Politecnico di Torino

Master of Science in Mechatronic Engineering

Design and Implementation of a Touchless Machine Interface using Embedded Linux

SUPERVISOR
Prof. Massimo Violante

CANDIDATE
Fabio Garcea

Academic year 2017-2018

I dedicate this thesis to

my family, my girlfriend Vittoria and my friends.
Your love is the real author of this document.

Abstract

The following document will pass through the several steps that brought to the production of an embedded system able to interpret hand movements, a human gesture interface. This project has born as a thesis matter of study and eventually evolved into the real implementation of a complex electronic device; the following chapters represents a walk through the choices and methodologies that brought a scratched design to an object, from the customization of the Operating System to the development of an application.

Acknowledgements

Firstly, I would like to express my sincere gratitude to my advisor Prof. Massimo Violante for having constantly provided me with material and support. A second thank goes to the colleagues from Hodierna S.R.L. and EMA S.R.L. for having shared their skills with me. At last a special thank goes to Vittoria, Chiara, Pise, Rudy, Dave, Guarna and all my friends for their feedbacks and suggestions.

List of Figures

Contents

1 Introduction

The main purpose of this thesis is to show each of the design and development phases that brought to the production of an embedded system for human gesture recognition.

The human gesture controller developed by Microchip, the MGC3130, is able to estimate the position of a moving point in a 3D space and to understand some of most common gestures that can be made with the hand in order to control an action during an application. This is possible thanks to the presence of both a controller and a panel hosting five electrodes positioned as the cardinal points plus the center of the panel.

This technology can offer a new way to send a control within an application; the possible fields of interest for such an interface could be for example the home automation which could exploit the capability of this interface to understand the gestures in order to make easier the remote control of our electronic devices, or the video gaming industry that could use it as a new kind of controller or embeds it as an extension of the classical joysticks. Moreover it could be used to control a moving rover or to swap the menus of our car infotainment while driving.
With a noise reduction study and the design of a different electrode-panel shape able to improve the precision of the position tracking, this controller could be used to control robotics arms during a precision application which could require a direct human control.

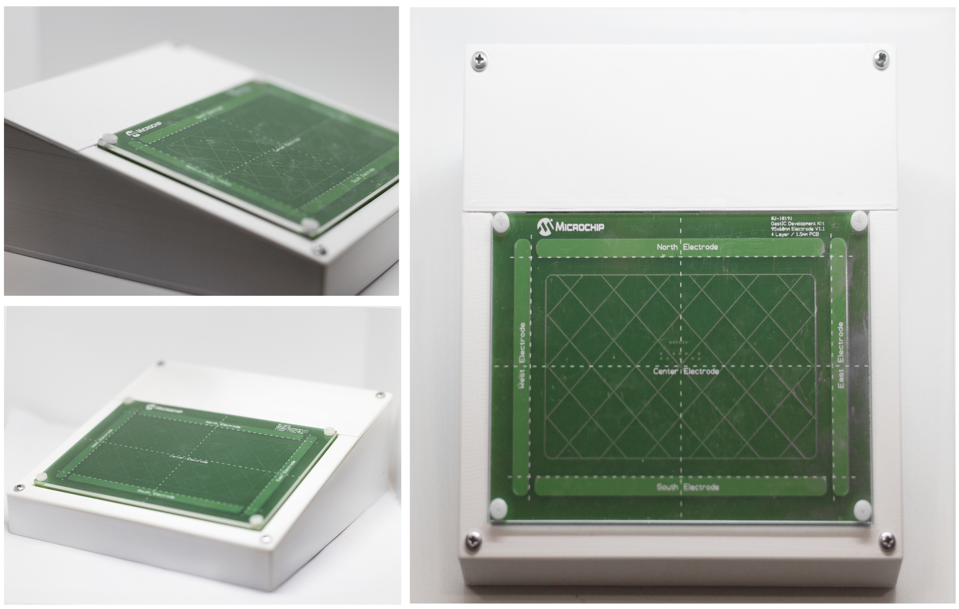


Figure 1: Packaging produced for the GestIC interface

The gesture recognition is a field in computer science whose goal is to study the human gestures and represent them using mathematical function; the gesture recognition feature may in fact become the natural successor of the touch-based technology. A well suited gesture interface may one day make redundant the presence of any hardware that nowadays is used for controlling an electronic system; this was basically the main reason that brought to the birth of this project.

