others) on ARM.

- Device Tree Source, because to enable usage of the TT01v1 cape for the BeagleBone (designed by R. Pérez López), a file (the Device Tree Binary Object, DTBO) describing how the Cape Manager found at the kernel must configure the system in order to allow it interacting with the cape hardware must present at the cape. DTBO is generated from the Device Tree Source (DTS).

- Ubuntu 14.04 Image for ARM, which was the latest Ubuntu image available when this work started

Dark yellow components on the diagram correspond to processes which need to be executed as steps in the image build process:

- GCC cross-compiler, as the source code for the Kernel must be compiled and it will be executed on the BeagleBone processor, a compiler which runs on the VM guest (which is x86/x64) but generates code for an ARM processor is needed.

This is known as a cross-compiler. This compiler will be used later to generate the executable files and drivers during the completion of the exercises' guides.

- Device Tree Compiler, which compiles Device Tree Source files (DTS) into Device Tree Binary Objects, which are the files accepted by the Cape Manager found at the Linux Kernel. Although there is also an option to compile DTS files directly from the BeagleBone, in order to avoid bloating the BeagleBone with additional software, this step will be done from the VM guest (the development machine)

Purple component on the diagram corresponds to the final result of the image build process:

- Prepared SD-Card, which on the DSX labs were 4 GB cards. Unfortunately, during the development of this work there were no 4 GB cards available for purchase, so for the prototyping a 16 GB card was used with satisfactory results

Green components on the diagram correspond to intermediate results which feed the process that generates the result:

- Compiled Kernel, which will be copied into the SD-card ready to be loaded from the already prepared Ubuntu 14.04 ARM image
- Device Tree Binary Object, which will be copied into the SD-card and the boot loading process instructed to load it to enable access to the cape hardware

Now, the process for generating a working Ubuntu image for the Beagle Bone will be illustrated. The explanation is highly based on the work done by a previous student [1].

The build process will be explained from the point a fresh Ubuntu 14.04 OS installed on the guest system. For completeness on the process' description, excerpts from [1] may be included here, with including additional information.

2.2. Working environment set-up

Code presented at the guides has been developed using a VM:

- Host OS for the VM was Windows 10 64-bit (coding started on an HP Pavilion DV3 and finished on a Dell XPS13)
- Guest OS for the VM was Ubuntu 14.04 Desktop LTS 64-bit

The software used to set up the VM is Oracle Virtual Box (chosen based on good personal experience with it in the past and because its free availability for this kind of work). Setup of the virtual machine is quite straightforward.

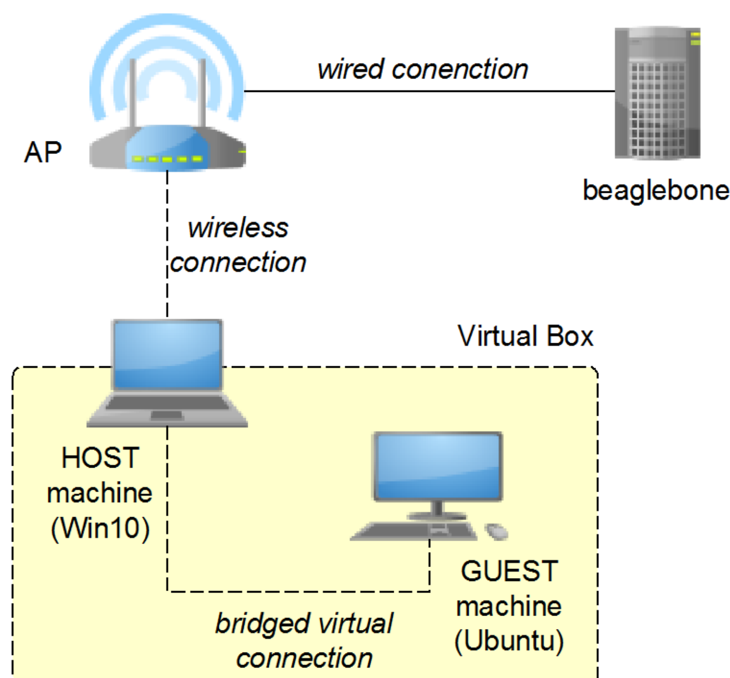
Oracle Virtual Box can be downloaded from the following webpage:

https://www.virtualbox.org/wiki/Downloads

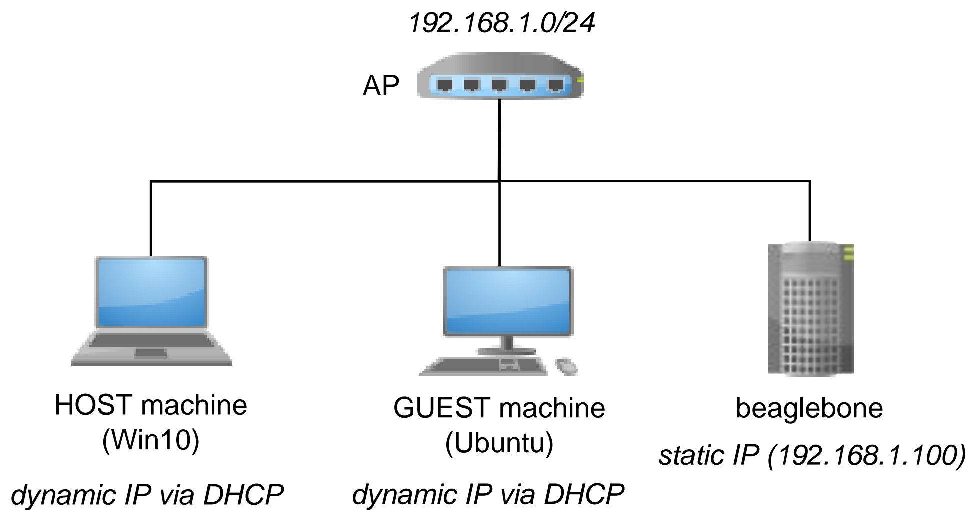
Once Oracle Virtual Box was installed, a 64-bit VM with following features was created (if the host OS is 64-bit and runs on a 64-bit PC, it's better to create 64-bit VM's):

RAM	2048 MB
Processors	2 (not really needed, but will speed up kernel compilation
Storage	HDD with 40 GB
Network type	Bridged adapter
Video	128 MB RAM, 3D acceleration enabled

Regarding network configuration, the physical connection scheme between all system's components has been the following:



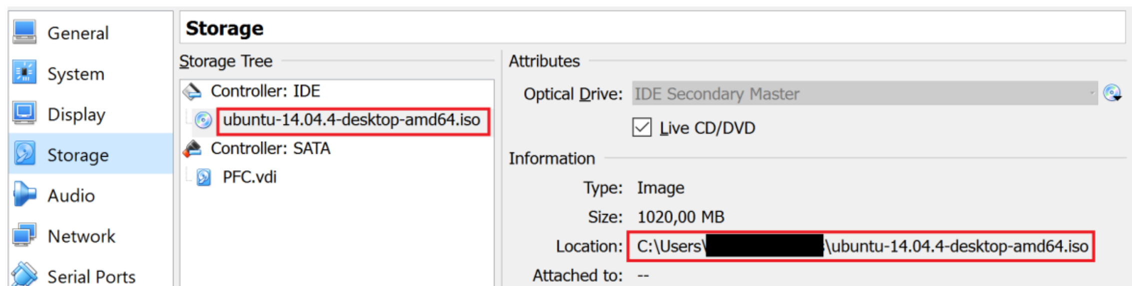
Thanks to the use of a *bridged* network configuration, the logical connection scheme was the following (which avoided usage of NAT techniques between the guest and the host and thus allowed direct bidirectional IP communication between the guest machine and the BeagleBone):



When this work started, latest available LTS version of Ubuntu Desktop was 14.04, whose ISO image can be downloaded from the following webpage:

http://releases.ubuntu.com/14.04/

Once the ISO file for Ubuntu is downloaded, settings for the VM must be open and the ISO attached file to the optical drive of the VM.

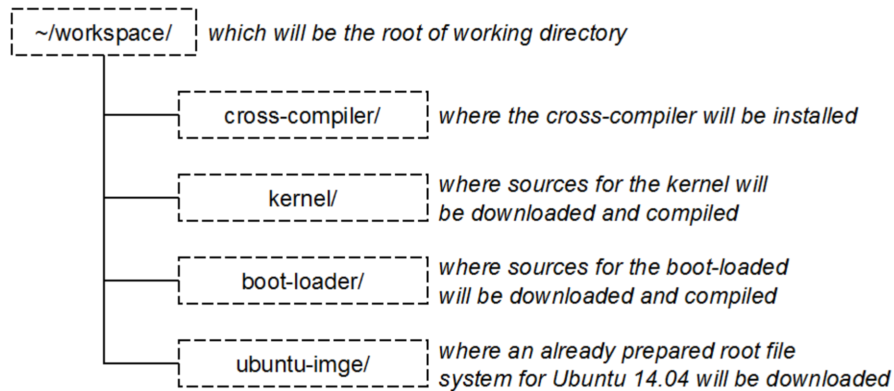


Next, start the VM and follow on-screen instructions to install Ubuntu 14.04 on the hard disk drive (HDD) of the VM. The process is quite straightforward, but in case of problems this webpage contains additional help to aid in the process:

https://help.ubuntu.com/community/LiveCD

2.3. Configuring VM workspace

One very important aspect of the development process is the working directory layout, so as it is recommended at the compilation guide in [1], all software will be downloaded inside a root directory called `workspace`, just to keep things together and ease backups. This root directory will be located inside the default user home directory, and following structure will be used:



2.3.1. Tuning VM

Once the working environment has been set up, it must be configured.

Right after booting the freshly installed Ubuntu system on the guest machine, an upgrade should be done to the system (this will upgrade system packages to the latest version found at the official repositories).

With this goal, check that that you have internet access pinging one of Google's DNS servers. Responses from the pings should be showed next:

```
$ ping 8.8.8.8
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_req=1 ttl=58 time=53.7 ms
64 bytes from 8.8.8.8: icmp_req=2 ttl=58 time=64.7 ms
64 bytes from 8.8.8.8: icmp_req=3 ttl=58 time=53.0 ms
```

If internet access is available, it's time to upgrade the system, so type following command on a terminal (which may take a while to execute depending on internet connection speed):

```
$ sudo su -c "apt-get update && apt-get -y upgrade"
```

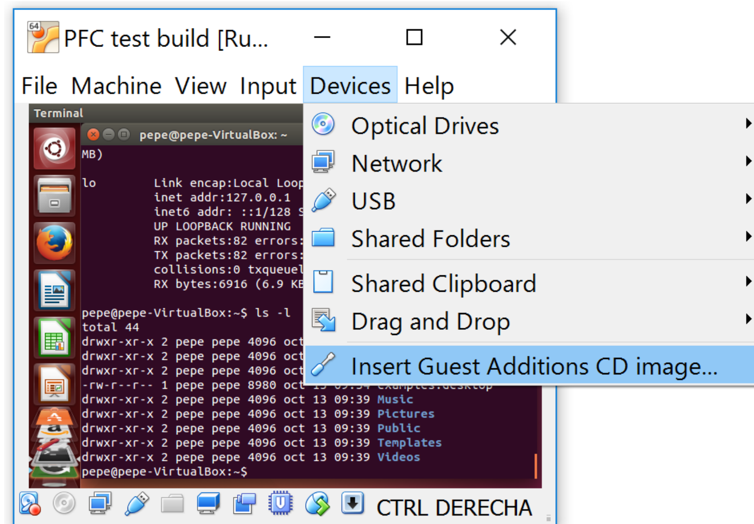
Having finished execution of the previous command and with system up-to-date, the VM is ready to be used with latest patches, but if following this guide to reproduce the steps it may be useful to enable things such as 3D Hardware acceleration, shared clipboard between the guest/host or enabling shared folders from the guest to the host.

So an additional utility must be installed, which comes bundled with Virtual Box (this utility is called *Guest Additions*).

Before installing *Guest Additions*, open a terminal and type following command (which will install some packages needed to properly install *Guest Additions*):

```
$ sudo apt-get install -y linux-headers-generic dkms
```

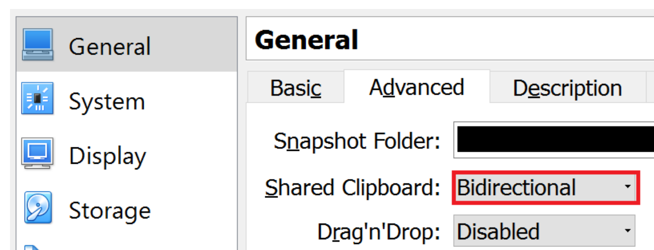
Now, at the running VM menu, click on the following option: **Devices -> Insert Guest Additions CD image…**



An ISO image will be automatically mounted at Ubuntu, which will then show a message asking for permission to automatically run the mounted CD. Click on **Cancel**, and type following commands:

```
$ cd /media/$USER/VBOXADDITIONS*
$ sudo ./VBoxLinuxAdditions.run
```

Once *Guest Additions* are installed shutdown the VM, open the VM settings and under the *General* options, enable *Bidirectional Shared Clipboard*:



Now start the VM again.

Note: If using a high density display screen, it may be useful to scale up display. To do this, go to **Configuration -> Display** and select a different value for *Scale for menu and title bars*.

2.3.2. Needed tools description

Although now the VM is up-to-date and ready to be used, some tools need to be installed to enable preparation of the Ubuntu/Linux image for the BeagleBone:

- git, which is a source code versioning tool. This is the tool used to get the source code for various components (in a process known as *cloning*) which, once compiled, must be assembled to produce the final bootable system image
- cross-compiler, which, as explained before, will transform the source code into executable binary files for the ARM platform
- complementary tools, which give support to the Kernel compilation process
- device tree compiler, which, as explained before, transforms the Device Tree Source (DTS) files into its binary representation (DTBO files)
- ssh tools, which in order for the development VM to be accessed remotely or to mount remote filesystems on the BeagleBone are needed

2.3.3. Install system utilities

Most of the needed software to be installed can be installed through Ubuntu package management system. This can be done executing the following command:

```
$ sudo apt-get install sshfs ssh git bc build-essential device-tree-compiler
fakeroot lsb-release lzma lzop man-db
```

If host machine, development VM and guest OS are both 64-bit, additional packages need to be installed to be able to execute the cross-compiler (which consist on 32-bit executables). The packages enable executing 32-bit executable files from a 64-bit machine. These additional packages can be installed with the following command:

```
$ sudo apt-get install libc6-i386 lib32stdc++6 lib32gcc1 lib32ncurses5 lib32z1
```

2.3.4. Install Device Tree Compiler

The Device Tree Compiler must be installed in a separate step. Commands needed to install it are:

```
$ cd /tmp
$ wget -c https://raw.github.com/RobertCNelson/tools/master/pkgs/dtc.sh
$ chmod a+x dtc.sh
$ sudo ./dtc.sh
```

Last command install DTC at system level, but at the end of the process it will leave a folder (~/git/dtc) with its source code. This folder is useless unless the DTC source code is to be studied or modified, so it’s safe to remove the folder with the following command (just to remove garbage from the system):

```
$ sudo rm -Rf ~/git/dtc
```

2.3.5. *Configure git command line tools*

To be able to build the Linux Kernel (whose build process implies downloading additional patches), username and email must be configured for *git command line tools* (installed on the last command executed).

Before this, a GitHub user account needs to be configured, which can be freely created at this website: https://github.com/ *(GitHub is a service which offers free hosting for public code repositories with an interface compatible with git tools)*

Once an user account is created, execute following commands, where:

- `{USERNAME}` is the user name for the GitHub account
- `{EMAIL}` is the email for the GitHub account

```
git config --global user.name "{USERNAME}"
git config --global user.email {EMAIL}
```

2.3.6. *Creating work directory structure*

Following commands will create structure described previously:

```
$ mkdir -p ~/workspace/ubuntu-image
$ mkdir -p ~/workspace/kernel
$ mkdir -p ~/workspace/cross-compiler
$ mkdir -p ~/workspace/boot-loader
```

2.3.7. *Download cross-compiler*

Note: same compiler version as the one used in [1] has been kept, to avoid compatibility problems, but there are newer compiler versions.

Following commands will install the cross-compiler using the previously mentioned folder structure:

```
$ cd ~/workspace/cross-compiler
$ wget -c https://releases.linaro.org/14.09/components/toolchain/binaries/gcc-linaro-arm-linux-gnueabihf-4.9-2014.09_linux.tar.xz
$ tar xf gcc-linaro-arm-linux-gnueabihf-4.9-2014.09_linux.tar.xz
$ rm gcc-linaro-arm-linux-gnueabihf-4.9-2014.09_linux.tar.xz
```

Note: as suggested in [1], now it’s possible to set an environment variable pointing at the C cross-compiler binary file:

```
$ export CC=~/workspace/cross-compiler/gcc-linaro-arm-linux-gnueabihf-4.9-2014.09_linux/bin/arm-linux-gnueabihf-
```

2.4. *Bootable image preparation: download and compile*

Once all tools have been installed into the VM, it's time to download ingredients needed to mount a bootable image.

Following sub-sections will provide enough detail to make the process repeatable.

2.4.1. Download and compile boot-loader

Following commands will download the bootloader source code using *git*, and apply needed patches to it in order to support the ARM processor used by the BeagleBone:

```
$ cd ~/workspace/boot-loader
$ git clone git://git.denx.de/u-boot.git
$ cd u-boot/
$ git checkout v2015.01 -b tmp
$ wget -c https://raw.githubusercontent.com/eewiki/u-boot-
patches/master/v2015.01/0001-am335x_evm-uEnv.txt-bootz-n-fixes.patch
$ patch -p1 < 0001-am335x_evm-uEnv.txt-bootz-n-fixes.patch
```

Then, the bootloader can be compiled with the following commands:

```
$ make ARCH=arm CROSS_COMPILE=${CC} distclean
$ make ARCH=arm CROSS_COMPILE=${CC} am335x_evm_defconfig
$ make ARCH=arm CROSS_COMPILE=${CC}
```

2.4.2. Download, configure and compile Kernel

Kernel source code partial download and preparation for compilation is done by the following commands:

```
$ cd ~/workspace/kernel
$ git clone https://github.com/RobertCNelson/bb-kernel.git
$ cd bb-kernel/
$ git checkout origin/am33x-v3.8 -b tmp
```

Once partial source code downloaded, following command will check for additional dependencies and download the rest of the source code:

```
$ ./build_kernel.sh
```

Note I: in there is some unmet system package dependency during execution of the previous command, a message will be shown, indicating how to proceed.

Note II: during execution of previous command while reproducing these steps, an error related to *git* and *gnutls_handshake* and occurred, which was solved by following the steps described in this article: https://forums.aws.amazon.com/message.jspa?messageID=539373

Download may take a while depending on Internet connection speed, so be patient.

Once previous command finishes downloading source code is downloaded and applying patches to it, a screen where kernel parameters can be tweaked will be shown:

```
.config - Linux/arm 3.8.13 Kernel Configuration
              Linux/arm 3.8.13 Kernel Configuration
  Arrow keys navigate the menu.  <Enter> selects submenus --->.
  Highlighted letters are hotkeys.  Pressing <Y> includes, <N>
  excludes, <M> modularizes features.  Press <Esc><Esc> to exit,
  <?> for Help, </> for Search.  Legend: [*] built-in  [ ] excluded

        General setup  --->
    [*] Enable loadable module support  --->
    [*] Enable the block layer  --->
        System Type  --->
        Bus support  --->
        Kernel Features  --->
        Boot options  --->
        CPU Power Management  --->
        Floating point emulation  --->
    └(+)

           <Select>    < Exit >    < Help >
```

Navigating through the menus, locate and enable following options:

- under **General Setup -> IRQ subsystem**, enable following option:

  - Expose hardware/virtual IRQ mapping via debugfs, this will make a file available in the */proc* virtual filesystem, with information about IRQ assignments

- under **Device Drivers -> DMA Engine Support**, make sure following options are enabled:
  - TI EDMA support, this will enable support the DMA controller included with the BeagleBone processor
  - Async_tx: Offload support for the async_tx api, this will enable asynchronous memory copy support at the kernel
  - DMA Engine debugging, this will enable showing debug messages from the DMA Engine through *dmesg* interface
  - DMA Engine verbose debugging, this will enable extended debug message info from the DMA Engine through *dmesg* interface
- under **Kernel Hacking**, make sure following options are enabled (which will both provide extended debug information through *dmesg* interface):
  - Enable dynamic printk() support
  - Enable debugging of DMA-API usage

Make sure that all previous options are set with a * tick, which means built-in support. Otherwise compilation may fail.

Also, make sure (from the menu system) that the following components are disabled:

- under **Device Drivers -> Misc devices**
  - Amazing GPIO DMA Event Test Driver(tm), trying to compile this driver made the build process stop with errors in a recent revision of this guide

Once done, exit and confirm to save changes. Compilation process will begin and may take a time to complete of about the order of an hour (depending on speed and number of processors assigned to the development machine).

Once compilation finishes, it will show kernel version (write down this version), on a line like this:

```
Script Complete
eewiki.net: [user@localhost:~$ export kernel_version=3.8.13-bone80]
```

2.4.3. Download prepared Ubuntu 14.04 image for ARM

Next, an already prepared Ubuntu 14.04 image for ARM will be downloaded.

Although there exist newer versions of the OS available at [3], this version has been chosen because it was the version selected at [1], has Long Term Support (it's an LTS release), and its size is smaller than that of 16.04.

Following commands download and decompress the file system:

```
$ cd ~/workspace/ubuntu-image
$ wget https://rcn-ee.net/rootfs/eewiki/minfs/ubuntu-14.04.4-minimal-armhf-
2016-04-02.tar.xz
$ tar xf ubuntu-14.04.4-minimal-armhf-2016-04-02.tar.xz
```

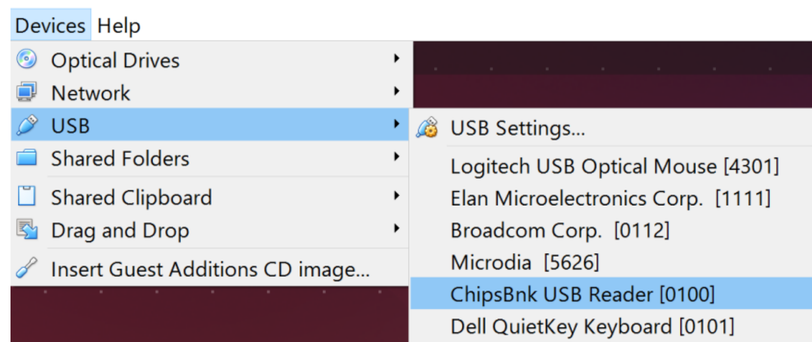
2.5. *Bootable image preparation: image assembly*

Now that the build process is done and all the pieces ready, the image assembly process can begin.

Start with setting an environment variable which stored the compiled Kernel version (obtained at the end of kernel compilation):

```
$ export kernel_version=3.8.13-bone80
```

The SD-card has to be inserted on the host PC. After this, go to the Virtual Box menu for the running development VM and attach the SD-card to the VM:



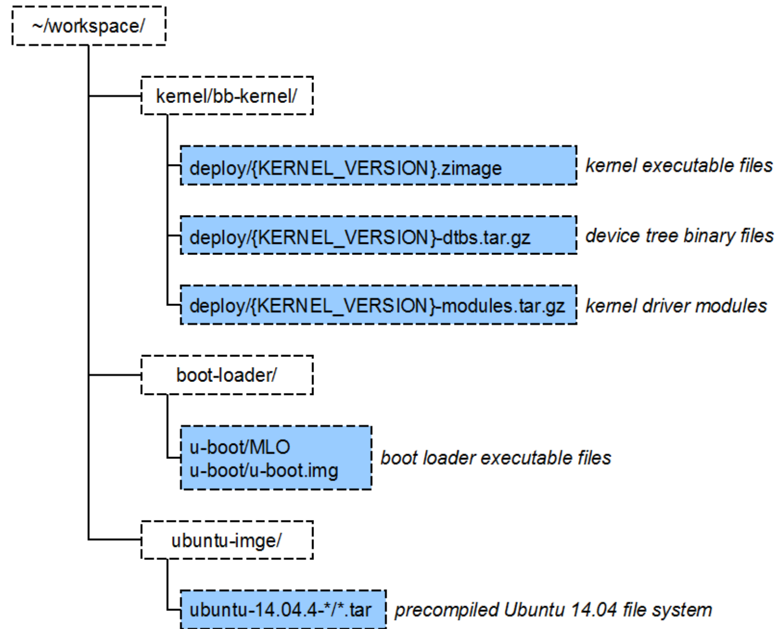
Under **Devices** -> **USB**, an option which resembles a card reader should appear. Once this is done, the SD-card will be attached to the VM, which can be confirmed executing the `lsblk` utility:

```
$ lsblk
NAME    MAJ:MIN RM   SIZE RO TYPE MOUNTPOINT
sda       8:0    0    25G  0 disk
├─sda1    8:1    0    23G  0 part /
├─sda2    8:2    0     1K  0 part
└─sda5    8:5    0     2G  0 part [SWAP]
sdb       8:16   1   3,7G  0 disk
├─sdb1    8:17   1  70,6M  0 part
└─sdb2    8:18   1   3,3G  0 part
sr0      11:0    1  55,6M  0 rom  /media/pepe/VBOXADDITIONS_5.0.22_108108
```

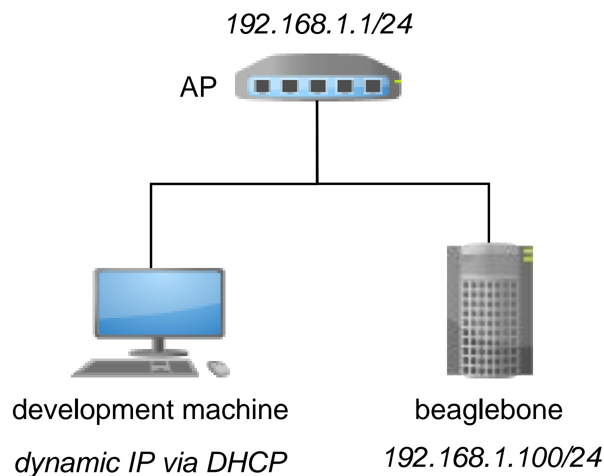
From previous command output, it’s known that the SD-card is located at `/dev/sdb`, so execute following command which will cretate an environment variable used during subsequent commands execution:

```
export DISK=/dev/sdb
```

Following diagram depicts components previously prepared which will be used during image preparation (blue boxes):



Later, when it's time to configure networking for the BeagleBone, it will be taken into account the logical network configuration which during preparation of this guide was the following:



Previous scheme takes into account that the DHCP range does not include 192.168.1.100 address, and will allow the BeagleBone connect to the Internet in order to upgrade packages.

2.5.1. Formatting, installing the bootloader and partitioning SD-card

Following commands will erase contents at the beginning of the SD-card:

```
$ sudo dd if=/dev/zero of=${DISK} bs=1M count=10
```

And following commands will install the bootloader into the SD-card.

```
$  sudo  dd  if=~/workspace/boot-loader/u-boot/MLO  of=${DISK}  count=1  seek=1
conv=notrunc bs=128k
$ sudo dd if=~/workspace/boot-loader/u-boot/u-boot.img of=${DISK} count=2 seek=1
conv=notrunc bs=384k
```

Next command will create partition table at the SD-card (command takes 3 lines):

```
$ sudo sfdisk --in-order --Linux --unit M ${DISK} <<-EOF
1,,0x83,*
EOF
```

Note: It is possible that (in order to execute previous command), all filesystems which use the SD-card device have to be unmounted first. Filesystems currently mounted can be shown using the `mount` command.

And finally, following command will create an EXT4 partition which uses all card space:

```
sudo mkfs.ext4 ${DISK}1 -L rootfs
```

2.5.2. *Mount SD-card and prepare image*

Once the SD-card is mounted, it's time to assemble all software components (prepared and compiled before) into it.

First step is mounting the SD-card filesystem on a local filesystem:

```
$ mkdir -p ~/workspace/rootfs
$ sudo mount ${DISK}1 ~/workspace/rootfs
```

Then, copy the Ubuntu image with following command:

```
$ sudo tar xvfp ~/workspace/ubuntu-image/ubuntu-14.04.4-*/*.tar -C
~/workspace/rootfs
```

Next, copy the Kernel image into the SD-card:

```
$ sudo cp -v ~/workspace/kernel/bb-kernel/deploy/${kernel_version}.zImage
~/workspace/rootfs/boot/vmlinuz-${kernel_version}
```

And the modules:

```
$ sudo tar xfv ~/workspace/kernel/bb-kernel/deploy/${kernel_version}-
modules.tar.gz -C ~/workspace/rootfs
```

2.5.3. *Image configuration*

Up this point, all software is already copied into the SD-card. Now it is needed to tweak it and set some basic configuration options:

Following command is used to set Kernel version information:

```
$ echo "uname_r=${kernel_version}" | sudo tee ~/workspace/rootfs/boot/uEnv.txt
```

Next, the network configuration must be set. This configuration was exposed before and will take into account the logical network topology during elaboration of this guide. Following command (take into account that it expands more than one line) does it:

```
$ cat | sudo tee ~/workspace/rootfs/etc/network/interfaces <<-EOF
```

```
# interfaces(5) file used by ifup(8) and ifdown(8)
# Include files from /etc/network/interfaces.d:
source-directory /etc/network/interfaces.d
auto lo
iface lo inet loopback
# primary network interface
auto eth0
iface eth0 inet static
address 192.168.1.100
netmask 255.255.255.0
network 192.168.1.0
gateway 192.168.1.1
EOF
```

Finally, the compiled Device Tree Binary Object should be copied into the BeagleBone. Refer to [1] for details on this.

Next flush pending buffers and unmount the SD-card with following commands:

```
$ sudo sync
$ sudo umount ~/workspace/rootfs
```

2.6. *Bootable image testing*

Now that everything is configured, SD-card can be inserted into the BeagleBone and the board powered on.

2.6.1. *Connection to BeagleBone*

If all previous steps went well, it will be possible to connect the board from development VM, using `ssh`.

User and password for the system image running at the Board can be obtained executing following command:

```
$ cat ~/workspace/ubuntu-image/ubuntu-*/user_password.list
```

Authentication parameters (if Ubuntu ARM system image has not changed) should be:

- username: **ubuntu**
- password: **temppwd**

Hence, it is possible to establish a shell session using `ssh` with following command:

```
$ ssh ubuntu@192.168.1.100
```

2.6.2. *Mount BeagleBone file system as a local folder*

Although (to send files to the BeagleBone) `scp` command can be used, it can be more interesting to be able to mount some folder from the Beagle at a local folder on development machine. To achieve this, `sshfs` utility will be used. This method has been chosen because no *root* privileges on the development machine are required to do it.

Executing `sshfs` is simple, and prior to executing the command the mount point must be created. Following commands do that:

```
$ mkdir -p ~/workspace/beagle
$ sshfs ubuntu@192.168.1.100:/home/ubuntu ~/workspace/beagle
```

Whenever the remote folder has to be unmounted, it will be seen that executing `umount` requires root privileges at the development VM.

However, there exists a command which doesn’t require root privileges and can be used for unmounting the remote folder (mounted via `sshfs`). To do it, execute:

```
$ fusermount -u ~/workspace/beagle
```

[1] R. González Muñoz. " Improving the BeagleBone board with embedded Ubuntu, enhanced GPMC driver and Python for communication and graphical prototypes". M.S. thesis, Electrical Engineering department, Technical University of Catalonia (UPC), Barcelona, Spain, 2015

3. Kernel Programming Techniques

Following sections contains the programs and guides whose development is the core of this work.

Programs are presented in difficulty order, starting with most basic topics and in increasing difficulty level (sometimes understanding how a program works is a pre-requisite before moving to the next one).

3.1. Guides organization and common treats

An important fact on how the guides for the for the different topics areas are presented is that they're not exposed on a *this is done that way* fashion.

Instead, guides are structured in a mode which resembles the way information for this work was researched, and almost all of them follow the same structure:

- Introduction: main guide topic is introduced, exposing the motivations for using the technique which supports the topic, sometimes with selected examples which illustrate common use-cases
- Landing into Linux: i.e. how the technique meets Linux, sometimes functionally while others explaining the API offered by the kernel supports the explained concept, highlighting which are the main points which need to be considered when implementing it
- Previous reader work: some simple exercises (which don't yet require programming) are proposed, which aim to get the reader familiar with the concepts and do some further documentation research
- Technique details: the technique explanation is fully developed, be it with explanation or with example (and commented) code
- In-progress reader work: exercises for the reader are proposed, which are increasingly difficult and make use of the concepts illustrated during the *technique details* part. They're sometimes interleaved with the previous part development

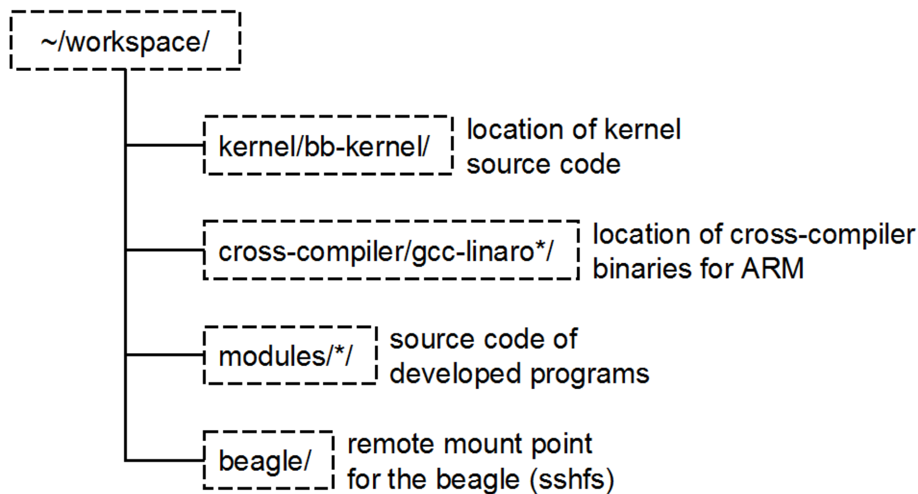
To not bloat the guides with too much information, and not to reveal solutions to the proposed exercises to the readers willing to learn by themselves, a section is later included in this work which covers details on:

- extended comments about the guides themselves
- commented solutions to proposed exercises
- when suitable, screenshots and/or photos illustrating proper behavior of the solutions

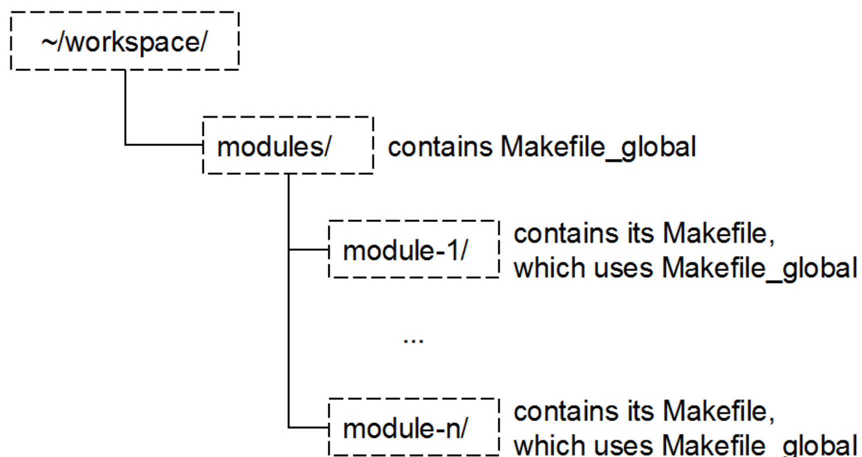
The guides are intended to be taken in same order as they're presented on this work, because some knowledge offered during the first guides is supposed to be acquired on later guides.

3.1.1. Assumed directory structure

During all this chapter, and to compile and deploy featured programs into the BeagleBone, it will be assumed that the following directory structure is in use at the development machine:



During development of the example drivers, following directory structure has been used.



3.1.2. Boilerplate code for all developed drivers

All drivers developed and showed during the guides contain code boilerplate code (code which is repeated along all modules), related to the driver (un)load process and some of its default configuration.

As this code is repeated over and over it will not be explained at each guide, but here:

- driver (unload), which is done through following code near the end of provided code:

```
module_init (XXX);
module_exit (YYY);
```

  The reason for this code is that it is the part of the standardized user-tweakable driver (un)load process, where:

  - the load is delegated to the driver entry point (XXX function). Generally, here go all driver-initialization related tasks.
  - the unload is delegated to the driver exit point (YYY function). Generally, all resources that have been allocated at driver entry point are released at the driver exit point.

- include directives, needed by the (un)load process are also always the same, and are composed of the following C statements at the beginning of every driver example:

```
// Needed by the driver itself
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
```

- information for the driver, which in this case is composed of license and version number, is some code which is always included at the end of the provided code:

```
MODULE_LICENSE ("GPL");
MODULE_VERSION ("1.0");
```

3.1.3. All global variables declared static

Note that generally every driver provided as example on this work declares all its global variables with the `static` modifier. This makes the variables local to that compilation unit (that C file), and avoids collision during *linkage* stage of the compilation process. It makes sense because no other part of the kernel space (nor other drivers) should have access to the driver variables (unless of course we want this to happen, e.g. by invoking kernel functions providing pointers to the local variables).

3.1.4. Trace mechanism

Example drivers make extensive use of the `printk` function, as it is a *cheap* way of validating proper behavior of the drivers which doesn't require a complex setup.

This way, using `dmesg` is enough to get information about the example drivers' inner working.

Unfortunately, the Kernel has been compiled with a high verbosity level (specifically for DMA-related functions), and this produces the `dmesg` output to get flooded with debug messages from the Kernel.

For this reason, on the provided examples a macro is defined, of this kind:

```
#define MODULE_TRACE KERN_ALERT "example-driver-1: "
```

And every time printk is invoked from the driver, it is invoked this way:

```
printk (MODULE_TRACE "initializing\n");
```

Thanks to this, the text (this is only an example) “example-driver-1” is prefixed to every *printk-ed* message.

Then, from the console it’s possible to execute following command to only show messages produced by the driver under study:

```
$ dmesg | grep "example-driver-1"
```

3.2. GPIO And Interrupts Guide

3.3.1. Interrupts

Interrupts are used as a mechanism used to notify the processor of an exceptional situation which requires immediate attention.

When an interrupt occurs, the processor temporarily suspends the code being executed (hence the name interrupt) and invokes the associated interrupt handler (or ISR), which contains the code needed to manage the exceptional exception. Once the ISR code ends, the processor resumes the execution of the code it was previously executing.

Interrupts can be originated by software, because of some code execution. For example:

- *when some code tries to write to a non-mapped memory address, the MMU generates an exception. In this case, usually the behaviour of the OS is to terminate the context of the offending code (the process) and optionally generate a core dump file.*
- *when some code tries to execute an invalid opcode (a binary code which the processor does not know how to implement). This feature is used for example for floating point unit emulation.:*
    - *When the processor encounters an invalid opcode, it invokes the associated ISR*
    - *Back to the OS, the associated ISR may contain a floating point unit emulator, which will decode the invalid opcode and, if it is a floating point unit opcode, simulate its execution entirely on software*
    - *Once finished with the execution, the ISR terminates without error and the program continues execution*

  Obviously, this kind of mechanism behaves several orders of magnitude slower than a real floating point unit, but may allow execution of code in simpler processor for which the code was originally designed.
- *when a process tries to access to a memory address which is physically stored inside virtual memory (for example the hard disk). Is this case the OS takes care of this situation and puts again the swapped memory into the RAM, making it available to the process.*

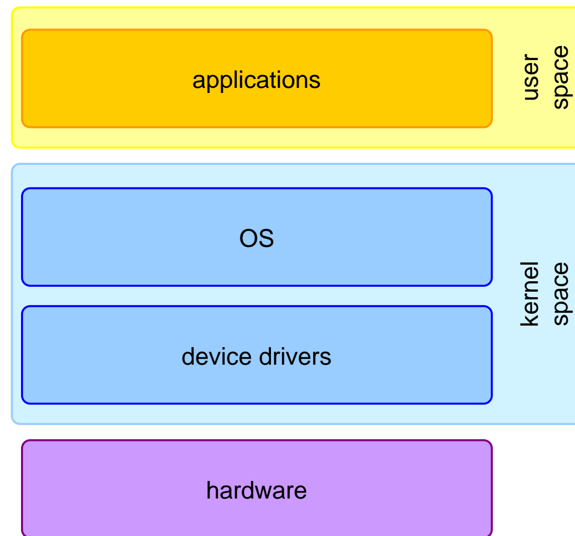
Interrupts can also be originated by hardware, as a need for some peripheral device to notify some situation. For example:

- *when a key is pressed on the keyboard, an interrupt is generated, then the ISR in charge may read the newly pressed key's code and makes it available to the OS*
- *when some data is received through a network adapter, an interrupt is generated, then the ISR in charge may read the received data and make it available to the OS*
- *the system clock may generate an interrupt at each clock tick, then the ISR in charge may read the current timestamp and inform the OS, which in turn can trigger the appropriate events*

3.3.2. *Who provides each ISR code?*

The OS usually provides the needed device drivers for some standard or highly extended devices like common sound card brands, Ethernet adapters, video cards, disk controllers, keyboards, mouses, etc…

These device drivers provide the OS with the needed functions to interact with the devices' hardware:



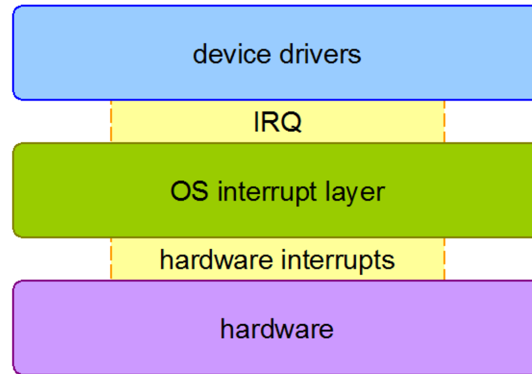
It is important to remark that the OS code and the device drivers' code executes in kernel space: *they have access to all hardware resources and, if not programmed well, they may lead to application (or even system) hangs and odd behaviours*.

3.3.3. *Basic interrupt management in Linux*

Most modern processors do have an interrupt handler table, which is a memory region which contains one address for every possible interrupt. Each one of these addresses is the memory address for the ISR associated to a particular interrupt.

From Linux kernel point of view (and in kernel source code), an *interrupt* is always referred to as an *IRQ* (for **I**nterrupt **R**eq**Q**uest).

In Linux, when drivers wish to install an interrupt handler (an ISR), they do not alter directly the interrupt table. What really happens is that the kernel sits in the middle of the hardware interrupt and the device driver:



Due to this extra layer, the kernel can provide extra functionality around interrupts, like:

- accounting the number of times an interrupt has been signaled
- knowing which driver has requested to be notified for every interrupt
- allowing multiple drivers to register a *callback* for a single interrupt

Hence, from a global point of view, interrupt handler table addresses always point to the basic interrupt management subsystem, never to a device driver ISR or *callback* function.

When the processor (or some device) needs to signal an interrupt, the ISR from the interrupt management subsystem is invoked. Then, this subsystem invokes the necessary driver-registered callbacks.

3.3.4. Basic API for interrupt management from drivers

The linux kernel provides two basic functions for processes to be able to register a *callback* when an interrupt happens, which are declared in the following include file:

```
linux/interrupt.h
```

Functions are:

```
int request_irq(unsigned int irq,
                irq_handler_t handler,
                unsigned long flags,
                const char *name,
                void *dev);
```

This function registers a *callback* function for an IRQ. Usually, drivers register IRQ *callbacks* during its initialisation.
Return value for this function is 0 on success and <0 in case of some error. Its arguments are:

- **`irq`**: is the IRQ number (see below for comments on this)
- **`handler`**: is the pointer to the *callback* function which will be invoked when the interrupt is dispatched. The signature of this parameter is the following:

```
typedef irqreturn_t (*irq_handler_t)(int irq, void * dev);
```

Where:

- **`irq`**: is the IRQ number (the same used to invoke `request_irq`)
- **`dev`**: is the same `dev` argument used to invoke `request_irq`

Return value for this function should always be `IRQ_HANDLED`.

- **`flags`**: define additional behaviour related to the requested IRQ's processing.

  Only the flags relevant for the exercises are presented here. This argument can be a combination of any of the following flags:

  - **`IRQF_TRIGGER_RISING`**: the *callback* function will be invoked on the rising flank of the signal wired to the interrupt line.
  - **`IRQF_TRIGGER_FALLING`**: the *callback* function will be invoked on the falling flank of the signal wired to the interrupt line.
  - **`IRQF_SHARED`**: this IRQ is shared among one or more device driver. This is, more than one *callback* function may be registered for the same IRQ. If this flag is not set, trying to register more than one *callback* function for the same IRQ will fail after the first registration.

- **`irq`**: is the IRQ number (see below for comments on this).
- **`name`**: can be an arbitrary null-terminated string, but it is strongly recommended to use the driver name for this parameter (this will ease further diagnostics if things don't work).
- **`dev`**: if the IRQ is not shared (this is, there is just a single registered *callback* for this IRQ in the system), this parameter can be NULL. Otherwise, this parameter must be some valid pointer at the driver memory. If the same driver tries to register more than one *callback* for the same IRQ, each time the `request_irq` function is invoked the `dev` parameter must be different.

```
void free_irq(unsigned int irq, void * dev);
```

This function unregisters a *callback* function for an IRQ. Usually drivers unregister *callbacks* during its deinitialisation.

`irq` and `dev` arguments must the same as the used to invoke the corresponding `request_irq` function.

This function has no return value.

It is very important that each `request_irq` during driver initialisation has a matching `free_irq` during deinitialisation. Otherwise, system may become unstable.

3.3.5. Notes on IRQ management from a driver

There are some aspects which have to be taken into account when writing a *callback* function for an IRQ (this is, an ISR) in Linux:

- **asynchronous behaviour**: As the code of the ISR may be executed as a result of an interruption of code in other parts of the driver, special care has to be taken when the ISR code accessed to data structures. If the code is bad written, the following situation would be possible:
  - 1st, the driver is working with some data structure (for example, a linked list)
  - 2nd, during the data structure usage of previous point, an IRQ is generated, and it happens that the ISR code manipulates the same data structure. As the interrupt code may interrupt the normal flux of the rest of the driver code, it happens that the ISR code is working a data structure which has a manipulation in course.

  Very probably, the data structure will become corrupted if not designed for such a situation. In this case, good re-entrancy practices (only use stack (but not static) variables, no shared data manipulation; if strictly necessary to access shared data, do it in an atomic way, etc…) may help avoid some data corruptions or race conditions.

- **blocking calls**: it may be thought that the previous *asynchronous behaviour problem* could be solved by means of some blocking mechanism, like kernel-semaphores or wait events.

  Suppose this scenario:
  - 1st, the driver code decrements a semaphore to gain access to an exclusive resource or data structure.
  - 2nd, whilst the driver has gained access to the semaphore, an IRQ is generated and the associated ISR tries to gain access to the resource. For this reason, the ISR also decrements the same semaphore.

  On the previous scenario:
  - the driver code holds the semaphore, but cannot continue execution until the ISR code ends.
  - the ISR code is blocked on semaphore decrement, but it has to wait for the driver code to increment it.

  The previous situation is a deadlock situation.

- **type of simple variables**: if using simple variables inside an ISR code (for example an *int* typed variable), they should be declared as **volatile**.

  Otherwise, other driver code which uses the variables but don't modify them may be *optimized* by the compiler, who is smart enough to know that the other driver code does not modify the variable. This way, the compiler may optimize the code assuming that the variable's value does not change. Sadly, the compiler is not smart enough to figure that the variable can be modified from some asynchronous code, like the ISR.

- **fast response**: the processor may generate thousands of interrupt requests per second. So the code of an ISR has to be **fast**. For debugging purposes it's fine to use some ***printk***

statements (although invocation to ***printk*** is slow) inside the ISR code, but be careful not to spend much time inside the ISR, because the system may get blocked.

3.3.6. Mapping between interrupt number and IRQ number

As the Linux kernel sits in the middle between the interrupt handler and the ISR code from drivers, it may alter in the way the interrupts are processed.

One of these alterations is that usually, for legacy reasons, the IRQ number does not correspond to the interrupt number. On modern x86/x64 computers, for historical reasons, there may be an offset between interrupt number and IRQ number. In the past this represented a problem for device drivers, which could need to guess the IRQ numbers by means of probing.

Fortunately, for the practices, it will not be needed to guess IRQ numbers.

Linux kernel can be compiled with a debug option which makes available a file on the *debug filesystem* which shows the mapping between Linux IRQ number and hardware IRQ. On the Beaglebone Board used in the lab, the kernel is compiled with such option.

LAB: Exercise 1 - IRQ mapping file

Look at the contents of the **`/sys/kernel/debug/irq_domain_mapping`**:

- what does it contain?
- do you observe any mapping that is different than the others?
- search in the AMM335X processor datasheet the usage for the interrupts 0x12 and 0x13

3.3.7. GIPO usage from drivers

In the kernel source code directory, there is a document named **Documentation/gpio.txt** which describes the API drivers can use to access GIPO functionality.

As it will be seen, most of the functions available for drivers are also available manipulating the files found at the **`sysfs`** directory.

The most important functions are:

```
int gpio_export(unsigned gpio, bool direction_may_change)
```

Makes it possible to work with the GPIO from the **`sysfs`** directory. Its parameters are:

- **`gpio`**: the GPIO number.
- **`direction_may_change`**: whether this GPIO direction may be changed on the fly.

```
void gpio_unexport(unsigned gpio)
```

Removes the associated GPIO files from the **`sysfs`** directory, and thus not allowing to work with the GPIO from the file system. Its parameters are:

- **`gpio`**: the GPIO number.

```
int gpio_request(unsigned gpio, const char *label)
```

Allocates memory for a GPIO and reserves it for exclusive use by a driver. Usually, this function is invoked during the initialization of the driver. Its parameters are:

- **`gpio`**: the GPIO number.
- **`label`**: a null-terminated string indicating who requested the GPIO. Usually this parameter is the driver name.

A GPIO cannot be requested if has previously been requested and not freed. Return value for this function is 0 on success or <0 in case of error.

As a good practice, a driver should always call this function before using a GPIO.

```
void gpio_free(unsigned gpio)
```

This function indicates the kernel that the driver is not going to use the GPIO anymore. Usually, this function is invoked during the deinitialisation of the driver. Its parameter is:

- **`gpio`**: the GPIO number.

```
int gpio_direction_input(unsigned gpio)
```

Sets the direction of a GPIO to input. Its parameter is:

- **`gpio`**: the GPIO number.

Returns 0 on success and <0 in case of error.

```
int gpio_direction_output(unsigned gpio, int value)
```

Sets the direction of a GPIO to output, setting its value. Its parameters are:

- **`gpio`**: the GPIO number.
- **`value`**: the value to assign to the GPIO. May be **0** or **1**.

Returns 0 on success and <0 in case of error.

```
int gpio_get_value(unsigned gpio)
```

Returns the value of an input GPIO. Its parameters are:

- **`gpio`**: the GPIO number.

Returns **0** or **1** depending on the value of the GPIO.

```
void gpio_set_value(unsigned gpio, int value)
```

Sets the value of an output GPIO. Its parameters are:

- **`gpio`**: the GPIO number.
- **`value`**: the value to assign to the GPIO. May be **0** or **1**.

LAB: Exercise 2 - Documentation

In the kernel source code directory, look inside the **Documentation/gpio.txt** file.

- what is the purpose of the `gpio_to_irq` function?