Moreover a lot of companies are interested in the production and research of new gesture recognition sensors like:

- Intel;
- Microchip;
- Qualcomm;

Since gesture recognition is a really actual topic new technologies are constantly produced in order to find always a better solution for this purpose; the different projects that implement this feature may thus be divided according to deployment application and the tools used to realize the interface.

Historically the first technology able to actuate gesture recognition has been the wired gloves technology; these devices also known as "data gloves" basically are electronic gesture interfaces that can be worn as common gloves. They are structured as a complex system of sensors whose output can be studied to recognize a certain movement made with the hand; made of tactile sensors for the fingers and bend sensors to catch finger bending movements, the gloves offer a full gesture based control.

The first emerged prototype of data gloves is the Sayre Glove produced by MIT researchers and was based on the use of LEDs and flexible tubes; the idea was to sense, by using photodiods, any light variations on the tubes and estimating the bending moment received by the hand.

These kind of gloves can be defined as *active* since they can send direct informations through the installed sensors. Passive gloves were, on the other hand, made as a complex system of colored regions whose movements could be sense by a camera and eventually interpreted as gesture. The first prototypes evolution brought to the emission non the market of the first models of data gloves used for virtual reality as the Acceleglove, again from MIT researchers and developers.

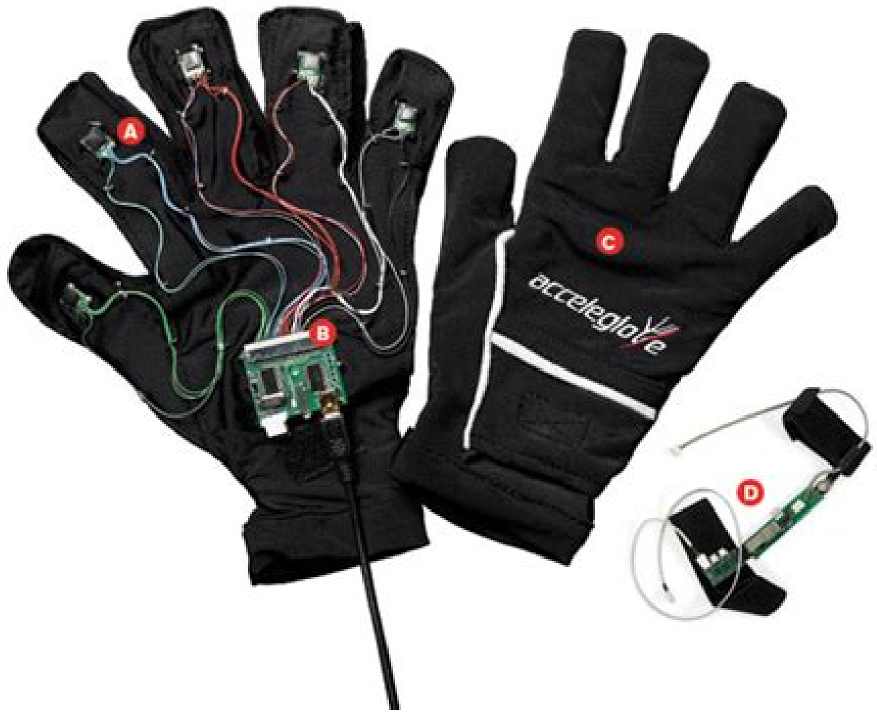


Figure 2: Acceleglove data gloves by MIT

As pointed out there are several strategies that could be adopted to recognize a gesture; one of them is the stereo camera vision. By using two camera whose relationship in terms of distance is well known it is possible to model a 3D space in which the gesture can be performed. By comparing the object position with a reference position (that could be for instance implemented as a source of light) the motion and consequently the gesture can be estimated. One of the probably most known implementation of this strategy can be found in the video gaming industry, the *Kinect* controller produced by Microsoft for Xbox 360 and Xbox One video gaming consoles. This device exploit both an RGB camera and a depth sensor to provide full-body 3D motion capture; this interface is also capable to sense movements of several players at the same time.