A simple code to turn on a LED from the Beaglebone when the driver module is loaded and turn it off when the module is unloaded may be:

```
/* in initialization function of the driver */
if (gpio_request(48, "module.ko") < 0))
    printk("error requesting GPIO 48\n");
else
    gpio_direction_output (48, 1);

/* in deinitialization function of the driver */
gpio_set_value (48, 0);
gpio_free (48);
```

LAB: Exercise 3 - Driving LEDS from the driver

Modify the driver so that:

- during initialization, it turns on the four LEDS.
- during deinitialisation, it swaps the current value of the four LEDS

Take into account that the following situation should be managed well:

- load the driver module, so all LEDS will be turned on
- using **`echo`** command from the **`sysfs`** directory, turn off LEDS 2 and 4
- unload the driver module, so LEDS 1 & 3 should turn off and LEDS 2 & 4 should turn on

3.3.8. Event aware drivers

Drivers are normally used in order communicate user programs with peripheral devices. In some of these cases, the execution flow of the code of the driver depends on events triggered by the peripheral device:

- for an ADC converter: the converter may generate an event when the conversion finishes.
- for a frame grabber: frame grabber may generate an event when there is a picture available.
- for an Ethernet card: the card may generate an event when there is data ready to be read.

In the previous case, when the driver needs to act as a consequence of an event generated by the external device, two strategies may be used:

- **polling mechanism**: using this mechanism, when there is a need to receive data, the driver keeps polling the external device until some condition asserts.

  A simple pseudocode illustrating this mechanism may be the following:

  ```
  function wait until event happens
  ```

```
{
    ask the external device if an event has occurred

    while the event has not occurred
    {
        sleep some amount of time
        ask again to external device if an event has occurred
    }

    // arrived at this point, the event has occurred
}

main block of code
{
    wait until event happens

    execute some code that depends on event occurred
}
```

The problem with this strategy is that the driver continuously keeps *asking* the device if there is data ready to be read, and this implies:

- **resource consumption** from the driver
    - CPU usage: because the process of asking if the event occurred uses CPU
    - I/O usage: because the process of asking if the event occurred uses I/O resources
- **blocking**
    - because the driver must keep waiting until the event happens at the external device. Meanwhile, the driver keeps blocked waiting until the event occurs

- **interrupt mechanism**: using this strategy, the driver subscribes itself to some event, and when the event occurs, some code is executed.

A simple pseudocode illustrating this mechanism may be the following:

```
function to execute when event happens
{
    execute some code that depends on event occurred
}

main block of code
{
    subscribe "function to execute when event happens" to the event
}
```

Previous code implies that the driver tells the processor that the later should execute the subscribed code when the event occurs. After this, the driver code may do another work (not related to event polling) or even return. The processor will take care of the rest.

Although not all use cases are suitable for an interrupt mechanism, the main advantages with respect to the polling mechanism related to acting on event detection are:

- **resource consumption** decreases. As the driver is not continuously asking the external device, there is no constant I/O resource occupation. For the same reason, as the driver does not spend CPU cycles with the continuous polling.
- **blocking** may be avoided. As the driver subscribed itself with the processor to the event, the processor will be in charge of notifying the driver when the event happens, so the driver no more needs a blocking loop in order to be aware that the external event happened.

All these advantages come at a cost, which also has to be taken into account. For example, the subscribed code will be executed in an **asynchronous** way, in the same manner that a signal handler is executed in an asynchronous way related to the rest of the code of the program which receives the signal. This implies that the subscribed code must take care of issues (explained before in this document) like re-entrancy (related to data integrity), low latency, types of variables, possible deadlocks, etc...

An extreme scenario is when a driver code does all its work triggered by interrupts. Such a system is called an interrupt-driven system.

3.3.9. GPIO and interrupts

When operating with GPIO signals from the driver code it is possible to instruct the OS that, when a GPIO changes its value, an interrupt is generated.

Following code registers an ISR when a GPIO changes its value from 0 to 1 (for example, when the push button on the Beaglebone is pressed). Behaviour of the ISR is to print a trace which may be seen executing the `dmesg` utility or doing a grep on `/var/log/syslog` file:

```
// the ISR code, executed when the push button is pressed
irqreturn_t gpio_irq_handler(int irq, void * dev)
{
    printk("GPIO: gpio_irq_handler triggered\n");
    return IRQ_HANDLED;
}

int register_ISR_for_GPIO (int gpio, irq_handler_t isr)
{
    int error, irqnumber;

    error = gpio_request(gpio, "module.ko");
    if (error < 0)
    {
        printk (“ERROR %d: could not request GPIO %d\n”, error, gpio);
        return error;
    }
    printk (“GPIO %d: requested successfully\n”, gpio);

    error = gpio_direction_input (gpio);
    if (error < 0)
    {
        printk (“ERROR %d: could set input direction for GPIO %d\n”, error,
gpio);
        return error;
    }
    printk (“GPIO %d: input direction set successfully\n”, gpio);

    // TODO: determine the irq number for the GPIO
```

```
    irqnumber = ...

    printk (“GPIO %d: corresponds to IRQ number %d\n”, gpio, irqnumber);

    error = request_irq(irqnumber, isr, IRQF_TRIGGER_RISING, "module.ko", NULL);
    if (error < 0)
    {
        printk (“ERROR %d: could not request IRQ %d for GPIO %d”, irqnumber,
gpio);
        return error;
    }
    printk (“GPIO %d: IRQ %d requested sucessfully\n”, gpio, irqnumber);

    return 0;
}

void unregister_ISR_for_GPIO (int gpio)
{
    int irqnumber;

    // TODO: determine the irq number for the GPIO
    irqnumber = ...

    free_irq (irqnumber, NULL);
    printk (“GPIO %d: IRQ %d freed\n”, gpio, irqnumber);

    free_gpio (gpio);
    printk (“GPIO %d: freed\n”, gpio);
}
```

Previous code can be used in the following way:

- during driver initialization, use the following code:

```
register_ISR_for_GPIO (20, gpio_irq_handler)
```

- during driver deinitialisation, use the following code:

```
unregister_ISR_for_GPIO (20)
```

LAB: Exercise 4 – Completing the GPIO and interrupt code

a) Complete, in the previous code, the parts marked with TODO:

```
// TODO: determine the irq number for the GPIO
irqnumber = ...
```

b) Compile the driver code, and copy the binary into the Beaglebone

c) In the Beaglebone, execute the following command:

```
tail -f /var/log/syslog | grep GPIO
```

d) Now load the driver:

- What do you observe in the console?

- Which IRQ corresponds to GPIO 20?

f) Now press the push button (which corresponds to GPIO 20) in the Beaglebone. What do you observe in the console?

LAB: Exercise 5 - Details on interrupts

With the driver loaded, look at the contents of **`/sys/kernel/debug/irq_domain_mapping`**:

a) Search for the IRQ that corresponds the **Exercise 4.d)**

b) What do you observe related to that IRQ?

c) Which hardware IRQ corresponds to the GPIO? Is this number related in some way with the GPIO?

Now, you can unload the driver.

LAB: Exercise 6 - Some modifications on code

Modify the previous code in such a way that when the push button is pressed, the four LEDS are alternatively turned on and off.

It is possible to have more than one ISR registered for the same IRQ. To accomplish this, special care has to be taken when invoking `request_irq` and `free_irq` functions.

Search in the documentation (above in in this document) of the mentioned functions for the needed parameters to accomplish this.

LAB: Exercise 7 – Multiple ISRs for the same irq

Modify the previous code in such a way that:

- an ISR is registered for when the GPIO 20 value goes from 0 to 1. This ISR will turn of all the LEDS.

- an ISR is registered for when the GPIO 20 value goes from 1 to 0. The ISR will turn off all the LEDS.

For this exercise, it is mandatory to invoke `request_irq` twice, although the registered ISR may be the same.

3.3. File Drivers Guide

3.4.1. Introduction

In Linux, almost everything is treated as a file and programmers may operate with those files obtaining a file descriptor to them and using the appropriate access functions. For example:

- files found in `/proc` directory are special files which contain information about the system and its running processes. For example, `/proc/cpuinfo` and `/proc/meminfo` contain information about the system processors and system memory, respectively.

- files found in `/dev` directory correspond to devices, and allow interaction with all the devices present on the system. For example, `/dev/sda` and `/dev/sdb` are special files used by utilities like `mount` to allow interaction with the storage disks found on the system (like primary HDD and USB drives).

- when a process opens an Internet socket to a remote process, a file descriptor is obtained, allowing the process to send/receive data through it.

- processes always have file descriptors 0, 1 and 2 available, which correspond to *console input*, *console output* and *console error output*.

- documents found in the user home directory can also be accessed obtaining a file descriptor to them, using function `open`.

It is important to remark the following points:

- although files are a widely used abstraction in Linux, not all files do have a representation on the filesystem. For example: when creating a socket or a pipe it is possible to obtain a file descriptor which does not correspond to a file in the file system, and this means that only the process which obtained these file descriptors has access to them.

- in some other cases, when obtaining a file descriptor, this file descriptor is backed by a file on the filesystem. For example: the file `/proc/cpuinfo` or a PDF file found in the user home directory.

- Among the files found on the file system, not all of them correspond to physical files (which are stored on a HDD or a USB drive). For example: `/home/user/document.pdf` usually corresponds to a physical file, but `/proc/cpuinfo` does not correspond to a physical file.

Files on the file system which do not correspond to a physical file are always managed by drivers.

This means that when `cat /proc/cpuinfo` is executed, there is a driver which is in charge of returning the file contents, fulfilling it with system processors.

Depending on the type of file, different approaches do exist in order to manage access to the file, and one simple kind of driver that manages access to non-physical files is a *miscellaneous driver*.

3.4.2. Miscellaneous driver overview

One way to program a file driver for Linux is to implement a *miscellaneous driver*. This kind of driver creates a special file on the file system, in such a way that when process access it, the ultimate code which manages that access is the driver's code.

In a miscellaneous driver, functions which manage following access types can be implemented:

- `open`: which manages the case when the application wants to obtain a file descriptor for the special file

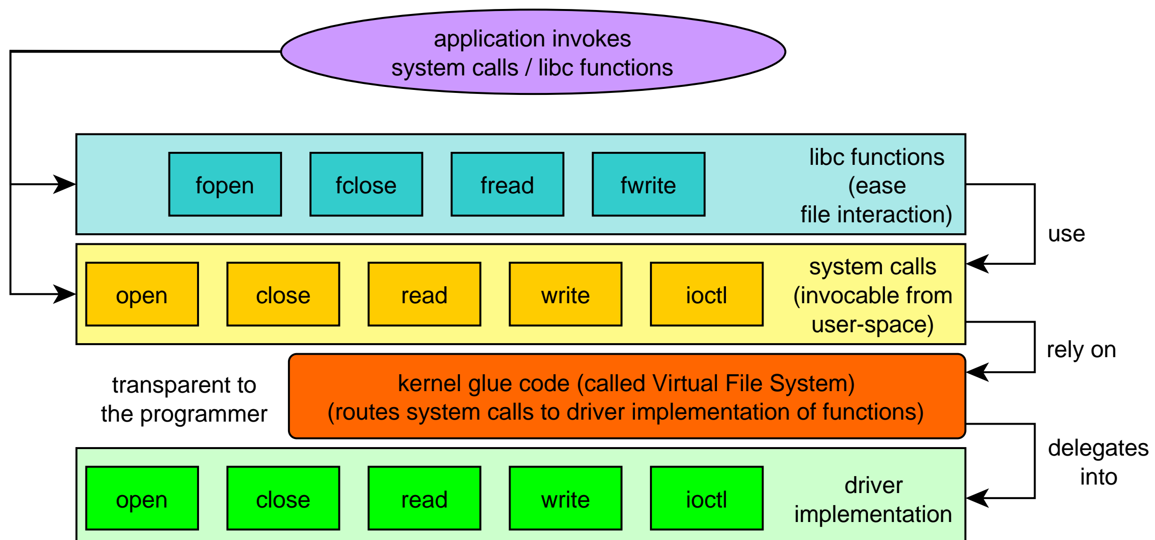
- `close`: which manages the case when the application wants to release a previously obtained file descriptor for the special file

- `write`: which is in charge of managing the write access to the special file (this is, when the application wants to put some bytes into it)

- `read`: which is in charge of managing the read access to the special file (this is, when the application wants to obtain some bytes from it)

- `ioctl`: which is in charge of managing the control commands sent from the application to the driver (`ioctl` stands for ***I****nput/****O****utput* ***C****ont****r****o****l***).

A simplified diagram of the components which come into play in a miscellaneous driver is depicted here:



Each layer in the above diagram has a specific purpose. Some illustrative examples are:

- **libc** functions simplify access to files found on the file system, providing additional capabilities such as buffering, opening files in text or binary mode and convenience functions like `fprintf`, etc... The basic data type used when dealing with **libc**'s file related functions is the `FILE` structure (or `FILE*` pointers). Although **libc** functions are very useful, they are limited to be used with files found on the file system, and they may not be used for example with Internet sockets.

- **system calls** represent a lower abstraction level, but may be used with any file supported by drivers (even when that file does not exist on the file system). For example, functions in this layer can be used to access console file descriptors for console (*stdin* / *stdout* / *stderr*), *sockets*, *pipes*, etc…

- **kernel glue code** sits in the middle between **system calls** and the **driver which implements access** to the accessed file. For example, when writing into a *socket*, it is the glue code which knows that the file descriptor corresponds to a *socket* and invokes the code from the corresponding driver.

- and finally, the **driver implementation** contains the code which performs the actual access. In the case of an Ethernet controller, the driver is in charge of communicating with the peripheral, with device-specific functions (interrupts, I/O memory mapped regions, I/O ports, etc..).

Related to files which allow interaction with devices:

- usually, only the code at the driver level contains device-specific code.

- the rest of the layers (specially **libc** functions and **system calls**) are the result of years of use from the programmer community and aim to provide standardized functions which not depend on the underlying specific device or driver.

- in some cases, however, applications need to do some specific tweak on the driver (or device). In such cases, it is common to interact with the drivers through the functionality provided by **`ioctl`**.

Signature (return type and input arguments) of access functions inside a miscellaneous driver is the following:

- file descriptor open function:

```
int open (struct inode* pInode, struct file* pFile)
```

- file descriptor release function:

```
int release (struct inode* pInode, struct file* pFile)
```

- ioctl function:

```
long ioctl (struct file* pFile,
            unsigned int cmd,
            unsigned long value)
```

- read bytes function:

```
ssize_t read (struct file *filp,
```

```
                char __user *buffer,
                size_t count,
                loff_t *f_pos)
```

- write bytes function:

```
ssize_t write (struct file *filp,
               const char __user *buffer,
               size_t count,
               loff_t *f_pos)
```

Exercise 1 (HOME): about API functions

Search on the book "*Linux device drivers*", on the chapter dedicated to *"char drivers*", the meaning of the arguments and the return value for each of the previous functions from the `struct file_operations`:

- *open*
- *release*
- *ioctl*
- *read*
- *write*

3.4.3. Simple implementation of a miscellaneous driver

To use a miscellaneous driver, some (de)initialization functions must be invoked when (de)initializing the driver.

As `open`, `release`, `ioctl`, `read` and `write` correspond to already existing *system call* function names, prefix `sd_` is used in the following example, which implements a very simple miscellaneous driver (important functions are highlighted in blue, while the rest is initialization code):

```
// Access functions: declaration
static int sd_open (struct inode* pInode, struct file* pFile);
static int sd_release (struct inode* pInode, struct file* pFile);
static ssize_t sd_write (struct file *filp,
                         const char __user *buffer,
                         size_t count,
                         loff_t *f_pos);
static ssize_t sd_read (struct file *filp,
                        char __user *buffer,
                        size_t count,
                        loff_t *f_pos);
static long sd_ioctl (struct file* pFile,
                      unsigned int cmd,
                      unsigned long pointer);

// Misc driver structures: declaration and initialization
static struct file_operations sd_fops =
{
    write:          sd_write,
    read:           sd_read,
    open:           sd_open,
    release:        sd_release,
    unlocked_ioctl: sd_ioctl
```

```
};
static struct miscdevice sd_devs =
{
    minor:   MISC_DYNAMIC_MINOR,
    name:    "driver_port",
    fops:    &sd_fops
};

// Access functions: definition
static int sd_open (struct inode* pInode, struct file* pFile)
{
      return 0;
}

static int sd_release (struct inode* pInode, struct file* pFile)
{

      return 0;
}

static long sd_ioctl (struct file* pFile,
                      unsigned int cmd,
                      unsigned long pointer)
{
    return 0;
}

static ssize_t sd_read (struct file *filp,
                        char __user *buffer,
                        size_t count,
                        loff_t *f_pos)
{
    return count;
}

static ssize_t sd_write (struct file *filp,
                         const char __user *buffer,
                         size_t count,
                         loff_t *f_pos)
{
    return count;
}

// Driver: (de)initialization functions
static int __init sd_init_module (void)
{
    if (misc_register (&sd_devs))
    {
        printk ("Error, the misc driver cannot be registered\n");
        return (-ENODEV);
    }
    return 0;
}

static void __exit sd_cleanup_module (void)
{
    misc_deregister (&sd_devs);
}
```

```
module_init (sd_init_module);
module_exit (sd_cleanup_module);
```

Code from the previous driver creates a file named `/dev/driver_port` (see the value of the struct `sd_devs` field `name`) and makes that each time the file is accessed, the corresponding `sd_*` function is invoked

- when the file is opened with `open` function, `sd_open` is invoked. In this case, function returns 0 (success), so the file may be opened and a file descriptor returned as return value from `open` function.
- when the file is going to be released (when the number of `close` operations on the file descriptor match the number of times the file was `open`), `sd_release` is invoked which returns 0 (success), to the file is actually released.
- every time a `write` function is invoked on the file descriptor, the `sd_write` function on the driver is invoked. In this case, the later function does nothing (it doesn't write any byte) except returning that the requested number of bytes were written (`count` parameter).
- similar to `sd_write`, every time a read function is invoked on the file descriptor, the `sd_read` function on the driver is invoked. The function does nothing but return that the requested number of bytes were read (`count` parameter).

Exercise 2 (HOME): about sd_write and sd_read functions

a) Look carefully at the second parameter from sd_write and sd_read functions, which is declared as **`const char __user *buffer`**.

What do you think the "__user" modifier indicates here?

You may search on the book "*Linux device drivers*", on the chapter dedicated to *"char drivers*", in the section dedicated to "*read and write*".

b) What is the purpose of the the `f_pos` parameter? Why this parameter type is "`int*`" instead of "`int`"?

c) Is it possible for a user-space program to modify the value pointed by `f_pos` whithout invoking system calls read or write? If yes, how can it be done? What else has to be implemented in the driver in order to support this functionality?

Exercise 3 (HOME): about address spaces

a) On the book "Linux device drivers", on the chapter dedicated to "char drivers", in the section that tells about "*read and write*", search and describe the purpose of the following functions:

```
unsigned long copy_to_user(void *to,
                           const void *from,
                           unsigned long count);

unsigned long copy_from_user(void *to,
```

```
                                              const void *from,
                                              unsigned long count);
```

b) On the same book, on the chapter dedicated to *"Enhanced Char Driver Operations*", search and describe the purpose of the following macros:

```
put_user(datum, ptr)

get_user(local, ptr)
```

c) Explain the differences:

- between `copy_to_user` and `put_user`
- between `copy_from_user` and `get_user`

Exercise 4 (LAB): returning some data from the driver

a) Modify the `sd_read` function from the driver in such a way that when an application wishes to read N bytes, the user-supplied buffer is filled as described next:

- byte 0 of user buffer: 0
- byte 1 of user buffer: 1
- …
- Byte N-1 of user buffer: N-1

b) Install the driver and create a simple test program (in user-space) which:

1. opens the file `/dev/driver_port` using the open function
2. asks the user for an integer between 1 and 255 and stores the result in the `count` variable
3. reads `count` bytes from the `/dev/driver_port` file
4. validates that the each read byte N (with N from 0 to `count - 1`) has value N
   a. if the validation passes, an "SUCCESS" message is printed to console
   b. if the validation fails, a "FAIL" message is printed to console
5. closes the file descriptor

As a method to test the application:

- if test program is run when installed driver is the provided example, it should print "FAIL"
- if test program is run when installed driver is your modification, it should print "SUCCESS"

3.4. Blocking I/O Guide

3.5.1. Introduction

Blocking I/O (also called synchronous I/O) is a way to access data in such a way that the function which actually performs the read/write blocks until the access is complete.

Focusing on files and in Linux, to let application programmers read/write data from/to a file descriptor, following functions do exist:

```
ssize_t write(int fd, const void *buf, size_t count);
```

```
ssize_t read(int fd, void *buf, size_t count);
```

By default, previous functions will not return until the read (or write) operation completes.

Note: There is an exception to the previous statement: invoke the function `open` providing the flag `O_NONBLOCK`. This returns a file descriptor open in non-blocking mode. As this guide purpose is to illustrate blocking I/O, the `O_NONBLOCK` flag will be ignored from now on.

Blocking I/O must be approached with different strategies depending on programming side:

- **driver-side** (executed in kernel space): when writing a file driver, if the driver's access operations (implementation of write/read functions) get blocked, the driver should *behave well* with the rest of the system and not spend unnecessary resources (CPU and/or I/O resources) when it waits for the data access to complete.

- **application-side** (executed in user space): when writing an application that uses blocking I/O, latency of accesses and requirements for responsiveness of the application have to be taken into account, and depending on it:

  - **for** low **latencies** (like when writing a small file into the main HDD) or when **not requiring low response times**, normally it is ignored the fact that the I/O is blocking, either because the program will remain blocked for a few *ms* or because the program executes a process known to be lengthy and that does not need to be highly responsive.

    Examples of this scenario are:

    - writing a few bytes into a file located in the main (and fast) system storage

    - executing an off-line process, which acquires large amounts of data from a device and then processes it, without user interaction

  - for **high latencies**, and when the program needs to keep **responsive** even when some data access is blocking, normally the access is done in a separate thread (or process), to overcome the blocking due to data access.

Examples of this scenario are:

- a TCP/IP server client
- accessing a file located in a distant network mount with high latencies, and the user has interaction with the application

From now on, this guide will focus on blocking I/O from the driver-side.

3.5.2. Considerations for blocking I/O from a driver point of view

Blocking I/O is not a bad behaviour *per se*. As its been explained in the introduction, application developers have means to circumvent the block (for example using threads) if they need to.

But when developing a driver, the driver programmer should take into account the cases where the data accesses may be blocking and make the driver *behave well* with the rest of the system.

Related to *behave well*, following aspects will be explained and treated in the exercises proposed in this guide:

- not consuming more system resources than necessary while the block occurs
- let the system interrupt lengthy blocking I/O

3.5.3. Keeping low resource usage during blocking I/O

During transfer operations on the driver-side (acquire/send data from/to an external device, for example), two data access scenarios do exist:

- **synchronous** data accesses: in this scenario, the driver is directly responsible for the access operations.

  One example of such scenario would be reading/writing data from/into an external memory mapped device, where the driver must perform the necessary memory reads/writes to/from the user-space memory from/into the mapped memory region in order to communicate with the device.

  In this scenario, the driver is directly responsible for the copy operation between the user and device memory regions. Also, in this case, the data access is done through code which executes on the processor (for example using a loop which copies the memory byte-to-byte).

  A simple pseudo-code which illustrates this scenario is the following:

```
read function implementation in the driver with synchronous data transfer
{
    // real data access with the external device
    for each byte to be read from the external memory
    {
        copy one byte from external memory into a buffer in the driver
memory
    }
```

```
    // auxiliary code needed to return the read memory to user-side
    copy memory from the driver buffer into the user supplied memory
}
```

As it can be seen, this mechanism:

- is synchronous, because all operations are sequential (it is impossible to reach the auxiliary code without first finishing the execution of the loop)

- uses code for the data transfer, so each access in the loop uses processor instructions

- **asynchronous** data accesses: in contrast to the synchronous data access, this scenario features:

  - data **access** is asynchronous: which means that the driver code orders the data transfer, and then waits for the operation to complete

  - the driver code, although responsible for ordering the data transfer, may not do the data transfer by itself, but delegate the transfer to some other entity. In this case:

    - this some other entity has to provide means to notify back the driver code when the transfer is complete

    - it is possible that this some other entity is implemented in such a way that no processor instructions are executed during the transfer

One example of the described situation is where a driver delegates the data transfer to a DMA controller. DMA stands for **D**irect **M**emory **A**ccess, and is used to name a family of controllers which have the ability to copy bytes from one memory address into another without intervention from the processor. In order to use the mechanism provided by the DMA controller, the later has to provide:

- some way to receive data transfer orders

- some way to notify back that the data transfer has ended

To illustrate this mechanism, look at the following code:

```
read function implementation in the driver with asynchronous data
transfer
{
    // data access order
    tell the DMA controller to copy from ext. memory into a driver
buffer
```

```
    // dealing with asynchronous nature of the access operation
    wait for DMA controller to finish the data transfer

    // auxiliary code needed to return the read memory to user-side
    copy memory from the driver buffer into the user supplied memory
}
```

Details on the DMA controller present on the Beaglebone will not be seen on this guide's exercises. What is important here is to note the asynchronous nature of the data access operation. Note how the driver ***instructs*** the DMA controller to do the actual data transfer and then ***waits*** for it to finish the operation.

As the data access order is asynchronous, once the transfer is ordered the "order" function returns immediately, very probably before the transfer is complete. This is the reason for which a wait has to be done until the data transfer completes.

Remember that this is blocking I/O, so the read function shall not return until the data is completely read. Do not confuse the blocking nature of the read operation with the fact that the real data access is asynchronous. The driver enforces a blocking behaviour by waiting until the asynchronous data access completes.

Examples of drivers which use DMA transfers are: Ethernet drivers, UART drivers, disk drivers (for example RAID controllers when they have to replicate data from one physical disk to another), etc.

So in resume:

- in **synchronous** data accesses, the driver is responsible for the actual data access

- in **asynchronous** data accesses, most of the times the driver delegates the actual data access to an external entity

From now on, this guide will focus on **asynchronous** data accesses and its relationship to blocking I/O.

Two very important aspects of asynchronous data transfers are:

- how the driver comes to **know that the transfer is finished**: almost always this depends on the mechanism that performs the actual data transfer

- how the driver implements **the wait** until the data transfer is finished: and this is the key to low resource usage

At this point, it is important to understand well the difference about polling and interrupt mechanisms explained in the "**GPIO and interrupts from driver point of view**" guide.

From now on, special attention will be put in interrupts as a notification mechanism.

Keeping in mind that when the device which performs the actual data transfer will generate an interrupt whenever the transfer completes, a more concrete example for the read function in the diver may be the following (changes in blue):

```
read function implementation in the driver with asynchronous data transfer
{
    // data access order
    tell the DMA controller to copy from ext. memory into a driver buffer

    // dealing with asynchronous nature of the access operation
    wait until the "finish of transfer" interrupt is generated

    // auxiliary code needed to return the read memory to user-side
    copy memory from the driver buffer into the user supplied memory
}
```

And the next question which arises is the following: how to wait until the interrupt is generated?

A simple pseudo-code may the following (changes in blue):

```
"finish of transfer" ISR
{
    transfer_finished_flag = true
}

read function implementation in the driver with asynchronous data transfer
{
    transfer_finished_flag = false

    // data access order
    tell the DMA controller to copy from ext. memory into a driver buffer

    // dealing with asynchronicity nature of the access operation
    // (wait until the transfer has finished)
    while transfer_finished_flag is not true
    {
        do nothing
    }

    // auxiliary code needed to return the read memory to user-side
    copy memory from the driver buffer into the user supplied memory
}
```

In the previous pseudocode, notice how:

- until the asynchronous operation is not finished, the auxiliary code is not executed, so this is blocking I/O example

- it uses interrupts as a bridge between the asynchronous nature of the memory transfer operation and the need to have blocking I/O

Although that pseudocode would work, notice also how:

- although the “end of transfer” operation is managed asynchronously (using an ISR), the `while` loop is implementing a continuous poll on the “transfer finished” flag

- due to the continuous polling, the driver is using CPU in order to wait.

The effect of using CPU in order to wait for an operation to complete is a *bad behaviour*, in the sense that CPU cycles are *wasted* in order to wait, and not doing real work.

So another solution has to be found which doesn't use continuous polling of a state variable. For this purpose, a variety of mechanisms may be used in the driver code, but only one of them will be presented here: **wait queues**.

3.5.4. Wait queues

A wait queues is (like a semaphore or a mutex) a synchronous event notification mechanism between different code flows.

This synchronous nature of wait queues is presented with the following interface:

- a **declaration and initialization** function, which is needed in order to use the wait queue:

  ```
  DECLARE_WAIT_QUEUE_HEAD(wait_queue_name)
  ```

  This macro must be used in the global scope of the driver code (not inside any function), and declares and initializes a queue named `wait_queue_name`.

- a function that **waits for an event notification**, in a blocking manner. Code that invokes this function gets blocked until an event is notified to the wait queue:

  ```
  wait_event(wait_queue_name, condition)
  ```

  This macro must be used in the code whenever an event has to be waited on the queue named `wait_queue_name`. an event is notified to the wait queue. Additionally, `condition` must evaluate to a true value

  The macro will not return until both conditions apply:

  - an event is notified to the wait queue
  - the condition evaluates to true

- a function that **notifies an event**, and unblocks other parts of code which are blocked waiting for an event notification on the wait queue:

  ```
  void wake_up(wait_queue_head_t *queue_ptr);
  ```

  This function sends an event to the wait queue pointed by `queue_ptr`.

  Remember that in order to use `wake_up` in conjunction with `wait_event`, the `condition` parameter on the later has to be used properly.

The main advantage with wait queues over "waiting within a loop" is that a wait queue uses kernel primitives which are more CPU friendly than a continuous poll on a state variable.

Being more CPU friendly means that the (blocking) function which waits for the event doesn't waste CPU cycles. Instead, while the process which invokes the function that waits for an event is blocked, the kernel puts the process to sleep, which means that from an OS scheduler point

of view, the process gets assigned no CPU cycles until the awakening event is triggered. Then, when an event is notified to the wait queue, the blocked (and sleeping) process is awaken and this is the reason to the name of function `wake_up`.

A pseudocode which illustrates how to introduce wait queues into the blocking I/O example is the following (code related to wait queues in blue):

```
// (using wait queue API)
declare and initialize wait queue

"finish of transfer" ISR
{
    // (using wait queue API)
    // wake up the process
    notify wait queue event
}

read function implementation in the driver with asynchronous data transfer
{
    // data access order
    tell the DMA controller to copy from ext. memory into a driver buffer

    // (using wait queue API)
    // puts the process to sleep until event received in wait queue
    blocking call which waits for an event on the wait queue

    // auxiliary code needed to return the read memory to user-side
    copy memory from the driver buffer into the user supplied memory
}
```

All the previous explanation has been done to aid in the understanding of the following key points:

- there are accesses (like a memory transfer) which depend on non-processor peripherals (like a DMA controller)
- sometimes it is needed to write code which blocks even when the underlying functionality is asynchronous
- there are ways to implement *bad* (resource hungry) blocks and *good* (resource friendly) blocks

From now on, in this guide, the DMA concept will be left apart, and the focus will be put in the blocking nature of this kind of I/O operations and its effect on the programs that use blocking I/O.

3.5.5. Blocking I/O with polling

Exercise 1 (LAB): block with poll

Write a simple file driver with the following characteristics:

- following code may be used for open, release and write operations:

```
static int sd_open (struct inode* pInode, struct file* pFile)
```

```
{
    return 0;
}

static int sd_release (struct inode* pInode,
                       struct file* pFile)
{
    return 0;
}

static ssize_t sd_write (struct file *filp,
                         const char __user *buffer,
                         size_t count,
                         loff_t *f_pos)
{
    return count;
}
```

- the read operation must implement the following logic:

When this function is invoked, the function will wait until the push button (GPIO 20) in the beagle bone is pressed, then it can simply either write dummy values into user-supplied buffer or it can return the `count` parameter.

The key to this exercise is to implement the wait with a `while (!contidion) { ; }` kind of construct.

Exercise 2 (LAB): behaviour of block with poll

Write a simple user-space application which opens the file created on the driver of the exercise 1 and reads any number of bytes from the file (it can read only byte, this is not important for the exercise) and then exits.

Compile the test application and load the driver from exercise 1 into the Beaglebone.

Now, open two console sessions into the Beaglebone.

- In the 1st console, execute the user-space application
- In the 2nd, execute the **`top`** command and search for the test application

a) What is the CPU usage percentage for the user-space application? What do you think is the reason for such CPU usage?

b) In the console 1, press CTRL+C. What happens?

c) Now press the push button in the Beaglebone. Verify that the application ends its execution.

3.5.6. Blocking I/O with interrupts and wait queues

Exercise 3 (LAB): block with wait queues

Write a driver similar to the one written for the exercise 1, but with the following differences:

- in the driver global variables section, declare and initialize a wait queue.

- during the driver initialisation, register an ISR for the GPIO 20. The code for the ISR must notify an event into the wait queue.

- in the read operation code, wait on the wait queue. After the wait, the function does not need to do anything additional. Just return the value of the `count` parameter of the function.

Note that this will implement the block in a more sophisticated manner than in the exercise 1.

Exercise 4 (LAB): behaviour of block with wait queues

Make sure that the program created in the exercise 2 is copied into the Beglebone and compile, copy and install the driver created in the exercise 3 into the beaglebone. It is possible that you need to unload the driver from the exercise 1 in order to load the new driver.

Open two consoles in the Beaglebone.

- In the 1st console, execute the user-space application
- In the 2nd, execute the **`top`** command and search for the test application

a) What is the CPU usage percentage for the test application? Is it different from the CPU usage seen during the exercise 2? If yes, what do you think it is the reason?

b) In the console 1, press CTRL+C. What happens?

c) Now press the push button on the Beaglebone. Verify that the application ends its exection.

Exercise 5 (LAB): behaviour of signals in previous exercises

Analysing the result from exercises 2.b and 4.b, what do you think it is happening when a signal is sent to a process which is blocked on the underlying driver code?

3.5.7. Interruptible wait queues

As it's been seen on exercise 5, behaviour of `wait_event` is not very signal-friendly.

When writing a blocking I/O driver, signals also have to be taken into account, for the sake of driver completeness.

Related to behaviour observed in exercise 5, following alternative functions related to wait queues do exist:

```
wait_event_interruptible(queue, condition)
```

```
void wake_up_interruptible(wait_queue_head_t *queue);
```

Exercise 6 (HOME): consulting man-pages

Search for in the man-pages (you may use google) and explain the differences:

- between `wait_event` and `wait_event_interruptible`

- between `wake_up` and `wake_up_interruptible`

As its been explained in the guide about *File drivers*, there exists a *glue code* (the Virtual File System , VFS) between system call functions `read` / `write` and driver implementation of functions `sd_read` / `sd_write`.

The VFS not only connects system calls to driver implementation of functions, but also has some logic inside:

- if value returned from `sd_read` / `sd_write` functions is **non-negative**, this value is returned directly as result of `read` / `write` functions.

- if, instead, vale returned from `sd_read` / `sd_write` functions is **negative**, the VFS performs further checks on the value, and its behaviour depends of the value.

  For example, if `sd_read` / `sd_swrite` return value is –`ERESTARTSYS`, this means that:

  - `-ERESTARTSYS` accounts for RESTARTable SYStem call, which notified to the *VFS* that it may attempt to invoke again the function `sd_read` / `sd_write`.

    A use case for this situation is when the STOP signal is sent to a process which is waiting on a blocking I/O operation. When the block in the driver is interrupted (for example pressing CTRL-Z on the console where the process is executing, or issuing a **`kill -STOP <PID>`** command to the process), the driver may detect this situation (see result from exercise 6) and return `-ERESTARTSYS`.

    Approximately after the function returns this value to the VFS, the OS puts the process in paused state.

    When the process is ordered to continue execution again (executing the **`fg`** command after the previous CTRL-Z, or issuing a **`kill -CONT <PID>`** after the previous **`kill -CONT <PID>`** on the process), the VFS resumes execution and (knowing that the last invocation to the driver returned `-ERESTARTSYS`, may decide to restart the system call. This means that the driver function which returned this value will be invoked again using exactly the same parameters.

    From the user-application point of view, all this mechanism is transparent, meaning that the user-application will not notice anything special, just a successful call to `read` / `write` system calls.

When functions sd_read / sd_write do return a value >= 0, the glue code returns this value as a result

Exercise 7 (HOME): about restartable system calls

When a file driver function (for example `sd_read` / `sd_write`) returns `-ERESTARTSYS` value, it means that this function is a restartable system call. It is commonly said that the function which returns this value should be idempotent.

Search for the definition of idempotent related to computing and put an example (related to accesses to an external peripheral) of why is it desirable that the restartable system call be idempotent.

Exercise 8 (LAB): playing with interruptible wait queues

a) Modify the code from exercise 3 in the following way:

- for the wait code, use the interruptible versions of functions related to wait queues

- if the wait on the wait queue (see result of exercise 6) is interrupted due to a signal, return value `-ERESTARTSYS`

- at the beginning of the sd_read function, put this code:

```
printk ("entering sd_read function\n");
```

Now compile the driver, open two consoles into the Beaglebone, and:

b) On the 1st console, load the driver into the Beaglebone with `insmod` (you may need to remove the previous driver with `rmmod` in order to load the new driver). Now execute the application from exercise 2 in this console.

c) On the 2nd console, execute the command `dmesg`. Check the message "`entering sd_read function`" is shown.

d) On the 1st console, press CTRL+Z and then execute the command `fg`. This will pause the process and resume its execution.

e) On the 2nd console, execute the `dmesg` command again. Check the message "`entering sd_read function`" is shown twice.

f) Go again to 1st console and press the push button from the Beaglebone. Check that the program ends its execution.

3.5. *Kernel Probes Guide*

3.6.1. *Introduction*

Sometimes, it is convenient to have a lean and mean mechanism to intercept invocations to functions from the kernel, either because of a wish to avoid the Kernel Debugger or because information related to interceptions must be processed in some way which makes the Kernel Debugger not suitable for the process.

Kernel Probes let a driver *hook* itself into invocations to Kernel functions, and do some action when this happens.

This kind of *hooking* may be useful for analysis of code that executes on kernel-space (for example, to debug or profile a driver), giving insight about invocation to *functions-under-study*.

Some scenarios where this kind of help may prove useful are:

- studying how memory is allocated on the Kernel (for example, using `kmalloc`)
- analysing how a driver under study registers callbacks for certain interrupts
- studying how many times per second a certain Kernel function is invoked (for example, `open`)
- knowing the list of file descriptors that have been opened system-wide during a period of time
- etc…

Different types of Kernel Probes allow to gather information on different aspects of function's invocation, and there are three types of probes:

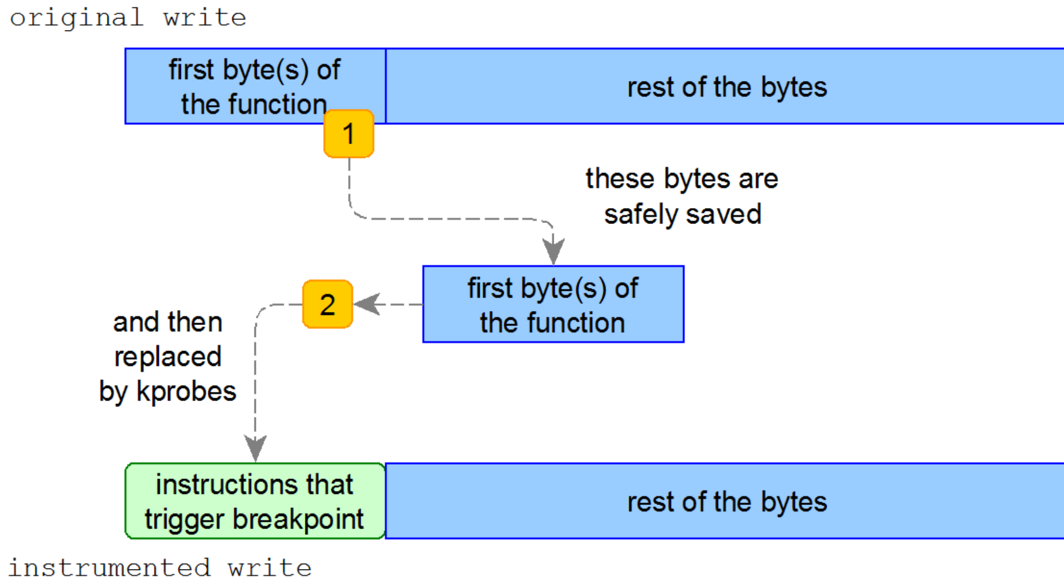
- *kprobes* allow to gather information right before a function is executed
- *jprobes* allow to gather information about the arguments used to invoke a function
- *return probes* allow to gather information right after a function is executed

3.6.2. *General mechanism*

As it is well explained on kernel documentation, *kprobes* names both the generic mechanism and a concrete type of probe. Here, the generic mechanism will be functionally explained.

kprobes mechanism does a little magic in order to instrument a function. Actually, it does not instrument functions, but instructions. So a *kprobe* can be effectively used to call a hook function whenever any instruction on the kernel space is hit. What will be seen on this guide is a concrete case where the hooked instruction is the first one of a function.

Visually, when a function is instrumented via a *kprobe*, two things happen:

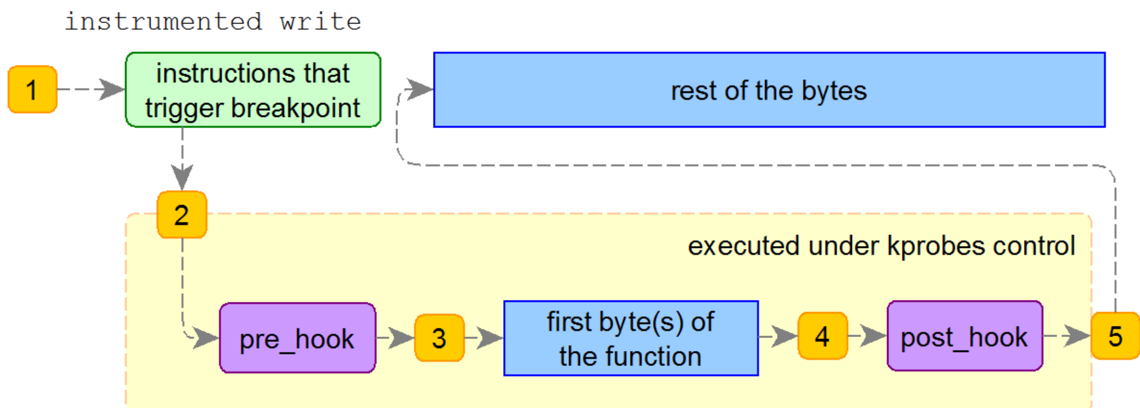


Which are explained here:

1) a copy of first function byte(s) is done, and safely saved because it needs to be executed later in order to reproduce original function behaviour

2) those saved first bytes are replaced with an instruction that triggers a breakpoint, and it's during breakpoint processing (which is managed by the *kprobes* system) that the hook function is called

Once the function is instrumented, when it comes to execute it, following sequence happens:



Which is explained here:

1) some code invoked the instrumented function, whose first bytes are now opcodes which generate a breakpoint

2) the breakpoint callback (when invoked) lets *kprobes* take control of the execution, and first thing it does is executing a *pre_hook* (user-definable via the *kprobes* API)

3) right after the *pre_hook* code is executed, previously saved first bytes of the function are executed with a process known as single-step (which means executing a single instruction at a time)

4) after the original first bytes of the function are executed, the *post_hook* (also user-definable) function is executed

5) at last, control is returned to the original function, which continues its execution normally

Note: during the rest of this guide, only *pre_hook* will be discussed.

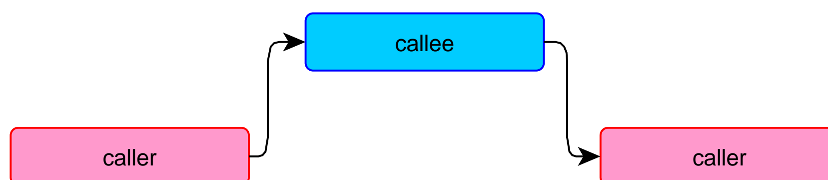
3.6.3. kprobes

The basic type of probe is the *kprobe*, which provides the basic functionality over which other types of probes are built.

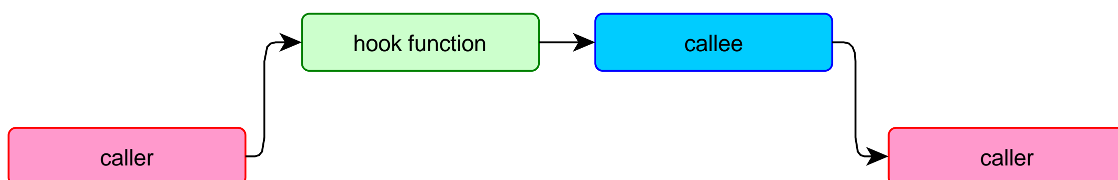
kprobes provide a convenient way to tell the Kernel: this function which I provide must be invoked whenever the instruction pointer reaches that address. This means that *kprobes* can be used to call a concrete function (the probe) whenever code execution reaches any point.

Despite of what the previous paragraph says, the *kprobes* mechanism provides a convenience way to supply a function name instead of having to know its address (which is in any case is trivial when using C).

So, it turns out that *kprobes* give name to both the generic mechanism and a concrete interception functionality. If this was invocation flow for a given function:



Following would be invocation flow when using a *kprobe*:



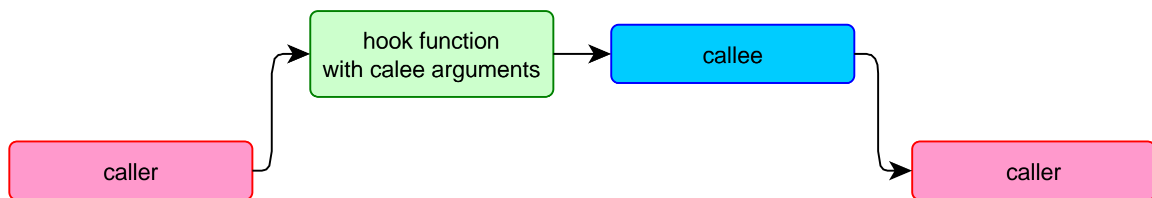
As it is depicted above, using a *kprobe* enables intercepting (and only intercepting) invocations of selected functions from the Kernel (neither knowing of modifying arguments supplied to the hooked function is possible using *kbprobes*).

In some cases, it can be interesting to intercept a function call right before the function is invoked. For example:

- for logging purposes, it may be interesting to know when `recv` is invoked on a socket, because waiting until the `recv` function returns may produce a not desired delay on the logs
- for debugging purposes, if it is known that invoking `write` on a certain file driver sometimes hangs the driver, it is mandatory to long exactly when `write` is invoked before execution of its code

3.6.4. jprobes

jprobes are built on top of *kprobes*, and allow to intercept the moment a function is called, providing means to inspect the arguments passed to the intercepted function.



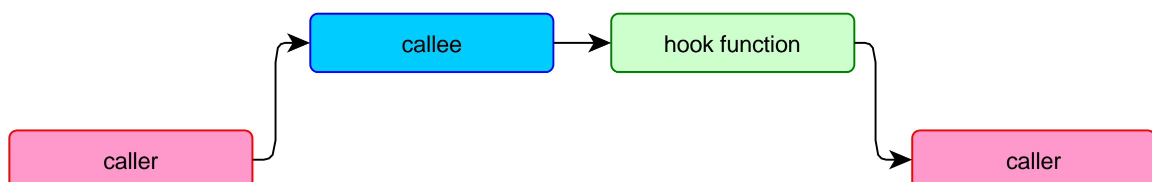
Note that using *jprobes* it is not possible to alter the arguments passed to the originally called function.

Some cases where knowing the exact moment where a function is invoked as well as its additional arguments are:

- for statistics gathering purposes, it can be useful to know exactly how much memory is requested to the kernel (for example using `kalloc`) during a certain time window. Hooking a *jprobe* into the `kalloc` function would allow to determine the exact amount of RAM requested at each invocation of the memory allocator.
- for debugging purposes, if is known that invoking `write` on a certain unknown file produces some undesired effect, it may be interesting to capture write calls and log some information about the write into a log file. Later, it may be possible to inspect the log to obtain further information about the error.

3.6.5. Return probes

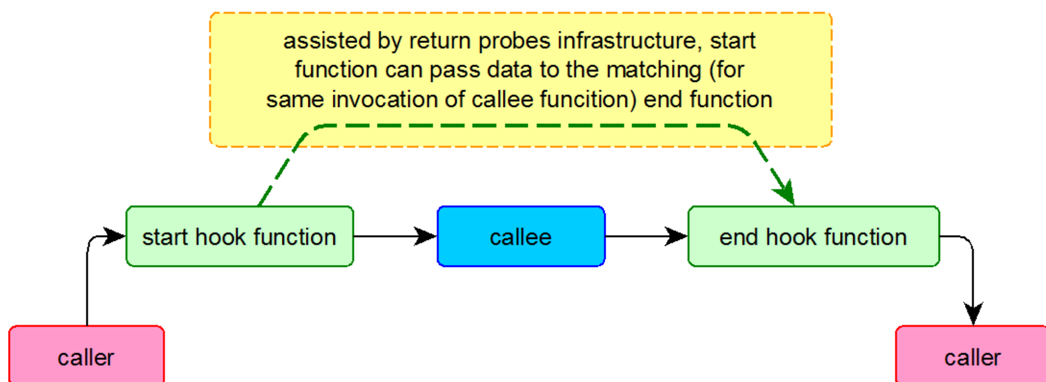
Return probes are also built on top of *kprobes*, and allow to intercept a function call, but this time at the moment when the function is about to return:



Just as it happens with *kprobes*, *return probes* enable to intercept the return from a function, but do not allow to either modify or inspect the return value from the inspected function.

In some cases, it may be interesting to intercept a function call just before the function returns. For example:

- for modelling purposes, it may be interesting to know the exact moment when the `recv` function returns, because it may take a while for the `recv` function to execute, and perhaps it is needed to know the exact moment when data is available to some driver for example to characterize system CPU load as function of received data frequency.
- for statistics gathering purposes, it may be interesting to know the total time spent on the `write` function, and a *return probe* can also be used for this purpose. It's possible to specify *both* a *start of function* and an *end of function* hook. In order to match the *start event* with the *end event*, return probes allow to pass data between the two functions using a temporary structure. This scenario is illustrated here:



When using a return probe, and in order to ease matching the event produced at function it is also possible to specify an *entry hook*

3.6.6. No need for instrumented functions' source code

As it's been roughly sketched through previous examples, *kprobes* can be useful to gather information about Kernel functions' usage.

One important fact about them is that it isn't needed to have the source code for the instrumented functions, just its addresses. But, in case it is desired to reference functions by symbol name instead of address, it'll be needed to have a running Kernel with symbolic information (which is an option that can be enabled during Kernel compilation). Regarding this last point, note that during driver prototyping (and this also applies to more areas of software development), it's usual to have special versions of the Kernel prepared with debug information and maybe other options that ease debugging and analysis processes.

Previous paragraph also means that *kprobes* can be used to analyse behaviour of third-party drivers (although a disassembler could also do a good work here on this scenario).

3.6.7. kprobes API

If Linux kernel is built with *kprobes* support, following functions will be available (examples will be seen later):

Functions related to *kprobes*:

```
int register_kprobe(struct kprobe *p);
```

This function registers a *kprobe*. Usually drivers register *kprobes* during its initialization.

Return value for this function is 0 on success and <> 0 in case of some error.