Figure 3: Kinect controller by Microsoft

The third possibility, that is the adopted one, is to sense and recognize gestures through gesture controllers and to process them through firmware or software. This is the strategy adopted by companies specialized on gesture recognition as Gestigon and uSens companies.

During the project each of the development phases of this embedded system was touched. Made an exception for the panel shaping problem and the design of the controller, that has been bought as it is from Microchip, the human interface controller modeled during the project has been created starting from the scratch; from the customized OS stack creation to the graphical interface, from the hardware choice to the packaging the steps occurred during the project were the following:

1. the choice of the hardware;
2. the development of a custom OS stack to run the application;
3. the design of a driver for the device;
4. the customization of an existing API to interact with the embedded system and the development of an application running on the device;
5. the design of the structure allowing the data exchange with an host PC;
6. the design of a packaging for the controller.

The system in his complex can be modeled as follows:

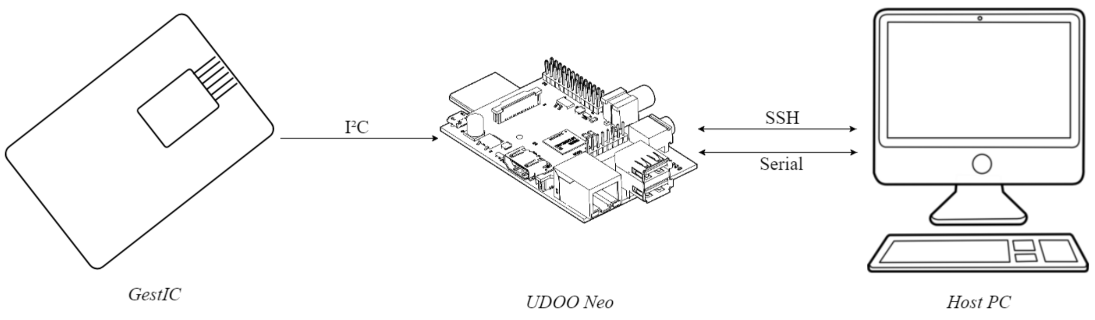


Figure 4: Overhaul system structure

The three main objects within the overhaul system of this projects are, as can be seen by the graphical model, the *GestIC* controller, a *UDOO Neo* board and a *Host PC*.

The *GestIC* controller represents the sensor interfacing to the external world and thus the entry point for the hand position raw data. It is made by two parts, the *electrode-panel* and the *MGC3130* controller.

The *UDOO Neo* board is the brain of the system as it hosts both the driver for the *GestIC* controller and the application exchanging the data with the *Host PC* as far as the *API* through which the signal coming from the sensor are interpreted.

As last the *Host PC* acts in this project as the Master of the chain; in fact it was used during the first phases to create the OS running on the *UDOO Neo* board as well as the application used to retrieve and send the data and, in the last phases, it represents the client within the system since it is in charge to starts the application remotely and visualize the received data in a graphical manner.

2 Linux Embedded

2.1 Linux Embedded for Embedded Systems

Even if Linux was historically born as a "general purpose operating system" (aka *GPOS*) and thus intended to animate PC hardware with Intel x86 architecture, the introduction in the market of low cost reliable flash memory devices (as *SD cards*) and some of the useful features offered by this operating system allowed him to grow in popularity also in the Embedded field of production.
Some of those features can be summarized:

1. his **File system** structure, labeling it as a *file oriented* operating system, fits very well those embedded application where the device hosting Linux has some mass storage components;

2. his capability of support **multi-tasking** applications;

3. his **high modularity** makes possible to import and export source code easily thus making it a very **highly reconfigurable** OS, which is a very useful feature in both the general purpose and in the embedded fields;

4. it is an Open Source project and thanks to his nature it brings a huge documentation and a active community support;

5. as it is a **Real-Time** operating system it can provide a deterministic behavior during such applications.

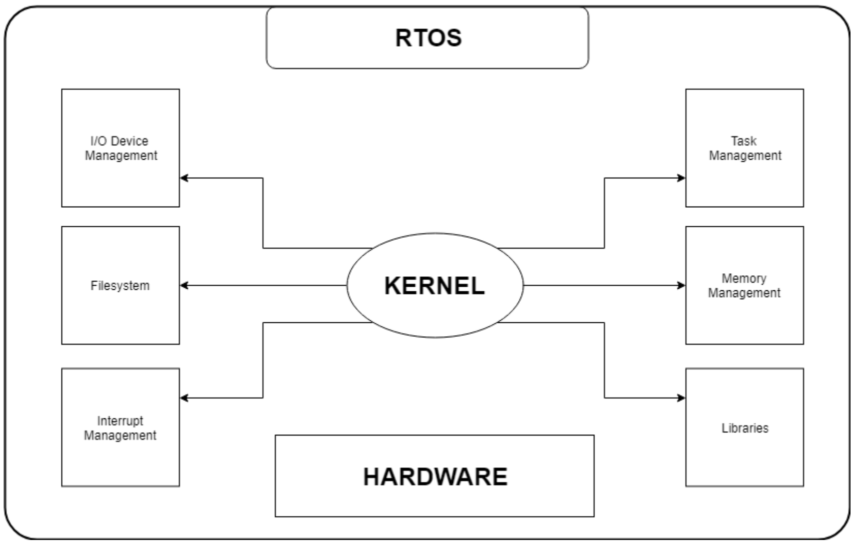


Figure 5: Real-Time operating systems basic structure

In order to exploit this last feature i.e. the possibility to handle inter-processes communication easily and to manage data exchange during a real-time activity, Linux fits well those applications where the device hosting the operating system is provided with a **MMU** (Memory Management Unit) to ensure data locking features through memory mapping for task running concurrently.

2.2 Linux Generalities

From the earliest phases of the project the development of a *custom* Linux Embedded distribution was necessary, since it is useful to introduce the three main components that are at the core of such an application.

The **Bootloader** is the first part of software running on the system and it is in charge of initialize the hardware and load the Operating System from a non-volatile memory to the RAM memory during the so called *Bootstrap* phase.

There are several structures in which a Bootloader can be designed: the easiest one is the *single-stage* Bootloader, a single file which can initialize the hardware needed by the system and at the same time run the Kernel. This kind of implementation is the easiest possible but it has a great limitation in the Bootloader size; in fact this measure has to be limited, typically to 512 bytes (as for instance for the x86 architecture). It is thus not always possible to store all the information needed during the bootstrap of a complex system within such a small portion of memory; in those cases the most suitable design for the Bootloader is the *multi-stage* structure.

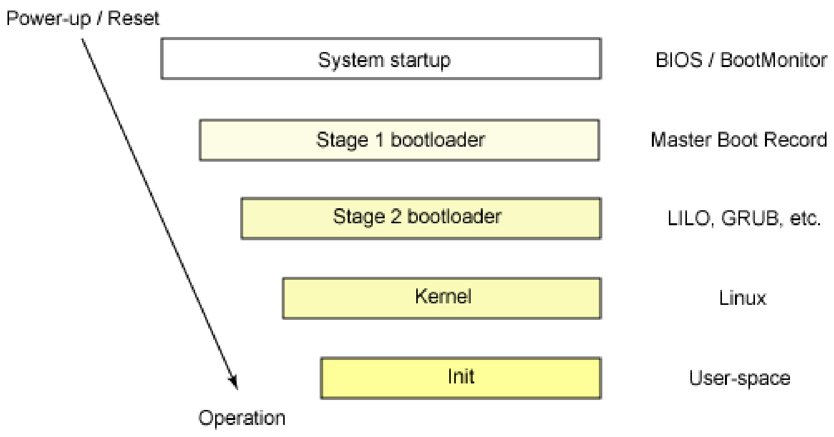


Figure 6: Example of Multi-Stage Bootloader power-up process

The multi-stage Bootloader is based on the concept of the single-stage Bootloader and represents it's natural extension in terms of capabilities and complexity. In this case the Bootloader is basically composed by several stages of Bootloading. During the bootstrap phase the first-stage Bootloader is in charge of loading in the memory the second-stage Bootloader that is significantly more complex than the first and is able to load in memory the Kernel of the Operating System (or even to choose between several Kernels). The famous Linux GRUB is an example of second-stage Bootloader.