Argument for this function is a pointer to a *kprobe* structure, whose most important fields are:

```
struct kprobe {
   /* omitting other fields.
   refer to documentation to get further information *
   ...

   /* location of the probe point */
   kprobe_opcode_t *addr;
   /* Allow user to indicate symbol name of the probe point */
   const char *symbol_name;
   /* Called before addr is executed. */
   kprobe_pre_handler_t pre_handler;
}
```

Where:

- `addr` is the address of the function which is desired to install the *kprobe* at
- `symbol_name` is the name of the function which is desired to install the *kprobe* at. This filed can be used instead of the `addr` field.
- `pre_handler` is the address of the function which will be called before the probed function is executed

Type of `pre_handler` fields is defined as follows:

```
typedef int (*kprobe_pre_handler_t) (struct kprobe *, struct pt_regs
*);
```

```
void unregister_kprobe(struct kprobe *p);
```

Unregisters a *kprobe*. Drivers usually unregister *kprobes* during its deinitialisation.

Returns no value.

Argument for this function is a pointer to a previously registered *kprobe*.

Functions related to *jprobes*:

```
int register_jprobe(struct jprobe *p);
```

This function registers a *jprobe*. Usually drivers register *jprobes* during its initialization.

Return value for this function is 0 on success and <> 0 in case of some error.

Argument for this function is a pointer to a *jprobe* structure, whose most important fields are:

```
struct jprobe {
    struct kprobe kp;
    void *entry;    /* probe handling code to jump to */
}
```

Where:

- `kp` is a `kprobe` struct which is used to indicate the symbol which is desired to install the *jprobe* at
- `entry` is a pointer to a function which will be called with the arguments of the probed function, which means that the function pointed to by `entry` must have the EXACT SAME signature (return type and arguments) as the probed function.

  Also, in order for *jprobes* to be able to continue executing the probes function, this must be the return code for the function:

  ```
  /* Always end with a call to jprobe_return(). */
  jprobe_return();
  return 0;
  ```

```
void unregister_jprobe(struct jprobe *p);
```

Unregisters a *jprobe*. Drivers usually unregister *jprobes* during its deinitialisation.

Returns no value.

Argument for this function is a pointer to a previously registered *jprobe*.

Functions related to *return probes*:

```
int register_kretprobe(struct kretprobe *p);
```

This function registers a *kretprobe*. Usually drivers register *kretprobes* during its initialization.

Return value for this function is 0 on success and <> 0 in case of some error.

Argument for this function is a pointer to a *kretprobe* structure, whose most important fields are:

```
struct kretprobe {
    struct kprobe kp;
    size_t data_size;
    kretprobe_handler_t handler;
    kretprobe_handler_t entry_handler;
    int maxactive;
};
```

Where:

- `kp` is a `kprobe` struct which is used to indicate the symbol which is desired to install the *kretprobe* at
- `data_size` instructs *kprobes* library to allocate `data_size` bytes of space for each *kretprobe* structure passed as argument to the `handler` / `entry_handler` functions. This will be explained later.

- `entry_handler` is a pointer to a function which will be called once the probed function starts its execution. Can be NULL, then no *hook* function will be called when the probed function starts its execution. Type of this function is:

  ```
  typedef int (*kretprobe_handler_t) (struct kretprobe_instance *,
                                      struct pt_regs *);
  ```

  `kretprobe_instance` structure has one very important field:

  ```
  struct kretprobe_instance {
      char data[0];
  };
  ```

  Data is really a pointer to a memory address. Available space at the memory address is the value of the `data_size` field for the `kretprobe` structure.

  This same pointer to the `kretprobe_instance` will also be passed to the function pointed by the `handler` field of the `kretprobe` structure.

  Therefore, this `data` field can be used to pass data between the `entry_handler` function and the `handler` function, providing a convenient wat to match the two callbacks to the same probed function invocation.

  This means that (for example) if an `int` value is desired to be passed from the `entry_handler` function to the `handler` function, first `sizeof(int)` should be assigned to the `data_size` field of the `kretprobe` structure. Then, from the `entry_handler` function, an integer could be stored at the address pointed to by the `data` field of the `kretprobe_instance` structure pointer, which could be later read from the handler function.

- `handler` is a pointer to a function which will be called once the probed function ends its execution. Type of this function is:

  ```
  typedef int (*kretprobe_handler_t) (struct kretprobe_instance *,
                                      struct pt_regs *);
  ```

  `kretprobe_instance` structure pointer is the same pointer that was passed to the `entry_ handler` function, and this is what allows passing data from `entry_handler` to `handler`.

- `maxactive` sets the maximum concurrent invocations of the probed function which can be simultaneously traces. Having this field with a too low value may produce the effect of missing some hits of the probed function. A safe value for this field is -1. Look *kprobes* documentation for further information about this field.

```
void unregister_kretprobe(struct jprobe *p);
```

Unregisters a *return probe*. Drivers usually unregister *return probes* during its deinitialisation.

Returns no value.

Argument for this function is a pointer to a previously registered *kretprobe*.

3.6.8. About commented usage examples

To illustrate behavior of previous functions, some commented code samples are provided next.

Each example illustrates a very simple *driver* which uses each of the concepts described at the *API Usage* section:

- for *kprobes*, a function is probed and its invocations counted
- for *jprobes*, a function is probed and one of its arguments is summed along all invocations
- for *return probes*, a function is probed *before* and *after* execution, and execution time is measured

On all examples, the selected probed function is `vfs_write`, which is used by the virtual file system (VFS) access functions behind the common *syscall* `write` function (which is available for user-space programs).

Source code for write function is the following:

```
SYSCALL_DEFINE3(write, unsigned int, fd, const char __user *, buf,
                size_t, count)
{
    struct fd f = fdget(fd);
    ssize_t ret = -EBADF;

    if (f.file) {
        loff_t pos = file_pos_read(f.file);
        ret = vfs_write(f.file, buf, count, &pos);
        file_pos_write(f.file, pos);
        fdput(f);
    }

    return ret;
}
```

Two things can be seen here:

- This function declaration is a little bit different from the usual.

  Instead of defining the `write` function directly, `SYSCALL_DEFINE3` macro is used. This is due to some internal mechanisms for Kernel compilation.

  Generally, all *syscall* functions which are implemented at the Kernel are defined this way.

  There is a file `<linux/syscalls.h>` which defines what this macro is:

```
#define    SYSCALL_DEFINE3(name,    ...)    SYSCALL_DEFINEx(3,    _##name,
__VA_ARGS__)
```

  This is complex macro (which uses another macro, concatenation operator and variable arguments ad), but the important fact is that it ends using the `SYSCAL_DEFINEx` macro, which the most important thing it does is declaring the functions with `asmlinkage` directive.

HOME: Exercise 1 - `asmlinkage`

Search for the meaning of `asmlinkage` keyword in GCC compiler.

What differentiates a function that is declared using `asmlinkage` from one that does not use it?

- write function internally uses `vfs_write` function to do its work.

  Due to the write function using `asmlikage` option (it is a *syscall*) and not to complicate examples' code, instead of probing the `write` function (it would provoke the handler function for the *jprobe* case) to also be defined with `asmlinkage`.

  This is the reason to install the probes at `vfs_write` function.

LAB: Exercise 2 - syscall internal kernel functions

Looking at the previous definition of *syscall* `write` function definition, search the definition of the following *syscalls* (on the kernel source) and tell which is the internal kernel function that the each of the following *syscalls* use:

- `open` (opens a file and returns a file descriptor)
- `read` (reads bytes from a file descriptor)
- `fork` (forks the current process creating a child process)
- `unlink` (removes a file or directory)
- `send` (sends data through a socket)

And last, all examples use a semaphore to access the counter variables, to avoid race conditions when kernel is compiled with preemption enabled.

HOME: Exercise 3 - kernel preemption

Search for the definition of preemption and explain it with your own words.

Does kernel preemption takes effect on uniprocessor systems?

3.6.9. Commented kprobes usage example

Here follows source code for a driver which uses *kprobes* mechanism. This driver counts the number of times that the `vfs_write` kernel-function is invoked.

```
// Needed by the driver itself
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>

// Needed by kprobes
#include <linux/kprobes.h>
#include <linux/ptrace.h>

#include <linux/semaphore.h>

#define MODULE_TRACE KERN_ALERT "kprobes-1: "
static struct semaphore sem;
```

```
static volatile unsigned int vfs_write_counter = 0;

// The hook function
int vfs_write_pre_handler(struct kprobe *p, struct pt_regs *regs)
{
        // Use a semaphore to avoid a race condition for vfs_write_counter
        down (&sem);
        vfs_write_counter++;
        up (&sem);
        return 0;
}

// kprobes usage
static struct kprobe kprobe_write = {
        .symbol_name = "vfs_write",
        .pre_handler = vfs_write_pre_handler
};

// Module initalization
static int __init kprobes_module_init (void)
{
        printk (MODULE_TRACE "initializing\n");

        int error = 0;

        if (0 != (error = register_kprobe (&kprobe_write)))
        {
                printk (MODULE_TRACE "error %d registering kprobe\n", error);
                return -1;
        }
        printk (MODULE_TRACE "kprobe registered\n");

        sema_init (&sem, 1);

        printk (MODULE_TRACE "initialized\n");

        return 0;
}

// Module clean up
static void __exit kprobes_module_exit (void)
{
        printk (MODULE_TRACE "write function hit %u times\n",
vfs_write_counter);

        unregister_kprobe (&kprobe_write);
        printk (MODULE_TRACE "kprobe unregistered\n");

        printk (MODULE_TRACE "cleaned up\n");
}

module_init (kprobes_module_init);
module_exit (kprobes_module_exit);

MODULE_LICENSE ("GPL");
MODULE_VERSION ("1.0");
```

Main function of previous code (from a *kprobes* point of view), is `vfs_write_pre_handler`, which simply counts the number of invocations done to `vfs_write` function.

During static initialization, only two fields are set for the `kprobe_write` structure:

- `symbol_name`, which is set to the name of the probed function
- `pre_handler`, which is set to the address of the probe (`vfs_write_pre_handler`)

Note how the signature of `vfs_write_pre_handler`...

```
int vfs_write_pre_handler(struct kprobe *p, struct pt_regs *regs)
```

... is equal to that of the type `kprobe_pre_handler_t`:

```
typedef int (*kprobe_pre_handler_t) (struct kprobe *, struct pt_regs *);
```

And for the hit counter variable (vfs_write_counter):

- it's statically initialized to 0
- it's increased during each probe hit
- when the module gets unloaded, its value is printed

3.6.10. Commented jprobes usage example

Here follows source code for a driver which uses *jprobes* mechanism. This driver counts the number of bytes ordered to write to the `vfs_write` kernel-function.

```
// Needed by the driver
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>

// Needed by kprobes
#include <linux/kprobes.h>
#include <linux/ptrace.h>

#include <linux/semaphore.h>
#include <linux/fs.h>

#define MODULE_TRACE KERN_ALERT "kprobes-2: "
static struct semaphore sem;

// The hook function
static volatile unsigned long long int written_bytes_counter = 0;

// Following function is declared at <linux/fs.h>, from there we can extract
the function signature
ssize_t vfs_write_hook (struct file * p_file, const char __user * p_buf,
size_t num_bytes, loff_t * pos)
{
        // Use a semaphore to avoid a race condition for written_bytes_counter
        down (&sem);
        written_bytes_counter += num_bytes;
        up (&sem);
        /* Always end with a call to jprobe_return(). */
        jprobe_return();
        return 0;
}

// jprobes usage
```

```
static struct jprobe jprobe_vfs_write = {
        .kp = {
                .symbol_name = "vfs_write",
        },
        .entry = vfs_write_hook
};

// Module initalization
static int __init jprobes_module_init (void)
{
        printk (MODULE_TRACE "initializing\n");

        int error = 0;
        if (0 != (error = register_jprobe (&jprobe_vfs_write)))
        {
                printk (MODULE_TRACE "error %d registering jprobe\n", error);
                return -1;
        }
        printk (MODULE_TRACE "jprobe registered\n");

        sema_init (&sem, 1);

        printk (MODULE_TRACE "initialized\n");

        return 0;
}

// Module clean up
static void __exit jprobes_module_exit (void)
{
        printk (MODULE_TRACE "written %llu KB\n", written_bytes_counter/1024);

        unregister_jprobe (&jprobe_vfs_write);
        printk (MODULE_TRACE "jprobe unregistered\n");

        printk (MODULE_TRACE "cleaned up\n");
}

module_init (jprobes_module_init);
module_exit (jprobes_module_exit);

MODULE_LICENSE ("GPL");
MODULE_VERSION ("1.0");
```

Main function of previous code (from a *jprobes* point of view), is `vfs_write_hook`, which simply counts the number of “bytes to be written“ value from the argument passed to the probed (`vfs_write`) function.

During static initialization, only two fields are set for the `jprobe_vfs_write` structure:

- as explained in the API reference section, to specify the probed address, sub-structure `kp` (which is a `struct kprobe`) is used to set the probed function name. As in the case of the *kprobes* example, only `symbol_name` field of the sub-structure is needed to be set.
- `entry`, which is set to the address of the probe (`vfs_write_hook`)

Note how the signature of `vfs_write_hook`...

```
ssize_t vfs_write_hook (struct file * p_file, const char __user * p_buf,
size_t num_bytes, loff_t * pos)
```

... matches signature of the probed function (`vfs_write`):

```
ssize_t vfs_write(struct file *file, const char __user *buf, size_t count, loff_t
*pos)
```

And for the counter variable (`written_bytes_counter`):

- it's statically initialized to 0
- the number of bytes desired to be written is added to it on each invocation to the probed function
- when the module gets unloaded, its value is printed

3.6.11. Commented return probes usage example

Here follows source code for a driver which uses *jprobes* mechanism. This driver counts the total time spent inside the `vfs_write` kernel-function, and the mean time spent for each invocation to the function.

```
// Needed by the driver
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>

// Needed by kprobes
#include <linux/kprobes.h>
#include <linux/ptrace.h>

#include <linux/semaphore.h>

#define MODULE_TRACE KERN_ALERT "kprobes-3: "
static struct semaphore sem;

// The hook functions
static volatile unsigned long long int
    vfs_write_total_spent_time = 0;
static volatile unsigned long long int
    vfs_write_invocation_counter = 0;

int vfs_write_entry_handler(struct kretprobe_instance *p, struct pt_regs
*regs)
{
    // Use a semaphore to avoid a race condition for
    // vfs_invocation_counter variables
    down (&sem);
    vfs_write_invocation_counter++;
    up (&sem);

    // Store current time
    *((unsigned long long int*)p->data) = ktime_to_ns(ktime_get());

    return 0;
}

int vfs_write_ret_handler(struct kretprobe_instance *p, struct pt_regs *regs)
```

```
{
    // Calculate spent time between entry and now
    unsigned long long int current_time= ktime_to_ns(ktime_get());
    unsigned long long int spent_time = current_time - *((unsigned long long
int*)p->data);

    // Use a semaphore to avoid a race condition for
    // vfs_write_total_spent_time variables
    down (&sem);
    vfs_write_total_spent_time += spent_time;
    up (&sem);
    return 0;
}

// kretprobes usage
static struct kretprobe kretprobe_write = {
    .kp = {
        .symbol_name = "vfs_write",
    },
    .handler = vfs_write_ret_handler,
    .entry_handler = vfs_write_entry_handler,
    .maxactive =1,  // use default value

    // data will store start time of the function
    .data_size = sizeof (unsigned long long int)
};

// Module initalization
static int __init kretprobes_module_init (void)
{
    printk (MODULE_TRACE "initializing\n");

    int error = 0;

    if (0 != (error = register_kretprobe (&kretprobe_write)))
    {
        printk (MODULE_TRACE "error %d registering kretprobe\n", error);
        return -1;
    }
    printk (MODULE_TRACE "kretprobe registered\n");
    sema_init (&sem, 1);

    printk (MODULE_TRACE "initialized\n");

    return 0;
}

// Module clean up
static void __exit kretprobes_module_exit (void)
{
    printk (MODULE_TRACE "vfs_write function hit %llu times\n",
            vfs_write_invocation_counter);
    printk (MODULE_TRACE "total time spent time in vfs_write function is %llu
ns\n",
            vfs_write_total_spent_time);
    printk (MODULE_TRACE "spent %llu ns per invocation to vfs_write
function\n",
            vfs_write_total_spent_time/vfs_write_invocation_counter);
```

```
    unregister_kretprobe (&kretprobe_write);
    printk (MODULE_TRACE "kretprobe unregistered\n");

    printk (MODULE_TRACE "cleaned up\n");
}

module_init (kretprobes_module_init);
module_exit (kretprobes_module_exit);

MODULE_LICENSE ("GPL");
MODULE_VERSION ("1.0");
```

From a *return probes* point of view, previous code has two main functions:

- `vfs_write_entry_handler`, which is the *hook* invoked at the beginning of `vfs_write` function. This hook counts the number of invocations to the probed function and stores the invocation time of the probed function inside the `data` field of the `kretprobe_instance` pointer.

- `vfs_write_ret_handler`, which is the *hook* invoked at the end of vfs_write function. This hook accesses the `data` field of the `kretprobe_instance` pointer and uses it to calculate the time spent on the write function, which sums to a counter.

During static initialization, following fields of the `kretprobe_write` structure are set:

- just as in the *jprobes* case, `kp` sub-structure is used to specify the probed function

- `handler` is set to the address of the function `vfs_write_ret_handler`, which will be executed upon probed function return

- `entry_handler` is set to the address of the function `vfs_write_entry_handler`, which will be executed before probed function starts execution

- `maxactive` is set to the recommended (see API section) value -1

- `data_size` is to the size of the data that `vfs_write_entry_handler` will store into the data pointer field of the `kretprobe_instance` structure

- `etry`, which is set to the address of the probe (`vfs_write_hook`)

It is very important to note that, as both `vfs_write_entry_handler` and `vfs_write_ret_handler` are specified on the same return probe structure, the pointer to the `kretprobe_instance` argument to both functions will be the same is they both are called for the same probed function invocation. This is what allows to store the time there from `vfs_write_entry_handler` function and use it at `vfs_write_ret_handler` function to calculate time spent on the probed function.

Note how the signature of both `vfs_write_entry_handler` and `vfs_write_ret_handler`...

```
int vfs_write_entry_handler(struct kretprobe_instance *p, struct pt_regs
*regs)

int vfs_write_ret_handler(struct kretprobe_instance *p, struct pt_regs *regs)
```

... is equal to that of the type `kretprobe_handler_t`:

```
typedef int (*kretprobe_handler_t) (struct kretprobe_instance *,
                                     struct pt_regs *);
```

About the hit counter variable (`vfs_write_invocation_counter`):

- it's statically initialized to 0
- it's increased during each time the probed function is about to begin execution
- when the module gets unloaded, its value is printed

And for the time counter variable (`vfs_write_total_spent_time`):

- it's statically initialized to 0
- at each probed function invocation, the value for the counter is increased by the time calculated as: $\text{time}_{\text{probed function ends}} - \text{time}_{\text{probed function starts}}$
- when the module gets unloaded, its value is printed

Also, when the module gets unloaded, value of both counters is used to calculate and print the mean time spent on each execution of the probed function.

LAB: Exercise 3 – testing drivers

Compile and test the programs. Check them are behaving as expected and show some example of its output.

3.6. Thread Local Storage (TLS) Guide

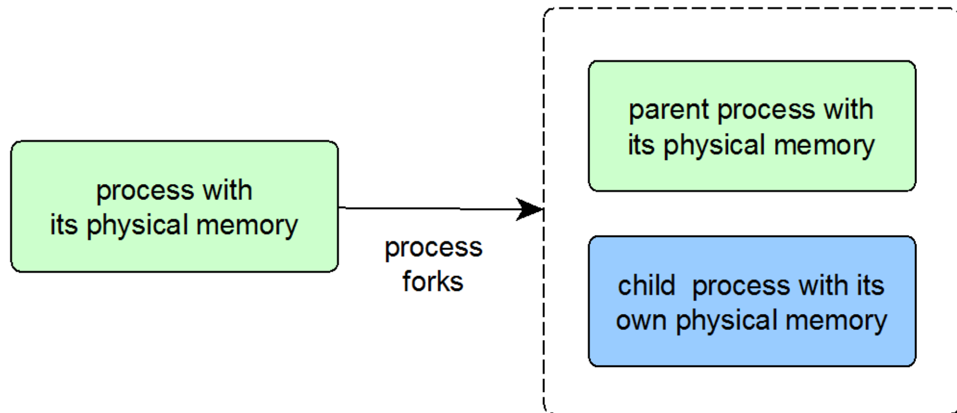
pre-requisite: background in thread programming

3.7.1. Introduction

Related to concurrent programming, during the design phase of an application, at some point a decision has to be made to choose either a multithreaded or a multiple process approach (or a combination of both).

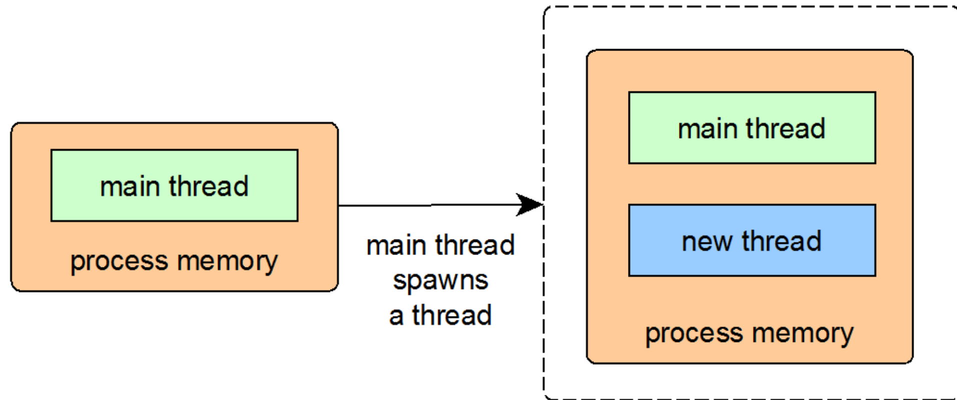
Regarding this decision, from an application programmer point of view, one of the main differences between the two approaches is the memory management implied on these two scenarios:

- in a multiple process scenario, each process is assigned a private memory mapping. Thus, by default the memory seen by each process is isolated (one process cannot access another process' memory)



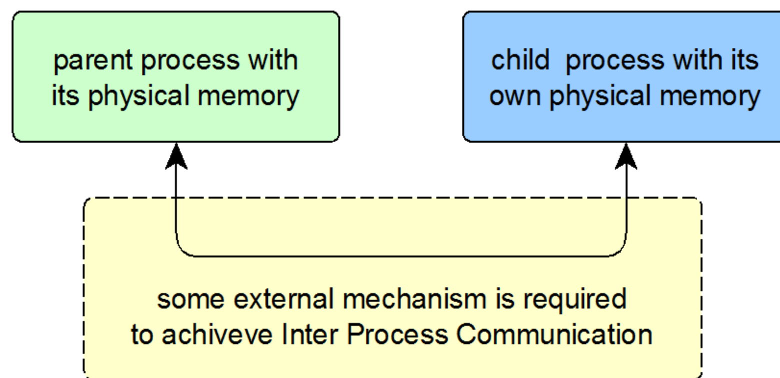
As a side note, when a process is forked it is possible that a Copy-On-Write strategy is used to achieve a faster process spawn operation, so during some time interval (until either of the processes do a write access to the memory) the physical memory seen by each process can be the same, but this is a concern which will not be further seen in this guide.

- in a multithreaded scenario, the memory mapping to which each thread has access is the same. Thus, by default the memory is shared between threads (all threads access the same physical memory)

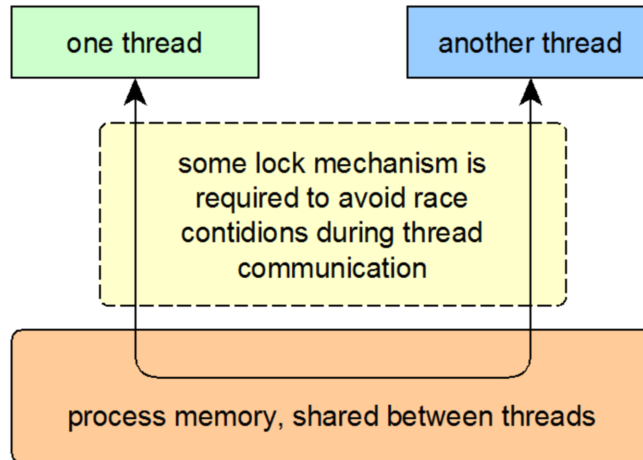


Broadly looking into which are implications for this difference in memory management, in general the following rule applies:

- in a multiple process scenario, an additional effort has to be done to enable communication between the processes (just because the memory model pushes towards isolated memory between processes), and therefore a variety of Inter Process Communication (IPC) techniques do exist which give support to the communication needs (some examples are signals, sockets, shared memory, files, ...)



- in a multithreaded scenario, if memory is used to communicate data between threads, ad additional effort has to be done when accessing this memory (for example to avoid race conditions), and therefore a variety of lock mechanisms do exist which give support to ensure exclusive access to the memory used to communicate the threads (some examples are binary semaphores, mutexes, file locks, ...)

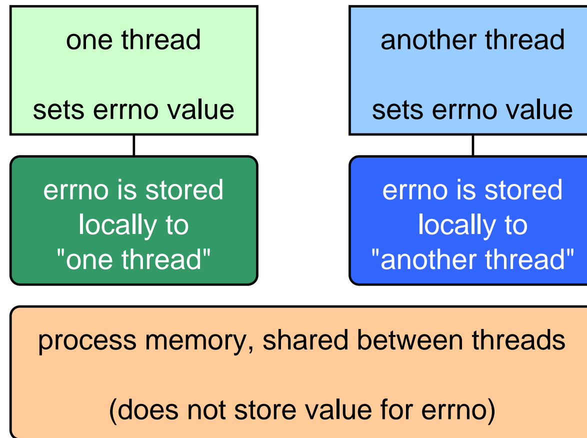


This shared behaviour has more implications than the ones described for thread communication, and another example of problems derived from this very nature is when using libraries which are not multithread-ready: back in the old days when threads were first introduced, special care had to be taken when migrating code to multithread, as some functions might use global variables (which due the shared-memory-nature of multithread programming, is subject to race conditions). An example of this situation is the `strtok` function (which splits a string using a delimiter token), whose usage implies that a state variable has to be stored as a global variable, and thus concurrent invocations of this function from different threads may corrupt this variable. To avoid this kind of situations, standardised C functions which are subject to thread-safety problems usually do have an equivalent version of the function which is suitable for use in a multithread environment (the `strtok` function has the equivalent `strtok_r` function which is thread-safe).