The second necessary component of a Linux Embedded operating system is the **Kernel**. The Kernel represents the core of the functionalities of an operating system and is the part of the OS able to access and control each component of the overhaul system. The Kernel is the first software running in RAM memory after the Bootstrap phase which in fact ends by loading this core software in memory; the Kernel is then in charge of terminating the Bootstarp phase and to start running the most basic functions of the system.

The primary function of the Kernel is to manage the usage of the available hardware resources of the system. Those resources are for example the CPU, any I/O devices, the memory; moreover it allows the abstraction of the mass memory as a file system for all the application running on the device. It is thus possible to identify within a Linux system several *layers* or *spaces* that model how the operating system allows any access to the physical resources.

In this structure any User Level application that needs to access some of the hardware resources of the system, as the CPU or the memory, will always do it passing through the Kernel space; this is done by using the so called "System Calls". The Kernel in fact provides to the user space level a set of calls to functionalities as device input/output, inter-process communication, thread management, networking operations and much more; the user space is then able to access the required functionalities only through the system calls. Moreover since those calls are wrapped by the C library, the user space rarely calls them directly but through those wrappers instead.
After a System Call the Kernel executes the required operations on the required resources and brings back the results to the user level passing through the same interface as well.

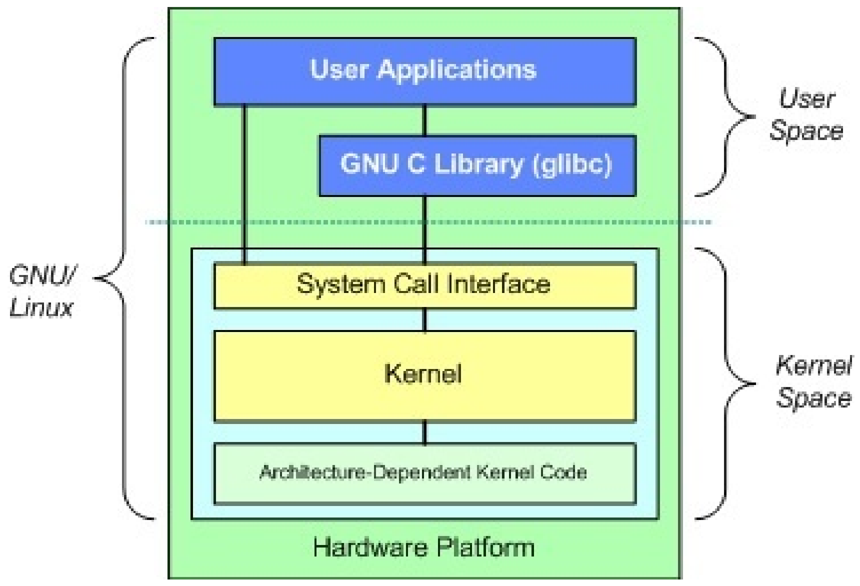


Figure 7: Linux Kernel structure

The structure described so far is the so called *Monolithic Kernel* structure and it is only one of the possible structures used to implement the Kernel.
An operating system can basically be based on three types of Kernel:

- the **Monolithic Kernel**; this kind of Kernel executes all the operations of the operating system in the same address space (the Kernel Space). This choice guarantees an high reliability of the system for any error occurring in the user space level, which in this case would not propagate into the Kernel address space. This choice allows to maintain a single very efficient Kernel but, as a drawback, any operation which needs to pass from the Kernel has to do it through the System Call Interface thus bringing a loss in efficiency due to non-direct access to the resources; moreover a crash of the Kernel would mean the crash of the whole system;

- the **MicroKernel**; this structure moves most of the features that in the previous one were attempted by the Kernel Space to the User Address space. This choice makes the system more flexible and allows a greater customization capability; for instance drivers can be loaded and unloaded at user space and any service causing errors at the user space can simply be restarted thus not causing the crash of the whole system.

- the **Hybryd Kernel**; this kind of Kernel integrates the robustness of monolithic Kernel for most critical processes and resources with the easy maintainability and the customization capability of the Microkernel structure.

In the Embedded field of production the Linux Kernel is often the choice for several reasons:

1. each version of the Kernel is stable and reliable;
2. it can run on each kind of system since it doesn't require any high computational capability;
3. it easy to customize thanks to the wide community support and the extensive documentation;
4. it's modularity features allow to decide weather a component should be or not included, even at run time;

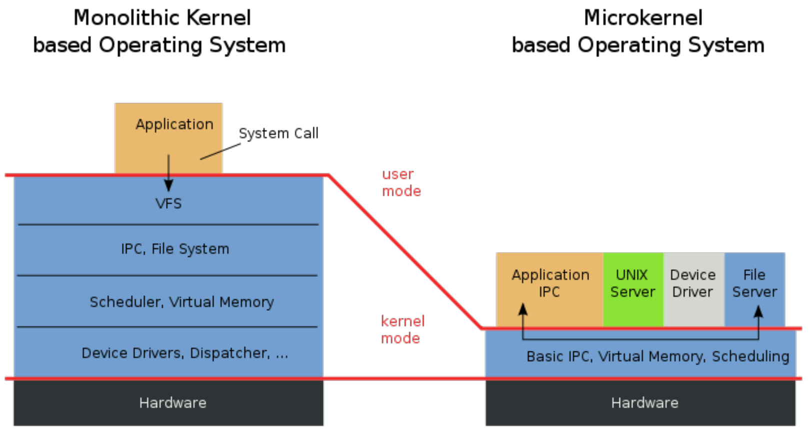

Monolithic Kernel based Operating System
Microkernel based Operating System
Application
System Call
VFS
IPC, File System
Scheduler, Virtual Memory
Device Drivers, Dispatcher, ...
Hardware
user mode
kernel mode
Application IPC
UNIX Server
Device Driver
File Server
Basic IPC, Virtual Memory, Scheduling
Hardware

Figure 8: Comparison between Monolithic Kernel and MicroKernel structures

The last essential part of the Linux OS is the **Device Driver**; since this piece of software has been one the first part of this project being developed, but not actually the first one, it will be explained more extensively in the further sections.

A device driver is basically a program which allows the operating system to communicate with a certain device (a peripheral for instance) by setting up and defining the basic functions to access it.

2.3 Open Embedded, Pyro and Bitbake

During the first part of the project the main need was to find a suitable operating system to be run on the selected board; the choice has eventually been the UDOO Neo which is sold in several packages. In our case it has been used a UDOO Neo Full Board; it is important to point out the board model since this information has been used from the very first steps of the developing process.
The operating system running on this board is a custom version of Linux Embedded well tailored on the hardware of the UDOO Neo Full board and it has been developed using the tools provided by the the Yocto Project.

The Yocto Project is an open source project whose aim is to provide a set of tools to build a custom Linux-based system regardless of the actual hardware architecture.

The latest version of the Yocto Project tools can always be found on the official page of the project and, at the beginning of the thesis wor,k the latest release of the Yocto reference distribution was the **poky-pyro-17.0.1-tar.bz2** version that can be found at `https://www.yoctoproject.org/software-overview/downloads/`. Poky is the Yocto Project reference distribution and the download package contains everything necessary to build a sample Linux Embedded distribution, i.e, the Linux OS *metadata* and the *Open Embedded* tools.
To understand better the contents of the package downloaded it's necessary to explain the most important groups and projects whose synergy is able to give life to the Yocto Project tools for Linux Embedded customization.

The previously mentioned OpenEmbedded is a project, older than the Yocto Project itself, whose main purpose was to develop and maintain the build system necessary to create an Operating system starting from information about "how to build the data" and ending with the production of an assembled package. This Project was born in 2003 but later, in 2005 it was splitted up into another project, whose importance during this work has been crucial: it is the *Bitbake Project.*

Bitbake is the build engine at the core of both the OpenEmbedded build processes and the Yocto Project as well. However since it is a standalone project it can be downloaded and used for different purposes. Bitbake basically is an automation engine that can be used to automatize software building processes, as in this case. In this sense it could be compared to other well known automation engines like **Ant**, but the way it works is quite different.
In order to work properly it is in fact necessary to setup a build environment for the engine before executing the automation process.
While a part of the original project redirected the effort into the new one, the OpenEmbedded Project continued working on *metadata maintenance*; the metadata in the Open Embedded terminology represents the data structure used to manage the software building phases of an operating system creation.