As is been seen by example, in the scenario of the previous paragraph, the usual workaround is to provide the programmer with an equivalent but thread-safe manner of invoking a function, and this usually implies using another version of the function with a similar name.

But there are cases different to that of the previous scenario: for example, the omnipresent (in C) `errno` variable, which is set by system calls and some library functions is the case of an error as a hint on what went wrong. This variable is global: it is not scoped (into a file, or a function, or whatever), but it's available from every part of the code. Furthermore, implementation of `errno` in thread-safe programs ensures that writing the value for this *global* variable from one thread does not affect the value another thread sees for this variable. And, at the same time, the way of reading/setting this variable is uniform across threads.

What the previous paragraph suggests is that there exists some kind of storage which is local to a thread:



And what is more important, the code each thread uses to set this *thread-local variable* is the same, and this enables an important fact: an example with two worker threads which do the same processing can use exactly the same code.

The previous functionality is achieved with which is known as Thread Local Storage (TLS), which means that there is some data which is private to a thread. The `errno` variable can be seen as an usage of this concept.

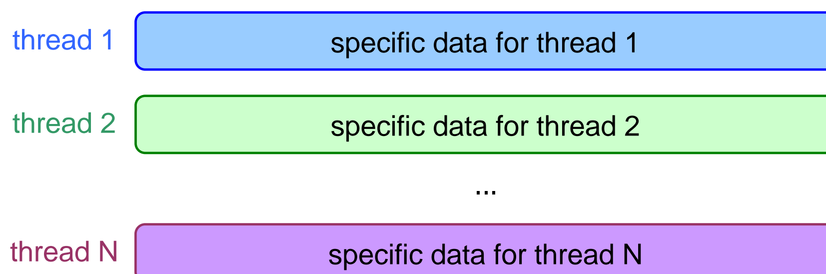
3.7.2. Outline of TLS' inner working

With TLS, an application programmer can store values inside a thread context, in such a way that when the value is set/get, the value is stored/read to/from a container which depends on the thread.

More on the previous statement below, but first let's see the idea behind TLS.

When using `libpthread` in order to give multithread functionality to an application, the library takes care for some initializations related to the threads it manages, and one part of *taking care* is allocating a dedicated data space related to every existing thread known as *specific data*.

Then, if for example an application has N threads, they automatically get some memory assigned for *specific data* purposes:



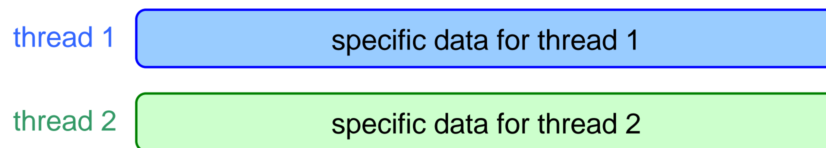
In order to manage threads, usually the functions from `libpthread` are used, what gives the library control over the threads' lifecycle, and this allows the library to allocate/free the memory related to the *specific data* for every application thread.

The TLS implementation is built on top of this very basic functionality, and when used from an application reminds the usage of a dictionary:

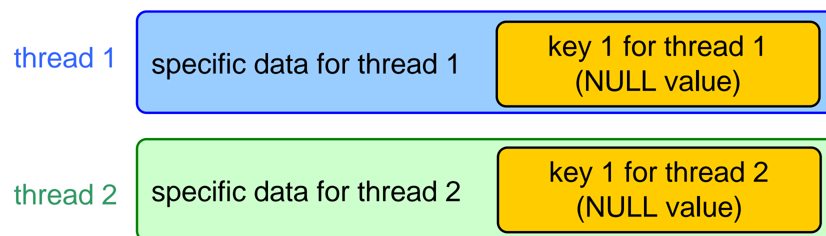
- first, the keys for the *dictionary* need to be created
- then, a thread sets/reads the value for some already created key

And that's all. To illustrate the previous explanation, imagine the following scenario, which is split across some steps:

a) an application creates two threads, so the library in charge of managing threads allocates some memory for the specific data for each thread:

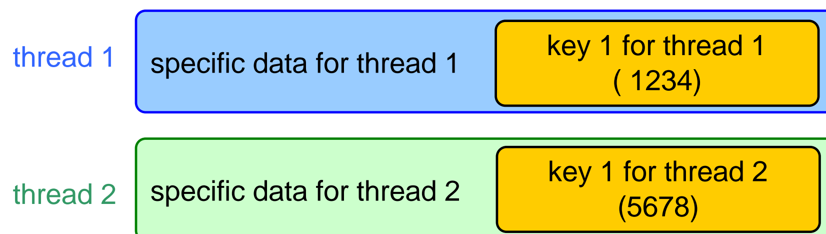


b) the application wishes to store some value, which will be global in the context of every thread but isolated from thread to thread. Then it decides to create a key (calling it **key1**):

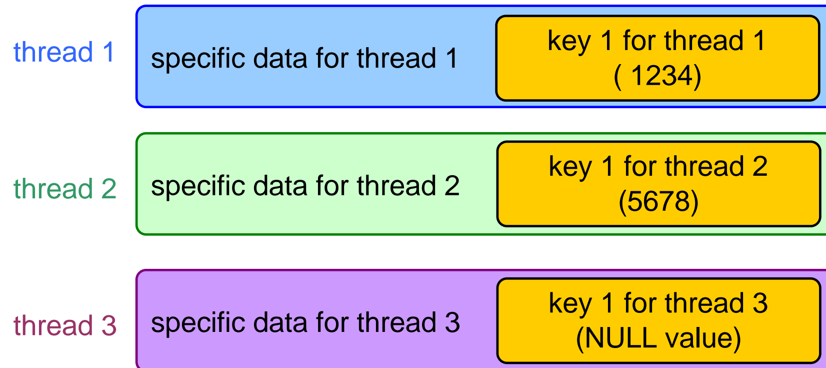


What happened here is that the library (when invoked to create a key), initialized the key on both threads.

Now thread 1 stores the value 1234 and thread 2 stores the value 5678, under the newly created key:



c) the application creates another thread, so the library takes care of automatically creating the key for the new thread (of course with a NULL value, indicating that the value is not yet initialized):



TLS implementation is also done at the `libpthread`, whose source code is part of the `glibc` (source code for a recent version of the former can be found on [1] under the `nptl` folder).

[1] http://ftp.gnu.org/gnu/glibc/glibc-2.24.tar.gz

3.7.3. *Restrictions to the usage of TLS*

TLS has two main restrictions, which are discussed here:

- number of keys is not unlimited, so there is a maximum number of keys which may be used at the same time. Although this number is high (1024 on the last versions of the `libpthread`), it is limited.
- stored values must be of type void *, and for this reason entities such as *structs* nor *vectors* can be stored. However, what can be stored are the pointers to those structures, although this requires a little bit more work from a thread perspective, because memory for the related entities must be allocated.

  And much more important: if a pointer to (for example) a *struct* is stored this means that at some point the memory for the structure has been allocated independently at each *thread*. Upon finishing the *thread,* it has to be remembered to invoke `free` with this allocated memory (otherwise a memory leak would be incurred).

  Regarding previous paragraph, fortunately the API offers a way to manage allocated memory stored as a value inside the TLS.

3.7.4. *The API for TLS*

This section will outline and describe the API offered by the `pthreads` library related to TLS functionalities:

```
int pthread_key_create(pthread_key_t *key, void (*destr_function) (void
*))
```

Allocates a key for further usage in the rest of functions. Its parameters are:

- **`key`**: a pointer to a `pthread_key_t` variable. The variable does not need to be initialized previous to this function invocation.

- **destr_function**: Address of a function that accepts a void * argument. If the supplied value for **destr_function** is not null, when a thread ends its execution, **destr_function** will be calle d with the value for the key as argument.

Returns 0 on success. Otherwise an error occurred.

Note that this function must be invoked only once for every key the application uses. In order to help with this, the pthread_once function can be used.

The purpose to have **destr_function** is aiding in the case described on the section ***Restrictions to the usage of TLS*** where a thread allocates memory in order to store a pointer to the allocated memory. **destr_function** simplifies freeing that allocated memory. Otherwise, *threads* would have to take care by themselves of freeing that memory (which, if the *threads'* lifecycle was complex would be not trivial to do). If the value of **destr_function** is NULL, it is not invoked.

```
int pthread_key_delete(pthread_key_t key)
```

Deallocates a previously allocated key. Its parameters are:

- **key**: a variable whose pointer has been previously supplied during an invocation to pthread_key_create.

Returns 0 on success. Otherwise an error occurred.

Deallocating a key does not invoke the **destr_function** associated with that key during the invocation to pthread_key_create.

```
int pthread_setspecific(pthread_key_t key, const void *pointer)
```

Stores the supplied value for the given key. Its parameters are:

- **key**: a variable whose pointer has been previously supplied during an invocation to pthread_key_create.
- **pointer**: the value to store for the key. Remember only pointers can be stored.

Returns 0 on success. Otherwise an error occurred.

```
void * pthread_getspecific(pthread_key_t key)
```

Obtains a previously stored value for the key. Its parameters are:

- **key**: a variable whose pointer has been previously supplied during an invocation to pthread_key_create.

Returns the currently stored value for the key. If the key was not previously allocated or a value is not yet stored, returns NULL.

HOME: Exercise 1 – about pthread_once

During description of the pthread_key_create function it's been said that pthread_once function may be useful:

- what does this function?

- how can it be useful related to calling `pthread_key_create` from the *threads*?

3.7.5. GCC altrerntive to TLS API

Although usage for TLS is not complicated once the concept is understood, GCC compiler offers a much simpler interface to partially obtain the same functionality, although there is no support for value destructors.

The reader of this guide is encourages to take a look at https://gcc.gnu.org/onlinedocs/gcc-3.3.1/gcc/Thread-Local.html for an overview of the functionality.

3.7.6. Example code 1: usage of a global variable

Here follows an example code which features:

- code uses a thread to do some arbitrary and non-deterministic processing
- in order to quantify the non-determinism, a counter global variable is used
- once the processing finishes, the global variable is printed

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <time.h>

int counter; // a global var

void sub_work_function ()
{
    // Just iterate a random number of times and increase counter
    int loop_iterations = (rand () % 99) + 1;

    while (loop_iterations--)
        counter++;
}

int work_function ()
{
    // Just iterate a random number of times and increase counter
    int loop_iterations = (rand () % 99) + 1;

    while (loop_iterations--)
    {
        counter ++;
        sub_work_function ();
    }
}

void* pthread_func (void* arg)
{
    // Reset global var
    counter = 0;

    // Invoke a dummy processing function
```

```
    work_function ();

    // And print global var value
    printf ("thread 1 iterated %d times\n", counter);

    return NULL;
}

int main (void)
{
    pthread_t thread_id;

    srand((unsigned) time(NULL));

    pthread_create (&thread_id, NULL, pthread_func, NULL);

    pthread_join (thread_id, NULL);

    return 0;
}
```

A brief analysis of the code:

- in the main function, a seed for the random number generator is set through the use of function `srand`
- `work_function` and `sub_work_function` represent some arbitrary processing, where non-determinism is introduced by doing a random number of iterations
- to avoid the `main` function return before the *thread* finishes, a `pthread_join` is done to await for the *thread* termination

Although this code may seem very simple, and one may think that the counter variable could be passed as an `int*` to `work_function` and `sub_work_function`, the truth is that the processing functions could be really complex accepting a large number of arguments, and adding extra arguments to them just to implement a counter could not be a good coding practice, because of the complexity overhead in the future maintenance of the functions.

HOME: Exercise 2 – about the `counter` global variable

a) At the above code, the `counter` variable has been used as a global storage for the *worker thread*:

- what would happen with this variable if more a total of two *worker threads* were created without modifying the code used by the threads?

b) A solution could be (if there were a total of two worker threads instead of one), to create another variable, but answer this questions about its implications:

- if two global variables were used (one for each thread), would it be necessary to modify the code for the functions `work_function` and `sub_work_function` so that they behave different depending on the thread? If yes, how would you modify them(illustrate the idea without coding, although you can use some pseudocode to show it)?

c) Now imagine that more worker threads are needed, and that the number of total needed worker threads is unknown beforehand:

- could the solution proposed at **b)** adapt to this new scenario? What modifications should be done in order to support this new scenario (illustrate the idea without coding, although you can use some pseudocode to show it)? Which do you think it would be the most difficult technical part if the proposed modifications were to be implemented?

3.7.7. Example code 2: usage of TLS

Here follows an example code which illustrates:

- code uses two threads which do some calculation
- both threads store the result of its calculation inside a TLS variable and then the threads print the result for the calculation
- **important!** the code used by both threads to do the calculation and store the result is the same

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <time.h>
#include <string.h>

// key used for TLS
pthread_key_t tls_key;

// some costant operators to do the calculation
const int operand_a = 10;
const int operand_b = 20;

// an enum used to add diversity to the program
typedef enum {
    multiplication,
    addition
} e_operation;

// this structure will be used as the value for the TLS key
typedef struct {
    char operation [100];
    int result_value;
} s_result;

// a little helper function
void fatal_error (char* message)
{
    printf ("fatal error: %s\n", message);
    exit (1);
}

void store_result (e_operation operation, int result_value)
{
    // allocate memory for the result
    s_result* new_result = (s_result*) malloc (sizeof (s_result));
```

```
    // print a message related to the operation
    // done and store it with the result
    sprintf (new_result->operation,
             "operation is %s",
             (operation == multiplication) ? "multiplication" : "addition");

    // store the result value with the result
    new_result->result_value = result_value;

    // and assign the value for the TLS key
    if (pthread_setspecific (tls_key, new_result))
        fatal_error ("setting value for TLS key");
}

void thread_do_calculation (e_operation operation)
{
    int result_value;

    printf ("doing calulation...\n");

    // simulate that the calculation takes some time
    sleep (2);

    // and do the calculation depending on the argument supplied
    if (operation == multiplication)
        result_value = operand_a * operand_b;
    else
        result_value = operand_a + operand_b;

    printf ("done calculation!\n");

    // finally store the result
    store_result (operation, result_value);
}

int thread_print_result ()
{
    // obtain the value for the TLS key
    s_result *result_ptr = (s_result*) pthread_getspecific (tls_key);

    // just a secutiry check
    if (result_ptr == NULL)
        return;

    // and print the result on screen
    printf ("%s, result = %d\n",
            result_ptr->operation,
            result_ptr->result_value);
}

void* pthread_func (void* arg)
{
    // Obtain the operation to be done from the thread argument
    e_operation* operation_ptr = (e_operation*) arg;

    // Do the calculation and store the result
    thread_do_calculation (*operation_ptr);
```

```
    // And print the result
    thread_print_result ();

    return NULL;
}

void free_result (void* ptr)
{
    // as memory for the result was allocated using 'malloc',
    // 'free' has to be used to free the memory
    free (ptr);
}

int main (void)
{
    pthread_t thread_id_1, thread_id_2;

    // Create the TLS key
    if (pthread_key_create (&tls_key, free_result))
        fatal_error ("allocating TLS key!\n");

    // Create one thread which multiplies
    e_operation operation_for_thread_a = multiplication;
    pthread_create (&thread_id_1, NULL, pthread_func,
&operation_for_thread_a);

    // Create another thread which does the addition
    e_operation operation_for_thread_b = addition;
    pthread_create (&thread_id_2, NULL, pthread_func,
&operation_for_thread_b);

    // And wait for the threads to finish before
    // exiting the main program
    pthread_join (thread_id_1, NULL);
    pthread_join (thread_id_2, NULL);

    return 0;
}
```

A brief analysis of the code:

- the way to pass an argument to a *thread* is by using the last argument of the function `pthread_create`, which during thread creation is passed as argument to `pthread_func` function. As these arguments are of type `void*`, a pointer to the value is passed (instead of the value itself casted to `void*`, which would trigger some compilation warnings), which is dereferenced at `pthread_func` in order to get its value.
- as allocation for TLS keys has to be done only once for each key, `pthread_key_create` is invoked before any thread is created, although the `pthread_once` function could be used if the key was allocated inside `pthread_func`.
- at `store_result` function, it can be seen that a pointer returned by `malloc` is set as a value for the key. This is because values which can be stored for keys are always of type `void*`, and not of any other type. If the result was a simple value (for example an `int`), it could be stored as a value via a cast, but this would in any case trigger a compiler warning.

- at `thread_do_calculation` function two things are done:
    - a delay of 2 seconds invoking the `sleep` function. This is done to simulate a lengthy operation and to ensure the threads overlap its execution.
    - the operation performed depends on an argument to the function. Casually the number of available operations coincides with the number of threads, but don't confuse this situation with that of the proposed at exercise **2.b)**. There could be an arbitrary number of threads using the same code, because the `if` sentence at function `thread_do_calculation` not related need to store the value depending on the thread, but to offer a little diversity to the program.
- in order to simplify the program, both `operand_a` and `operand_b` are constants (they have the `const` modifier and have a fixed value). But the program could be extended to pass different values for the operands to function `thread_do_calculation`.
- there is a helper function, `fatal_error`, which simply prints a message and terminates execution of the program in the case of some non-recoverable error is found.

TLS usage in the previous program can be seen as a state holder for the threads of the program, where different functions (`store_result` and `thread_print_result`) have access to the state of the program (respectively for modifying and consulting the state), without the need to propagate parameters between them.

This exercise is manly a showcase for TLS, to have a ready-to-go example which can be consulted while doing the next exercises.

3.7.8. Exercises

Sometimes, a single-threaded application has to be converted to multithread (sometimes to take advantage of multiple cores found in modern processors, sometimes to integrate simple processing applications into a more complex application which does different processing in parallel and has one main thread coordinating the separate processing).

While doing this kind of work, it's important to do as few changes as possible to original in order to minimise possible side effects during the transition to multithread.

LAB: Exercise 3 – extending *Example code 1*

This is the only practical exercise of this guide. Through it, the code from *Example code 1* will be adapted to support an arbitrary number of threads. For the resolution of this exercise, code from *Example code 2* can be used as a reference.

a) modify the code from Example code 1 so that a TLS key is created. This key can be created in the main function, prior to creating any thread.

b) create the code at the beginning of `pthread_func` so that it stores a zero (NULL) value for the key. Also, remove the code which does `counter = 0`.

c) create a function called `increment_counter`, which gets the value for the key, increases it and stores it again for the key.

d) modify functions `work_function` and `sub_work_function`, replacing `counter ++` by an invocation to `increment_counter`.

e) finally, at `pthread_func`, modify the `printf` invocation so that it prints the value for the key and eliminate the declaration of the global variable `counter` from the beginning of the program.

Now compile the program, execute it and test that it works well (it should print a different value each time).

f) what parameter have you used as argument to `pthread_key_create` argument? Why?

g) now, modify the program so that it creates 20 threads, sleeping one second after each thread creation.

Execute it again and validate that it works well (the printed value should never exceed 10000).

3.7. DMA Transfers Guide

DMA

pre-requisite: exercises about GIPO + interrupts

pre-requisite: exercises about file drivers

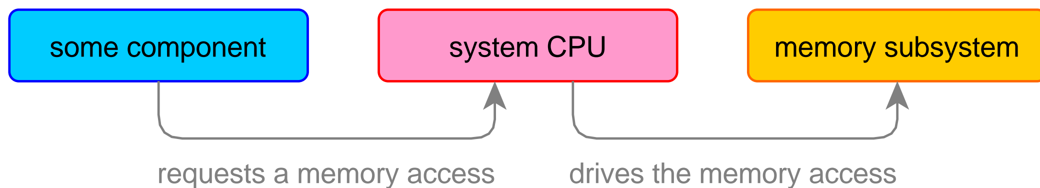
pre-requisite: exercises about blocking I/O

Introduction

A DMA (for **D**irect **M**emory **A**ccess) names a technique in which some system component accesses memory directly, without intervention from the main CPU. This means without code being executed on the CPU in order to achieve the memory access.

Although depending on the architecture used names vary a little, on the Beaglebone the component which is in charge of using the DMA technique is called *DMA Controller*. The reason of this name is because it is in charge of controlling the direct memory access.

Generally speaking, when performing a memory access (either using a *DMA Controller* or not), the following functional situation usually occurs:



Previous diagram is very generic, and illustrates a situation which is agnostic on how the memory access is done:

- some component, may represent a user space program, a device driver or even a hardware peripheral which *notifies* the system CPU when a memory access must be done.
- located on the system CPU, there must be *some code* which can take notice when **some component** is requesting an access and then instructing the CPU to drive the access (which is not necessarily *doing* the actual memory access, but *driving* it).
- memory subsystem part is just a placeholder for some memory external to the processor. It could be a RAM chip, some memory mapped region, etc... It may also contain caches and helper entities (like a DMA Controller).

So, at the end, the diagram only tells that there is **some component** which desires to access the **memory subsystem**, and that some component uses the **system CPU** as a helper agent in order to perform the memory access. The way some component communicates with the CPU takes the form of some **notification mechanism**.

Here are some illustrative examples:

a) there is an *external peripheral* which wishes to copy some data into system RAM

  - *some component* is the external peripheral
  - *some code* on the system CPU is a device driver which implements two things:
    - the receiver part of the *notification mechanism* which could be either based on interrupts or on a polling scheme, and some entity could be a device driver
    - the part which drives the memory access operation
  - *memory subsystem* is composed of two parts:
    - a memory mapped region which points to the device memory
    - a RAM region which will receive the data from the peripheral

b) there is some system *daemon* which wishes to flush buffers due certain user-signal

  - *some component* is the *daemon* program
  - some code on the system CPU is a *signal handler* for the *daemon* implementing:
    - the receiver part of the *notification mechanism* which is the *signal handler* and its registration code
    - the part which drives the memory access operation
  - memory subsystem is composed of two parts:
    - a RAM region which contains the buffers used by the daemon
    - some external memory system where to flush the buffers, like an HDD

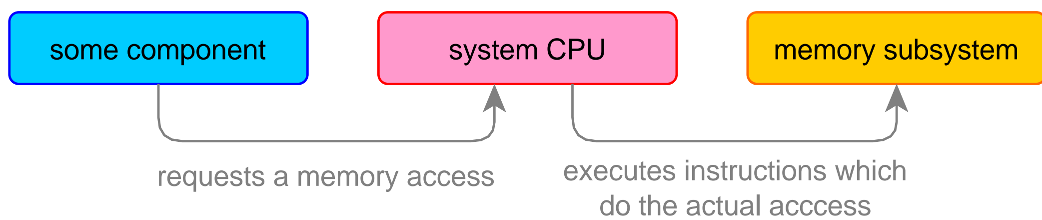
c) a disk file driver, which reads the contents of a file in HDD and stores them into RAM

  - *some component* may be a user space program which invokes the *read* operation
  - *some code* on the system CPU is the file driver code implementing:
    - the receiver part of the *notification mechanism* which are the `sd_open` and related functions' implementation and its registration during the driver initialization
    - the part which drives the memory access operation
  - *memory subsystem* is composed of two parts:
    - the primary HDD, from where to read the file contents

- a RAM region which contains the buffers where to store the file contents

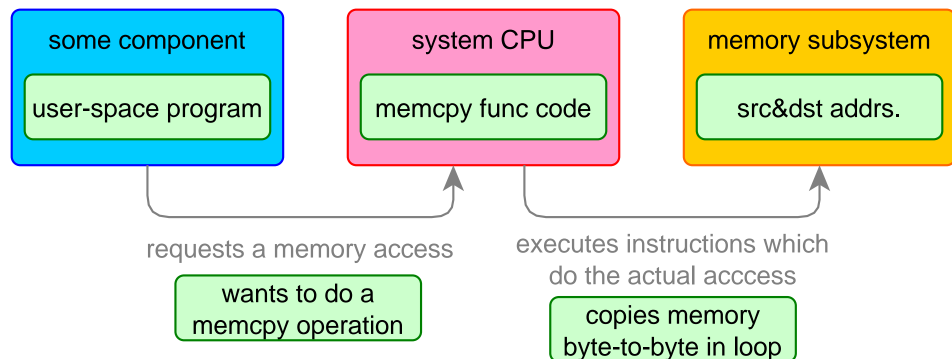
Note that on all previous examples, the *some code*'s part which is in charge of managing the memory accesses is references as “the part which drives the memory access operation”. The reason for this is to remark that this is referred to a situation which is agnostic on how the memory access is done. In order to accomplish this task, *some code* might use some helper entities (like a `syscall`, a `libc` call, a DMA Controller) or it may actually do the memory access (with a loop that reads/writes memory during each iteration).

Without using the DMA technique, this diagram would depict who is in charge of the memory access:



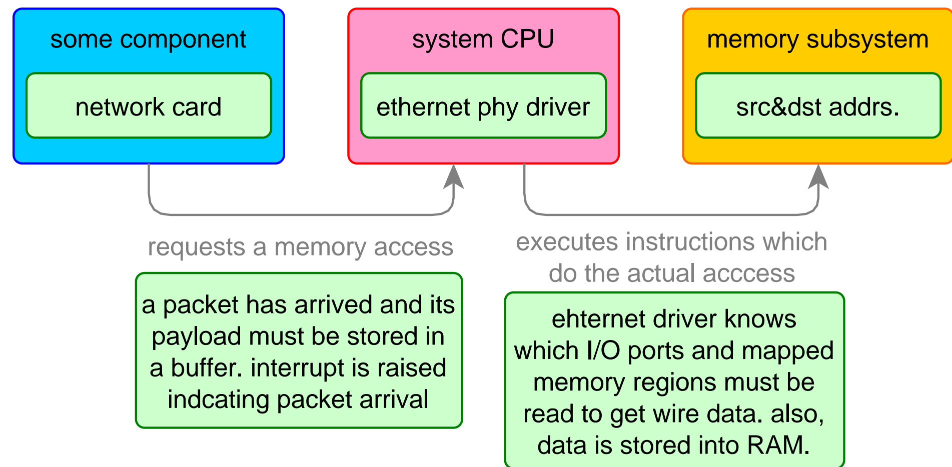
This diagram seems quite similar to the previous one, but in this case the code on the *system CPU* is ***doing*** the memory access.