Before moving on the custom Linux distribution some experiments were done on the previously downloaded package. As pointed out the package downloaded from the Yocto Project official website contains almost every source needed to compile a Poky Linux Embedded version, i.e. the Bitbake build engine and the metadata needed to instruct it.
The following software packages were installed, as they are required to proceed through the following steps, on the host PC used during this project which was running the Linux Ubuntu 16.04 version of the Ubuntu OS,at this time an LTS (Long Time Service) version:

- *gawk* needed to use the GNU Awk programming language for an easier handling of data reformatting;
- *wget* used to retrieve files through the most commons Internet protocols as HTTP, HTTPS, FTP;
- *git-core* used to retrieve file from GIT;
- *diffstat* used to apply patches to files;
- *unzip* used to decompress .zip archives;
- *texinfo* which is the official documentation format for the GNU project;
- *gcc_multilib* the collection of compilers for C/C++ of the GNU project;

- *build_essential* needed by the build system to create .debian packages;
- *chrpath* which allows to change the dynamic library load path;
- *socat* used to establish two bidirectional data streams for data transfer;
- *libsdll1.2-dev* useful to compile programs interacting with media features;
- *xterm* a standard terminal program;
- *minicom* a text based terminal emulation program;
- *curl* to transfer data through the URL syntax.

The required packages can be installed by typing this command on the terminal:

```
$ sudo apt-get install gawk wget git-core diffstat unzip texinfo gcc-multilib
build-essential chrpath socat libsdl1.2-dev xterm minicom curl
```

As previously pointed out before moving to the Yocto Project tools some experiment was done on the Open Embedded and the Bitbake tools in order to check if everything was working fine with the Poky Linux reference distribution, the one that in the following chapters is going to be at the base of the definitive Linux OS built for the project.
The archive file can be extracted with

```
$ tar xvfj poky-pyro-17.0.1-tar.bz2
```

The contents of the folder that will be created represent the core of the Open Embedded tools.

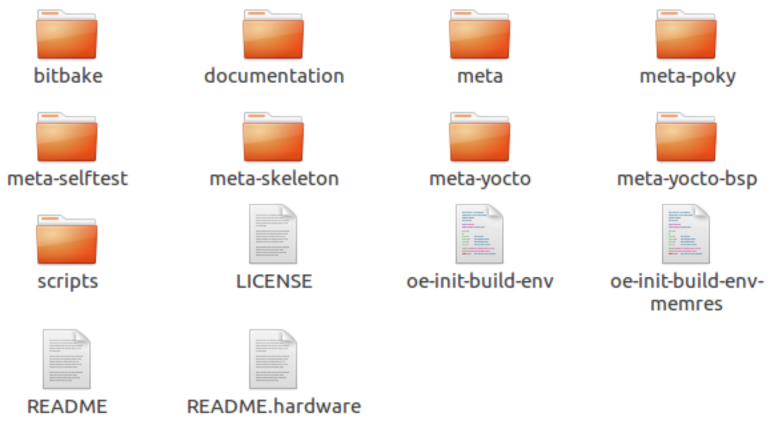


Figure 9: Poky-Pyro distribution package contents

The folder contains, as pointed out before, the Bitbake Engine tool in the form of a self contained folder as far as some documentation folder and the LICENSE which is a mix of MIT and GPLv2 licenses; moreover the remaining folders contain the scripts used during the building process and the metadata; the metadata folders, whose most right name in this context is *layers*, can be easily recognized as their name start with the keyword *meta-*.
Every layer is organized following a well defined structure and is basically composed by two types of information: the **configuration** files and the **recipe** files. Those two type of files represent the very necessary information to create a custom layer and will be later useful.