An example of this situation is the following scenario:



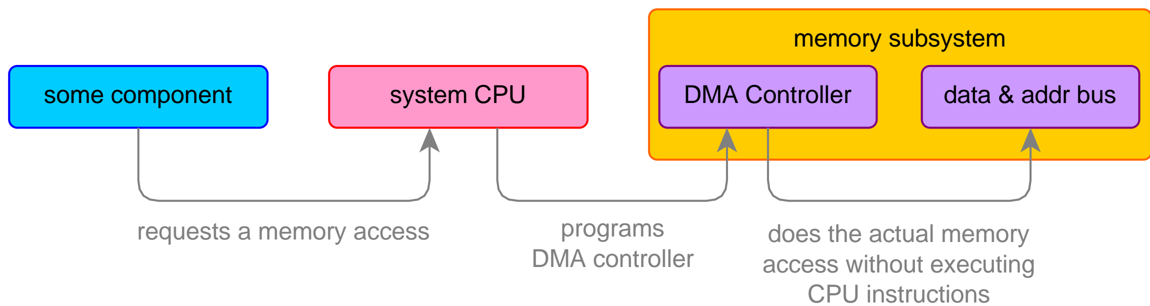
Note how the actual memory access is done by means of *some code* (executed in the processor) which, in this case, is a loop which copies memory byte to byte.

Another example of this situation can be following scenario:



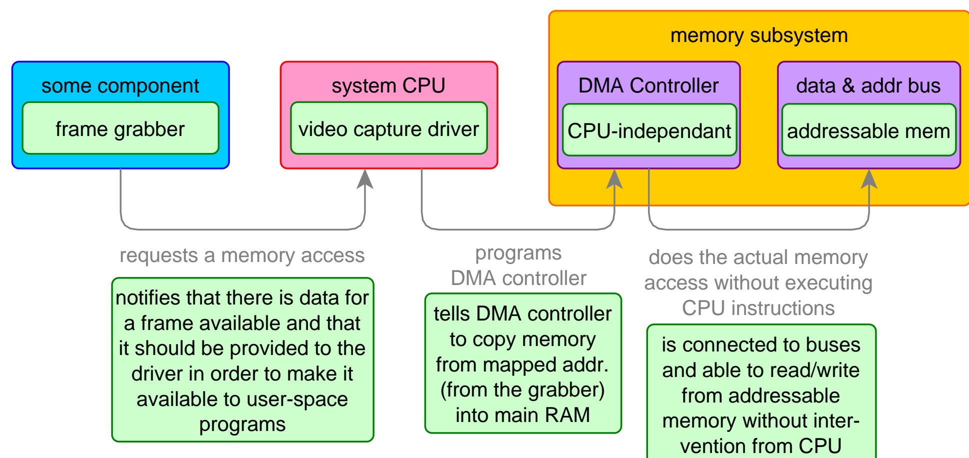
Note how the Ethernet driver code is the one which manages the data accesses. It contains code that reads mapped memory regions and I/O ports and writes some data into RAM.

Using the DMA technique, this diagram would depict who is in charge of the memory access:



Previous diagram is also very generic, but it will useful to explain the role of a DMA controller.

Here follows an example of the DMA Controller in action:



Note how in this case the processor only programs the DMA controller:

- the processor does not do the memory copy operation, but *tells* which operation has to be done to the DMA controller.
- the DMA controller, once configured, performs the memory copy operation on its own, without further intervention from the processor.

Functional overview of the DMA Controller

As it has been mentioned in the previous examples, the DMA Controller main characteristics are:

a) is external to the main CPU, in the sense that they have separate execution flows.

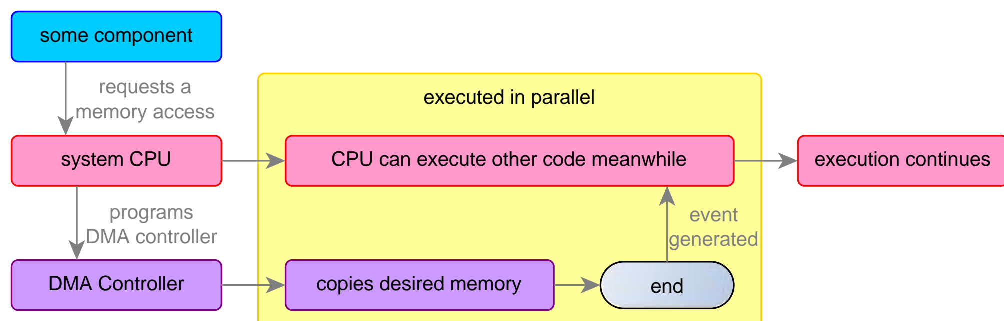
   Despite this and due to the manufacturing process, the DMA Controller may be found either on the same silicon chip as the main CPU (this is the case for the BeagleBone processor) or as a separate chip.

   The fact that the DMA Controller is a separate chip from the main CPU is reflected also in that the DMA Controller usually has some set of dedicated resources which are needed to interact with the main CPU, which may be any combination of:

   - interrupts
   - mapped memory addresses
   - I/O ports

b) its execution is detached from main CPU execution flow: in order for the DMA Controller to access the memory, it must not wait for the processor to execute some code.

   Although this, there are mechanisms to order the transfer start of transfer.



The controller, once it is programmed and start of transfer ordered, has direct access to the data and address bus. Also, usually there exists some way in which the CPU is gets noticed once the transfer ends (*event generated* annotation on the previous diagram).

Don't confuse these two facts:

- the memory transfer operation is done without intervention from the CPU

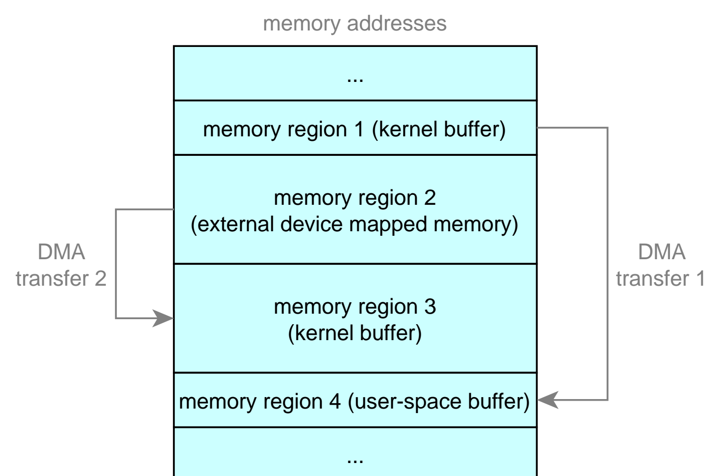
- some code which executes on the CPU must set up the memory transfer operation

Drivers which use a DMA controller usually contain some code related to programming the DMA Controller. The programming usually consists in telling the DMA Controller which is the origin address, the destination address and the amount of bytes to copy.

the transfer operation begins right after both having configured the transfer operation and ordered DMA Controlller to start the transfer. Is the transfer operation which is independent from CPU.

c) copies memory from one address into another: this means that the DMA Controller can access all addressable memory.

Examples of DMA transfers are:



Although the basic functionality of a DMA are memory copy operations, specific systems may make use of the extended bit-wise capabilities of an extended DMA-Controller.

For example, a chroma-key video effect might be implemented with intelligent use of bit-wise operations, where:

i) the destination memory-address is some memory which will be presented to the user in visual form (such as a bitmap).

ii) more than one source addresses could be used by means of more than one chained DMA operation, each of them using different bit-wise operations (i.e. a combination of AND and OR operations)

Advantages/drawbacks: independent execution flow

On the previous section it has been told that the execution flow of the DMA controller is detached from the execution flow of the main processor.

The main advantage of this is that the CPU can be doing some calculation at the same time the DMA Controller is doing I/O. This means that while the DMA Controller is doing memory operations, the processor is free to execute (almost) whatever code it wishes. This is the main reason which led to the invention of DMA Controllers.

The main drawback of this is that for the CPU to get noticed that a transfer has finished an additional mechanism is needed, which can be either:

- by polling, via continuously watching some variable related to the DMA Controller which lets the driver know that the transfer has finished.
- by interrupt, letting the DMA Controller raise an interrupt requested by the driver

Remember the exercises about blocking I/O, where it was explained the difference between these two notification schemes. In most cases, an interrupt scheme is used when dealing with DMA Controllers.

Exercise 1 (HOME): interrupt numbers for DMA

Search on the AMM335x *Technical Reference Manual* and explain:

a) Which is the hardware interrupt number for *Successful end of DMA transfer*?

b) Which is the hardware interrupt number for *Unsuccessful end of DMA transfer*?

Advantages/drawbacks: external to the main CPU

On the BeagleBone, although the DMA Controller lies on the same physical chip as the CPU and is electrically connected to it, functionally they are treated as separated entities. What this means is that there are no processor instructions directly related to programming the DMA Controller.

Instead, the way the CPU has to interact with the DMA Controller is through use of the following mechanisms which are supported by the CPU:

- memory mapped regions, composed of slotted memory regions which contain the configuration for the transfer operations among the available DMA Channels (more on this later on this document)
- events, which trigger the transfer operations
- interrupts, that provide support for the asynchronous notification mechanism to the CPU when the transfer ends or there is an error

One advantage of such a separation from the main CPU is that it decouples managing the DMA Controller from the specific processor model. This way, the management the DMA Controller has its own layer of kernel-code which may be (up to some point) independent from the processor itself. For example, the BeagleBone used at the laboratory is an AMM335x model. With careful programming it should be possible to program the DMA Controller and have code which compiles and executes fine on other processor which use the same DMA Controller. More about this paragraph will be seen later.

Advantages/drawbacks: direct access to address and data buses

If the DMA Controller is able to bypass the main CPU when accessing memory is because it is connected to the data and address buses. This way, the DMA Controller can *claim* the usage of the buses and do its operations, although this has some implications, which are the basis for the main advantages and drawbacks that must be taken into account when dealing with DMA Controllers.

Its advantages are:

- the CPU is not needed to access the memory, and thus it is free to execute some other code during the transfer operation.

  This is THE advantage of using a DMA Controller.

Its drawbacks are:

- the DMA Controller accesses physical addresses, which on systems that have a MMU, may imply some problems.

  An MMU (among other things) takes care of a memory mapping process. A primary parameter for this mapping process is the page/frame size.

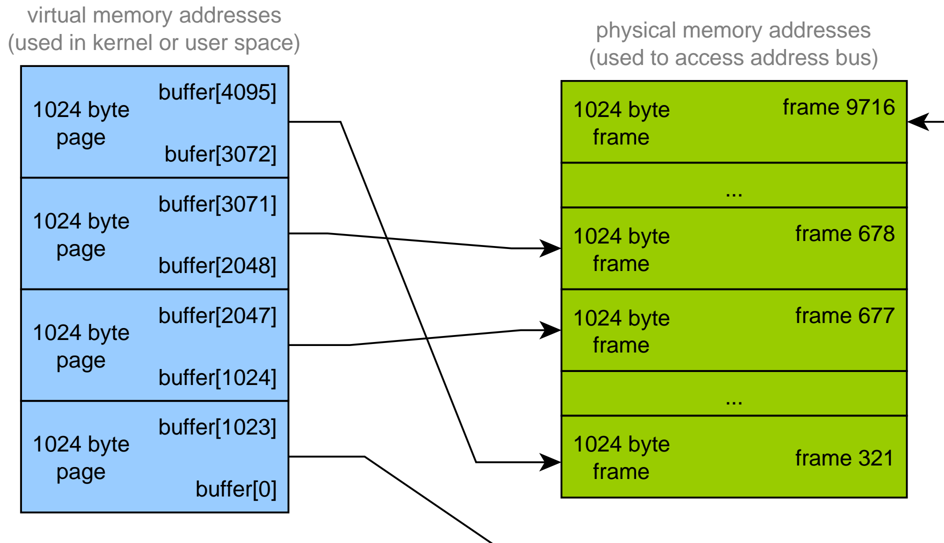
  Broadly speaking, a page (which is a word used on systems that support virtual memory) is the minimum chunk of RAM memory manageable by the system, so that when physical memory is allocated from the OS, it is done in chunks whose size is a multiple of the page size.

  The page refers to the memory in the virtual memory space, and the frame refers to the physical memory. Page and frame size are equal.

  The processor uses an internal lookup table to map between pages (virtual memory addresses) and frames (physical memory addresses).

  The reason for the importance of the page (or frame) size is depicted on the following diagram which shows a random allocation scheme for a buffer. In this case:

  - buffer size is 4096 bytes
  - page/frame size is 1024 bytes



In the previous diagram some facts can be appreciated:

- when a buffer is allocated in user or kernel-space, a contiguous logical block of memory is returned, so that pointers can be used to access the memory in a linear fashion.

  For example, following code would sum the values for all the bytes on the buffer:

  ```
  char* p = buffer;
  int i = 0, sum = 0;
  for (i = 0; i < 4096; i++, p++)
  {
     sum += *p;
  }
  ```

  It can be seen that the buffer contents can be accessed in an incremental way.

  This also allows for example to execute the `memcpy` function to copy several megabytes at once.

  The previous is true even given the fact that the buffer contents are distributed among different memory pages.

- the real layout of the buffer in the physical memory is different.

  Each virtual memory page maps to a physical frame, but as a lookup table is used for the mapping, there is nothing which prevents two contiguous pages (in virtual memory) to belong to two discontinuous frames (in physical memory).

  The mapping process (may be seen as if it) is arbitrary (and hardly predictable) from a user-space program or a kernel driver point of view.

Now let's return to the DMA Controller behaviour. As it's been said, the DMA Controller is directly connected to the data bus, so it uses PHYSICAL addresses.

Now recapitulate some usages for the DMA Controller:

- **copy memory from an external device into a kernel-driver (paged) buffer**:

  Suppose that buffer size is 4 KB and page size is 1 KB.

  The DMA Controller basic transfer operation takes as inputs the following parameters:

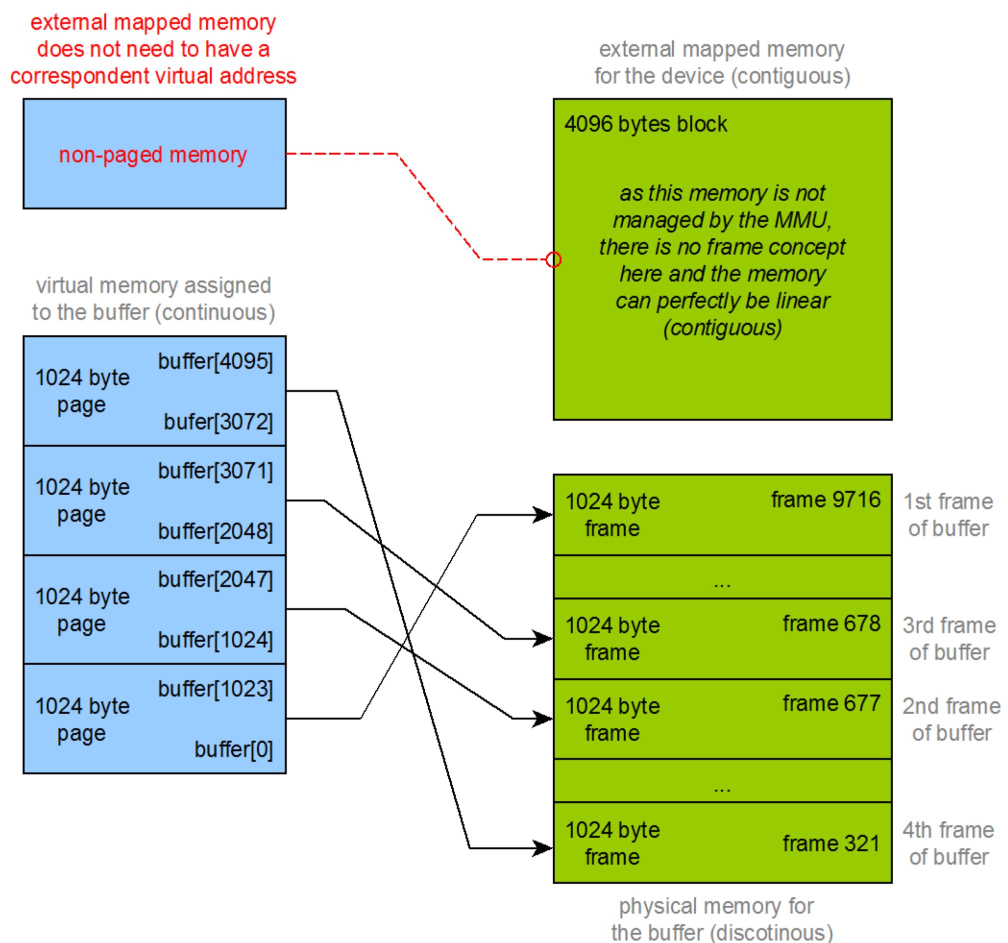
  - source physical address
  - destination physical address
  - transfer size in bytes

  Suppose that buffer size is 4 KB and page size is 1 KB.

  Memory from the external device may correspond to a mapped memory region which is 4KB in size, so in this case the device offers 4 KB of contiguous memory. The reason for the memory being contiguous here is that the device is directly connected to some data bus, and is able to read the address from the address bus, acting in such a way that the addresses are contiguous.

  The problem is the kernel-driver buffer. As the buffer physical memory is split across different frames, it may be physically stored at some not contiguous frames.

  Following diagram illustrates the situation:



What this diagram is telling is that:

- in order to access the external memory through the address bus, once the starting physical address is known and given a length (in bytes), anyone with access to the bus can simply access any position inside the external memory region adding the desired offset to the start address

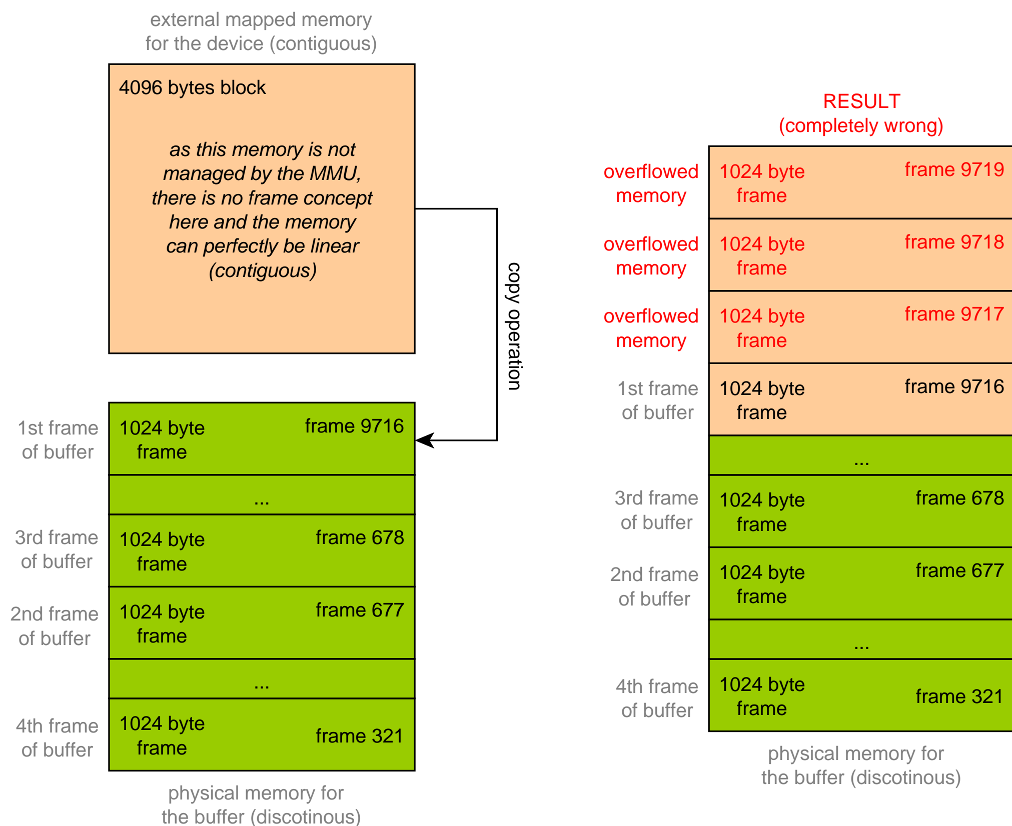
- in order to access the buffer memory, one has to know that the memory is fragmented across different frames, so for example knowing which address place into the bus to access the byte corresponding to `buffer [3540]` is not trivial and of course using a strategy like `*(ptr_to_start_of_1st_frame_buffer + 3540)` is completely wrong

Remember that the basic transfer operation of the DMA Controller takes three arguments: source + destination physical addresses + transfer size in bytes.

If (following the previous example) a transfer of 4096 bytes is desired, it is very tempting to provide the following parameters to the DMA Controller:

- source physical address: base address of the external mapped memory
- destination physical address: base address of the 1st buffer frame
- transfer size: 4096 bytes

This is what it would happen:



What happened here is that the memory transfer operation did not take into account the paged nature of destination buffer.

The first page is copied correctly, but only the 1024 bytes (which corresponds to the size of the first page/frame). Remaining bytes (4096 minus 1024) would be copied into physical RAM in wrong frames. So the result would be a memory overflow. The three frames which are above the 1st one, which don't belong to the buffer, are being written.

- **copy memory from an external device into another external device**:

  In this case, if both devices map the memory in a contiguous way, it would be safe to use a transfer operation with the size up to the memory size which the external device has available.

  In this concrete case, as the MMU of the CPU is not managing any of the involved physical memory, neither source nor destination memory is spread among different pages.

  This kind of copy operation (from external device to external device) would be safe if the devices which own the memory addresses expose those addresses in such a way that they're contiguous.

- **copy memory between kernel-driver (paged) buffers**:

  The problem here is the same as in the first case, but aggravated.

  This kind of copy operation copies memory between buffers which are both contained (source and destination) along different pages (and thus, very probably, non-contiguous frames).

  So the problem is duplicated in this case:

  - as in the first case, it is not possible to write into sequential addresses starting from the 1st destination frame base address, because bytes not belonging to the destination buffer (those which exceed the size of one frame) would be written

  - similarly, it is not possible to read from sequential addresses starting at the 1st source frame because bytes not belonging to the source buffer (those which exceed the size of one frame) would be read

Does this mean that a DMA Controller cannot be used to copy memory from/to a paged buffer where the size of the transfer exceeds the size of one page?

Not really, in fact the main problem with DMA transfers comes from the need of consecutive memory addresses in order for the DMA Controller to behave as expected, so two very simple solutions will be presented here:

- **solution 1**: applicable when a buffer is either the source or destination of a DMA transfer, if that buffer splits among more than one frame. Instead of doing one

single DMA transfer for the desired amount of bytes, split it into multiple DMA transfers, each of them operating on a single frame of the buffer.

- **solution 2**: applicable when the size of the buffer at maximum a few megabytes in length. Instead of allocating the memory for the kernel-buffer using the `kalloc` function, use another family of allocator functions which ensure that the buffer is reserved always along consecutive frames.

This is THE inconvenient of using a DMA Controller.

- the DMA Controller accesses the data bus directly, so it must be taken into account that the CPU memory cache could be bypassed.
- The data/addresses bus are shared among all devices which use it, so contention may be perceived if two or more devices try to access the buses at the same time.

  For example, if DMA Controller is doing a transfer operation and meanwhile the processor needs to access memory, either one of two things may happen:

  - one device must wait while the other device has access to the buses
  - both accesses are interleaved and thus each device perceives a slower access

  Previous effect can be mitigated by some caches connected to the processor, by avoiding the processor for the need to access the physical RAM and thus avoiding it to use the buses which are connected to RAM.

Exercise 2 (HOME): page/frame size

Find out how can it be obtained the page/frame size on the BeagleBone and:

- tell which command has to be used in order to get the parameter
- obtain the value for the parameter

Limited number of resources for the DMA Controller

Every device/controller found on an embedded system is composed by a finite set of resources.

And the same goes for the DMA Controller, which uses a resource called *DMA Channels*.

The number of available *DMA channels* indicates the number of simultaneous transfer operations that the DMA Controller can have configured at the same time.

So when a driver wants to do a DMA transfer, it must:

- during transfer initialization, (maybe during driver *load* function), request a *DMA Channel* which will get reserved for use by the driver

- when there is need to do a DMA transfer, use the *DMA Channel* returned on the previous point for the transfer
- during transfer shutdown, (when the driver is sure no more DMA transfers will be done, e.g. on the driver *unload* function), release the *DMA Chanel* so that it can be recycled by other drivers

This way or requesting-using-releasing follows the same schema as in the case of using some IRQ / GPIO / memory buffer from the driver.

Just as any driver which uses limited system resources, the usage of the DMA Controller implies that the driver must be responsible for requesting access to resources, using only resources for which it has been granted access and releasing the resources after they are not needed anymore.

Exercise 3 (HOME): about DMA Channels and PaRAM sets

Search on the AMM335x *Technical Reference Manual* and explain:

a) How many DMA Channels are available on the AM335X processor?

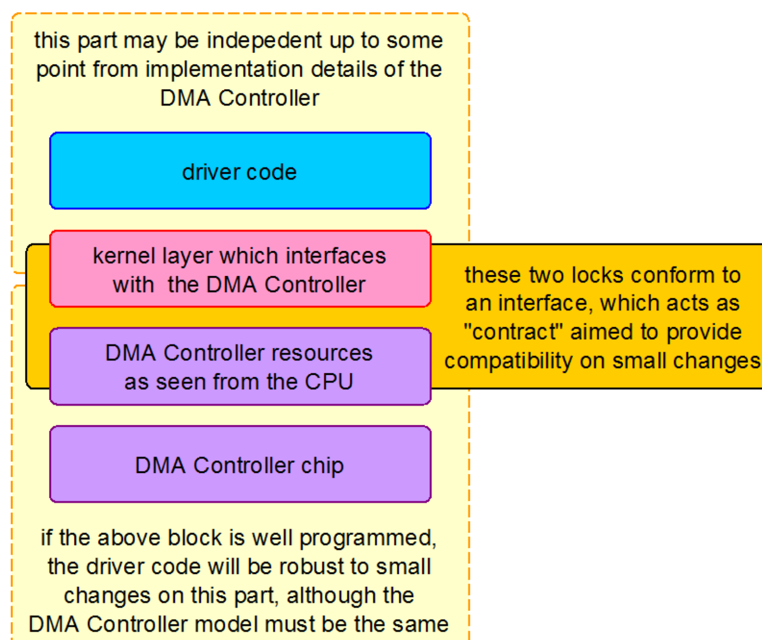
b) What is a PaRAM set?

c) How many PaRAM sets are available on the AM335X processor?

d) How are PaRAM sets and DMA channels related?

Linux Kernel DMA layered architecture

As it has been said, the Linux Kernel provides a layered architecture related to managing DMA transfers.



Just as always when programming some hardware, one may wonder why it is good to have a layered architecture.

As a general rule, one has to take into account that the Linux Kernel has a codebase which has evolved during decades, and usually most aspects of the Kernel have evolved to meet practical issues found during usage of its functionalities.

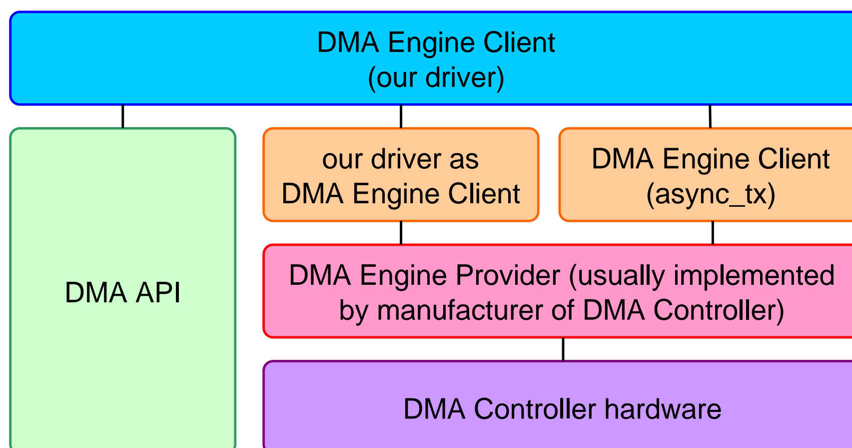
The previous been said, having a layered design for the DMA (like as with any other hardware) allows the kernel to:

- arbitrate access to the limited resources
- provide an abstraction layer, which allows a more standardized way to access the hardware
- isolate the driver writers against small changes, and (up to some point) provide the ability to adapt to evolutions on the real hardware specifications

Linux divides the DMA implementation on Kernel through three primary components and one optional component:

- DMA API, which provides auxiliary functions for users of DMA functionalities
- DMA Engine Client (slave) API, which offers an API to drivers that use DMA operations
- DMA Engine Provider (controller) API, which offers an API with which low-level drivers that interact with the DMA Controller must implement in order to enable the DMA Engine on the system
- `async_tx` (optional), provides a simplified API which uses the DMA in order to do memory operations in an asynchronous way

Following diagram illustrates dependencies between these components.



Some comments can be made on the previous diagram:

- DMA Engine Provider is code which usually is created by the manufacturer of the DMA Controller and which controls low-level operations on it (for example, setting values on the memory mapped regions which control the DMA Controller). This component provides (hence the name *provider*) low-level functionalities to the DMA Engine Clients.
- DMA Engine Client is code which uses (hence the name *client*) functionalities provided by the DMA Engine. Whenever a component wishes to use DMA Operations on systems which have the DMA Engine enabled, should use the DMA Engine Client API.
- DMA API is code which is orthogonal to all the previous code. It provides auxiliary functions which are convenient to use when dealing with DMA transfers but which are generic to any DMA hardware. For example, special memory allocators.
- `async_tx` has two sides: on one side it is a DMA Engine Client, so it uses functions of the DMA Engine in order to make DMA transfers; on the other side it also provides a simplified API to let drivers make use of the simplified DMA transfers it provides.
- our driver may use functions provided by any of the previous components. A wise choose of functions to be used from each component is up to the driver programmer.

Linux Kernel DMA layered architecture fies

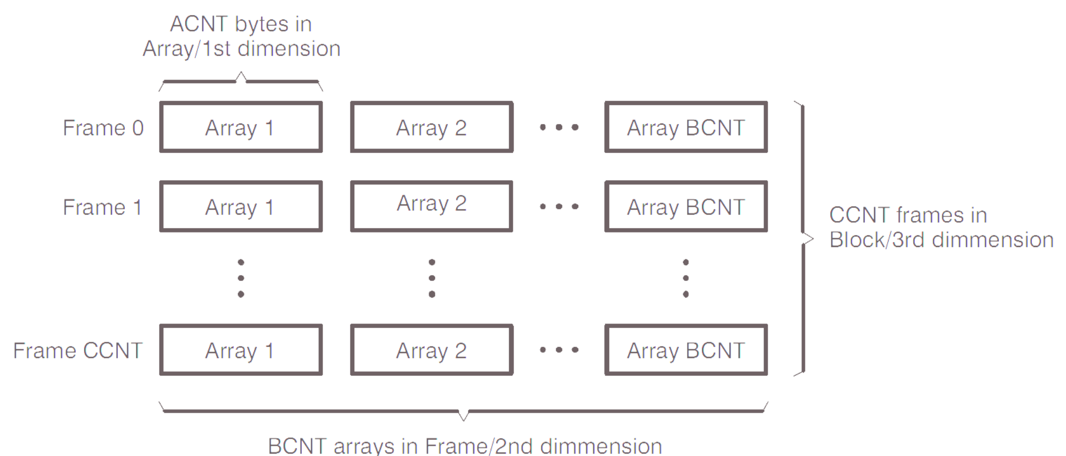
What comes next may serve as reference when further investigating the code where all the components from the previous section.

EDMA3 Controller

The DMA Controller which comes bundled into the AM335x processor (the BeagleBone CPU) is called EDMA3 (which stands for **E**nhanced **DMA**, revision **3**).

Special features of this DMA Controller are:

- transfers can be configured in order to act on several dimensions:



Previous diagram shows how actually BCNT x CCNT transfers are done, being each transfer the copy of ACNT bytes

It is allowed to specify all parameters which define the values for ACNT, BCNT and CCNT, and also (for each dimension) distance between the arrays

- transfers can be triggered in different ways: due to external event, manually triggered and due to the end of another transfer (this allows chaining transfers)
- end-of-transfer can generate an interrupt: either on successful transfer or on error
- the only supported access operation is copying memory: no extended operations (bitwise ones) are supported by this controller. Despite of this, controller allows some tuning on how the source and destination indexes are updated during the DMA transfer
- multiple sequential transfers can be automatically done one after the another: using either *Linking* or *Chaining*

Exercise 4 (HOME): about DMA Channels and PaRAM sets

Search on the AMM335x *Technical Reference Manual* and explain:

a) What is the difference between *Linking* and *Chaining*?

b) Find the definition of a scatter-gather operation related to DMA transfers. How can it be implemented with the EDMA3 Controller?

Example code for DMA transfer

Writing a code which performs a DMA transfer for the BeagleBone may be tricky and there is little information available on how to do it apart from some practical code at the Texas Instruments Forums.

Here is a working example of a file driver which does a DMA transfer from the GPMC bus to the driver space, using blocking I/O:

```
// Needed by the driver
#include <linux/module.h>
#include <linux/init.h>
#include <linux/errno.h>

#define MODULE_TRACE KERN_ALERT "dma-module: "

// Needed by the file driver
#include <linux/fs.h>
#include <linux/miscdevice.h>

static int sd_open (struct inode* pInode, struct file* pFile );
static int sd_release (struct inode* pInode, struct file* pFile );
static ssize_t sd_write (struct file *filp, const char __user *buffer,
                size_t count, loff_t *f_pos);
static ssize_t sd_read (struct file *filp, char __user *buffer,
                size_t count, loff_t *f_pos);
static loff_t sd_lseek (struct file *filp, loff_t off, int whence);
```

```
static struct file_operations sd_fops =
{
    write:      sd_write,
    read:       sd_read,
    open:       sd_open,
    release:    sd_release,
    llseek:     sd_lseek,
};

static struct miscdevice sd_devs =
{
    minor:      MISC_DYNAMIC_MINOR,
    name:       "dma_module_file",
    fops:       &sd_fops
};

// Needed for DMA-related function
#include <linux/dma-mapping.h>
#include <linux/platform_data/edma.h>

static volatile int transfer_complete_control = 0;
static void dma_complete_callback (unsigned lch, u16 ch_status, void *data);

static int edma3_dma (int acnt, int bcnt, int ccnt);

// Kernel-space buffer
static dma_addr_t dmaphysdest1 = 0;
static char *dmabufdest1 = NULL;

#define STATIC_SHIFT    3
#define TCINTEN_SHIFT   20
#define ITCINTEN_SHIFT  21
#define TCCHEN_SHIFT    22
#define ITCCHEN_SHIFT   23

#define MAX_DMA_TRANSFER_IN_BYTES 32768

// Base of external GPMC memory
#define EXTMEM_BASE 0x09000000

// *************************
// Here begins program source
// *************************
static int sd_open (struct inode* pInode, struct file* pFile ) {
    return 0;
}

static int sd_release (struct inode* pInode, struct file* pFile ) {
    return 0;
}

static ssize_t sd_read (struct file *filp, char __user *buffer,
                size_t count, loff_t *f_pos)
{
    // Do a 32k transfer
    edma3_dma (MAX_DMA_TRANSFER_IN_BYTES, 1, 1);

    return count;
}
```

```
static ssize_t sd_write (struct file *filp, const char __user *buffer,
                size_t count, loff_t *f_pos)
{
    return count;
}

loff_t sd_lseek (struct file *filp, loff_t off, int whence)
{
    return 0;
}


// DMA-related function implementation
static int edma3_dma (int acnt, int bcnt, int ccnt)
{
    int result = 0;
    unsigned int dma_ch = 0;
    int i;
    unsigned int numenabled = 0;
    unsigned int BRCnt = 0;
    int srcbidx = 0;
    int desbidx = 0;
    int srccidx = 0;
    int descidx = 0;
    struct edmacc_param param_set;

    /* Set B count reload as B count. */
    BRCnt = bcnt;

    /* Setting up the SRC/DES Index */
    srcbidx = acnt;
    desbidx = acnt;

    /* A Sync Transfer Mode */
    srccidx = acnt;
    descidx = acnt;

    if ((result = edma_alloc_channel (EDMA_CHANNEL_ANY, dma_complete_callback,
NULL, 0)) < 0)
        return result;

    dma_ch = result;
    edma_set_src (dma_ch, (unsigned long)(EXTMEM_BASE), INCR, W8BIT);
    edma_set_dest (dma_ch, (unsigned long)(dmaphysdest1), INCR, W8BIT);
    edma_set_src_index (dma_ch, srcbidx, srccidx);
    edma_set_dest_index (dma_ch, desbidx, descidx);

    /* A Sync Transfer Mode */
    edma_set_transfer_params (dma_ch, acnt, bcnt, ccnt, BRCnt, ASYNC);

    /* Enable the Interrupts on Channel 1 */
    edma_read_slot (dma_ch, &param_set);
    param_set.opt |= (1 << ITCINTEN_SHIFT);
    param_set.opt |= (1 << TCINTEN_SHIFT);
    param_set.opt |= EDMA_TCC(EDMA_CHAN_SLOT(dma_ch));
    edma_write_slot (dma_ch, &param_set);

    numenabled = bcnt * ccnt;
```

```
    for (i = 0; i < numenabled; i++) {
        transfer_complete_control = 0;

        result = edma_start (dma_ch);

        if (result != 0) {
            printk (MODULE_TRACE KERN_ALERT "edma_start failed\n");
            break;
        }

        while (transfer_complete_control == 0);

        /* Check the status of the completed transfer */
        if (transfer_complete_control < 0)
            break;
    }

    if (0 == result) {
        edma_stop (dma_ch);
        edma_free_channel (dma_ch);
    }

    return result;
}

// DMA delegate ISR
static void dma_complete_callback (
       unsigned lch, u16 ch_status, void *data)
{
    switch (ch_status) {
        case DMA_COMPLETE:
            transfer_complete_control = 1;
            break;
        case DMA_CC_ERROR:
            transfer_complete_control = -1;
            break;
        default:
            break;
    }
}

// Placeholder functions which should be replace by working ones
void pointers_init (void) { }
void pointers_deinit (void) { }
int gpmc_init (void) { return 0; }

static int __init dma_module_init (void)
{
    int result = 0;

    printk (MODULE_TRACE KERN_ALERT "Loading module\n");

    pointers_init ();
    gpmc_init ();

    if (misc_register (&sd_devs))
    {
```

```
        printk   (MODULE_TRACE   KERN_ALERT   "Misc   file   driver   cannot   be
registered\n");
                return (-ENODEV);
    }

    printk (MODULE_TRACE KERN_ALERT "Misc file driver registered\n");

    dmabufdest1 = dma_alloc_coherent (NULL, MAX_DMA_TRANSFER_IN_BYTES,
                                            &dmaphysdest1, 0);

    if (!dmabufdest1) {
        printk (MODULE_TRACE KERN_ALERT "Dma alloc coherent failed\n");
        return -ENOMEM;
    }

    printk (MODULE_TRACE KERN_ALERT "Module loaded\n");

    return result;
}

static void dma_module_exit (void)
{
    misc_deregister (&sd_devs);

    dma_free_coherent (NULL, MAX_DMA_TRANSFER_IN_BYTES, dmabufdest1,
                        dmaphysdest1);

    printk (MODULE_TRACE KERN_ALERT "Module unloaded\n");
}

module_init (dma_module_init);
module_exit (dma_module_exit);

MODULE_LICENSE("GPL");
MODULE_VERSION ("1.0");
```

Note that functions used for previous driver in order to trigger the DMA are neither DMA Engine functions nor `async_tx` functions, but the Manufacturer (Texas Instruments) functions (all of them prefixed with `edma_*`). This has been done to show all the implied configuration parameters in a DMA transfer using the EDMA3 Controller.

Also note that previous code uses placeholder function bodies for following functions:

```
void pointers_init (void) { }
void pointers_deinit (void) { }
int gpmc_init (void) { return 0; }
```

These functions must be replaced with the proper ones developed at the DSX course.

Two things are important in this example driver:

- memory allocation, using `dma_alloc_coherent` function during driver load.

  Using this function solves one of the problems when memory buffers are used for DMA transfers: it is guaranteed that the allocated memory uses a congruous block of physical memory.

This solves the problem with physical frames not being contiguous in memory, and buffers allocated this way are eligible to participate in a DMA transfer operation.

Unfortunately, this function is limited to small buffers (32 KB in the example), because the Kernel needs to allocate a continuous physical address range for the allocated memory size so it's possible that, if abused, this function invocations begin to fail.

Note how this function returns a virtual-address buffer and a physical address buffer (this is the reason to pass an argument as a pointer, because the function will modify the pointed value in order to be able to return the physical address).

- the DMA transfer function, which is split in different stages

First, the transfer is configured, starting with allocating a channel:

```
edma_alloc_channel (EDMA_CHANNEL_ANY, dma_complete_callback, NULL,
0)
```

Note that the channel is linked to a *callback* function. The ISR in charge of listening to DMA-related interrupts will act as a router that notifies the proper callback registered for each channel.

In the invocation to allocate a channel, any channel is good and the *callback* is this one:

```
// DMA delegate ISR
static void dma_complete_callback (
       unsigned lch, u16 ch_status, void *data)
{
    switch (ch_status) {
        case DMA_COMPLETE:
            transfer_complete_control = 1;
            break;
        case DMA_CC_ERROR:
            transfer_complete_control = -1;
            break;
        default:
            break;
    }
}
```

The only function of the callback is to set a *status flag*.

Then, the transfer is configured

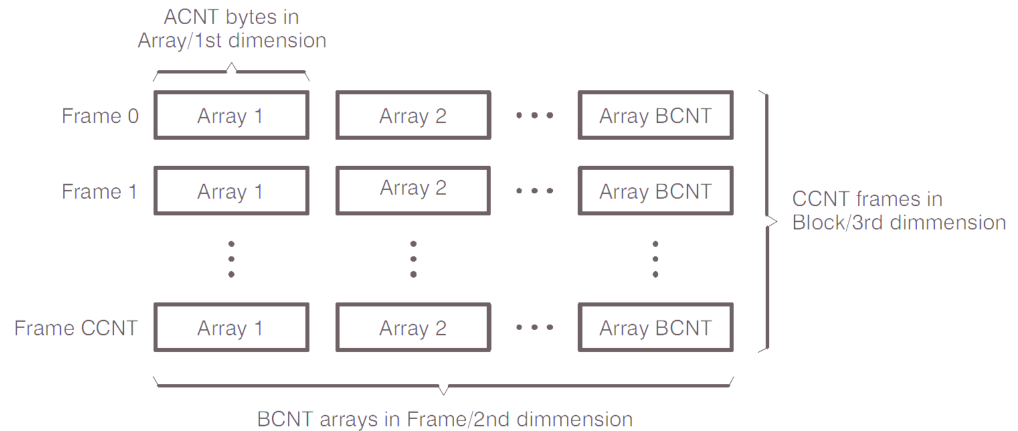
```
edma_set_src (dma_ch, (unsigned long)(EXTMEM_BASE), INCR, W8BIT);
edma_set_dest   (dma_ch,   (unsigned   long)(dmaphysdest1),   INCR,
W8BIT);
```

Now how, for the transfer to succeed, PHYSYCAL addresses are used.

Virtual addresses are very convenient almost in all cases, but is this case keep in mind that the DMA Controller has direct access to the data bus, so it uses physical addresses.

Also, not how the operation desired for both destination and source is `INCR`. Modifying this would enable doing a *scatter and gather* operation.

Remember this diagram?



This is what `srcbidx desbidx srccidx descidx acnt bcnt ccnt` variables control. In this case, the transfer only uses 1st dimension (ACNT), so all of them but `acnt` are set to 1.

Next, the transfer parameters are set

```
/* A Sync Transfer Mode */
edma_set_transfer_params (
    dma_ch, acnt, bcnt, ccnt, BRCnt, ASYNC);

/* Enable the Interrupts on Channel 1 */
edma_read_slot (dma_ch, &param_set);
param_set.opt |= (1 << ITCINTEN_SHIFT);
param_set.opt |= (1 << TCINTEN_SHIFT);
param_set.opt |= EDMA_TCC(EDMA_CHAN_SLOT(dma_ch));
edma_write_slot (dma_ch, &param_set);
```

Most important part of previous code is that the transfer is configured as asynchronous (`ASYNC` flag of `edma_set_transfer_params`).

Apart from this, interrupts are enabled for the DMA Channel.

And finally, the transfer is started

Note how the DMA transfer process is asynchronous:

```
transfer_complete_control = 0;

result = edma_start (dma_ch);

if (result != 0) {
    printk (MODULE_TRACE KERN_ALERT "edma_start failed\n");
    break;
}

while (transfer_complete_control == 0);
```

The DMA Transfer will block the driver execution until it’s completed:

- At first, status flag is set to 0

  When the DMA Transfer finishes, the end-of-transfer callback will modify this flag. As the callback is executed from a ISR context, the flag is marked with the `volatile` keyword.

- Then, the DMA Transfer is stared. As it is an asynchronous operation, the function will return immediately. It is driver responsibility to wait for the transfer to complete.

- The wait loop symbol waits until the transfer has ended (either successfully or with errors)

Exercise 4 (LAB): the transfer is a bad citizen

Now, remember (at the Guide about blocking I/O) how was a *bad blocking behaviour* defined.

This is just what happens with previous code. The wait done there keeps consuming CPU cycles.

So, using a wait queue (a semaphore) change this behaviour so that:

- the process is put to sleep while *waiting* for the DMA transfer to complete

- the *callback* code, in addition to updating the status flag must now awake the process

It is not needed to implement it as an *interruptible sleep*.

Previous exercise allowed the process to behave well, but as the transfer only copies 32 KB of memory the timing is almost negligible.

Exercise 5 (LAB): adding an interruptible wait

To tweak the process, do following modifications.

- modify the driver so, once the `sd_read` function is invoked, the program is put to *sleep* in an *interruptible* way

    - if the sleep is interrupted, return the value –`ERESTARTSYS` from `sd_read`

- the condition to wake from the *interruptible sleep* and continue execution (i.e. starting the DMA transfer) will be that the push button from the BeagleBone is pressed. This will require:

    - creating another wait queue

    - registering an ISR for the GIPO 20, whose callback will wake up the second wait queue

    - modifying `sd_read` function to add the *interruptible sleep*

4. Conclusions

4.1. The path to the guides

Elaborating the guides presented on this work has been an enrichening (yet sometimes hard) experience.

Lots of concepts have been exposed and reviewed through this work, most of them being completely unknown to me before starting this work.

I've been working as a professional programmer for some time now (programming over IBM/AIX, which is a UNIX variant), and never have had the need to learn the concepts exposed here.

I'm personally very happy with the knowledge gained during the process, and it would be awesome if some other student took this guide and follow the steps of this path.

4.2. Retrospect about Kernel code digging

Kernel programming is not one the easiest subjects to face. It's easy to enter a hair-pulling state when one or more of these things happen:

- Kernel compilation fails due to a recently introduced patch which has not been properly tested
- kernel functions return unexpected error codes, and there are no google results for the error
- documentation for some kernel-related aspects is missing and there is no other choice than dig through code
- system crashes when a driver is loaded
- a driver works well but suddenly stops working

Kernel source code is well structured, but it also:

- has passed through hands of thousands of people, so there is not a unique coding style there
- has a complex build system, and sometimes it's hard to know which Kernel configuration options are preventing the build to succeed
- has more than 35 MLOC (million lines of code) and almost 80k code files, so (although it's structured in directories) it's not obvious where to search for further information.

- man-pages for the Kernel ignore implementation details of the specified functions, so sometimes it's not obvious the inner-working of certain functionalities, which can only be known by looking at the functions' source codes
- make use of advanced compiler features (such as variadic macros, function modifiers, custom idiomatic macros, … and almost every C keyword and construct)

As usually happens, previous statements can be taken as a show-stopper or as a challenge awarded with knowledge about one of the most fascinating and portable open-source projects of history.

5. Bibliography

5.1. *Introduction bibliography*

- www.gnu.org. “Homepage”. Accessed during spring and summer 2016. Online at www.gnu.org
- google. “Google forums, first message published by Linus Torvalds mentioning Linux Project”. Online at https://groups.google.com/forum/#!msg/comp.os.minix/dlNtH7RRrGA/SwRavCzVE7gJ
- G. Keizer. “Windows comes up third in OS clash two years early”. Online at http://www.computerworld.com/article/3050931/microsoft-windows/windows-comes-up-third-in-os-clash-two-years-early.html
- www.top500.org. “Linux statistics: development over time categorized by OS Family”. Online at https://www.top500.org/statistics/overtime/
- gs.statcounter.com. “Top 8 operating systems on July 2016”. Online at http://gs.statcounter.com/#all-os-ww-monthly-201607-201607-bar
- G. Keizer. “Windows comes up third in OS clash two years early”. Online at http://www.computerworld.com/article/3050931/microsoft-windows/windows-comes-up-third-in-os-clash-two-years-early.html
- opensource.org. “Open Source Licenses by Name”. Online at https://opensource.org/licenses/alphabetical
- L. Torvalds. “Commit activity”. Github. https://github.com/torvalds/linux/graphs/commit-activity
- L. Clark. “The Top 10 Developers and Companies Contributing to the Linux Kernel in 2015-2016”. Linux.com. Online at https://www.linux.com/blog/top-10-developers-and-companies-contributing-linux-kernel-2015-2016
- dd-wrt.com. Homepage. Online at http://www.dd-wrt.com/site/index
- Guifi.net. “dd-gufi Firmware Project homepage”. Online at https://guifi.net/en/project/dd-guifi
- V. Wagner. “The Flying Penguin: Linux In-Flight Entertainment Systems”. Linux Insider. Online at http://www.linuxinsider.com/story/The-Flying-Penguin-Linux-In-Flight-Entertainment-Systems-65541.html
- Scientific Linux. “Where is Scientific Linux Running?”. Online at https://www.scientificlinux.org/about/where-is-scientific-linux-running/
- E. Brown. “Linux Leads Self-Driving Car Movement”. Linux.com. Online at https://www.linux.com/news/linux-leads-self-driving-car-movement

- BeagleBoard.org. “BeagleBone & BeagleBone Black Capes”. Online at http://beagleboard.org/cape

5.2. Mounting a bootable Ubuntu/Linux image bibliography

- R. Pérez López. "Design of a dedicated cape board for an embedded system lab course using BeagleBone". M.S. thesis, Electrical Engineering department, Technical University of Catalonia (UPC), Barcelona, Spain, 2014
- R. González Muñoz. " Improving the BeagleBone board with embedded Ubuntu, enhanced GPMC driver and Python for communication and graphical prototypes". M.S. thesis, Electrical Engineering department, Technical University of Catalonia (UPC), Barcelona, Spain, 2015

5.3. Guides bibliography

- J. Corbet, G. Kroah-Hartman, A. Rubini. “Linux Device Drivers, 3rd Edition”, 2005. Online at http://www.makelinux.net/ldd3/
- R. Love. “Linux Kernel Development, 2nd Edition”, 2005. Online at http://www.makelinux.net/books/lkd2/
- Texas Instruments. “Forum on Sitara™ processors”, accessed during spring 2016. Online at https://e2e.ti.com/support/arm/sitara_arm/
- R. C. Nelson. “Linux Kernel source code 3.8.13 patches for AMM 33x”. Last accessed September 2016. Online at https://github.com/RobertCNelson/bb-kernel/tree/am33x-v3.8
- www.kernel.org. “Linux Kernel Man Pages”. Accessed during spring and summer 2016. https://www.kernel.org/doc/man-pages/
- www.kernel.org. “Linux Kernel Documentation”. Accessed during spring and summer 2016. https://www.kernel.org/doc/Documentation/