While the configuration files set up the properties of each layer in the form of environment variables use by Bitbake, the recipe files are in charge of instruct it during the software building

process. There are several configuration file types but they can all be spotted by their extension which is *.conf*; one of the most important and easy to understand is the **layer.conf** file that can be found in the *conf* folder of each layer.
By looking at the one of the poky layer by typing

```
$ cat poky-pyro-17.0.1/meta-poky/conf/layer.conf
```

it is possible to see how this layer.conf files is instructing the Bitbake tool on the contents of the folder itself by adding the recipe files (**.bb** extension) and the "append" files (other files required during the building process, **.bbappend** extension) to an environment variable named **BBFILES**.

```
# We have a conf and classes directory, add to BBPATH
BBPATH =. "${LAYERDIR}:"

# We have recipes-* directories, add to BBFILES
BBFILES += "${LAYERDIR}/recipes-*/*/*.bb \
${LAYERDIR}/recipes-*/*/*.bbappend"

BBFILE_COLLECTIONS += "yocto"
BBFILE_PATTERN_yocto = "^${LAYERDIR}/"
BBFILE_PRIORITY_yocto = "5"

# This should only be incremented on significant changes that will
# cause compatibility issues with other layers
LAYERVERSION_yocto = "3"

LAYERDEPENDS_yocto = "core"

REQUIRED_POKY_BBLAYERS_CONF_VERSION = "2"
```

This is the basic concept behind the configuration files; a better look at the recipes will be given later while explaining the driver creation process adopted during the project.

From the main folder of the Poky-Pyro package is now possible to run the script which takes care of initializing the previously mentioned *building environment* for the Bitbake engine.
It is thus possible to run the script from inside the Poky-Pyro folder with

```
/poky-pyro-17.0.1$ ./oe-init-build-env
You had no conf/local.conf file. This configuration file has therefore been
created for you with some default values. You may wish to edit it to, for
example, select a different MACHINE (target hardware). See conf/local.conf
for more information as common configuration options are commented.

You had no conf/bblayers.conf file. This configuration file has therefore been
created for you with some default values. To add additional metadata layers
into your configuration please add entries to conf/bblayers.conf.

The Yocto Project has extensive documentation about OE including a reference
manual which can be found at:
```

```
http://yoctoproject.org/documentation

For more information about OpenEmbedded see their website:
http://www.openembedded.org/


### Shell environment set up for builds. ###

You can now run 'bitbake <target>'

Common targets are:
core-image-minimal
core-image-sato
meta-toolchain
meta-ide-support

You can also run generated qemu images with a command like 'runqemu qemux86'
```

As can be spotted by the output of the command the script "sets up shell environment for builds"; this means that each time a build is going to be modified the script has to be launched at first in order to prepare the environment expected by Bitbake.
The script results in the creation of a folder called "build", unless it is specified differently, within the main folder of the Poky-Pyro package; from now on that folder will contain any of the building output of the Bitbake building process.

Since the newly created **build** directory represents well the format of any of the building environment created in the several tries done during this project it is important to point out its basic topology and the configuration files that it stores before moving into the actual project build. After the first script execution the build folder will contain only the configuration file folder **conf**.

Figure 10: poky-pyro-17.0.1/build/conf folder contents

The two files **bblayers.conf** and **local.conf** inside the conf/ folder have a crucial importance when customizing the Linux Embedded build; the first one is the configuration file which sets for Bitbake the paths where it will find the source code tu be built up and some run time environment variables. Having a look at the file:

```
$ cat poky-pyro-17.0.1/build/conf/bblayers.conf
# POKY_BBLAYERS_CONF_VERSION is increased each time build/conf/bblayers.conf
# changes incompatibly
POKY_BBLAYERS_CONF_VERSION = "2"

BBPATH = "${TOPDIR}"
BBFILES ?= ""

BBLAYERS ?= " \
/home/user/poky-pyro-17.0.1/meta \
/home/user/poky-pyro-17.0.1/meta-poky \
/home/user/poky-pyro-17.0.1/meta-yocto-bsp \
"
```

can be noticed out that the variable **BBLAYERS** is the one storing the list of the layers included within the build as a concatenation of string paths; this variable will be parsed by Bitbake at run-time. At the moment the build is set up to build the three layers necessary to produce the the plain Poky version of the Linux Embedded OS for machines included within the **meta-yocto-bsp** layer.

One of the informations contained in the second file **local.conf** is in fact the MACHINE architecture selected for the build; by taking a first look at the local.conf file stored in the build/conf/ directory is possible to describe some of the most useful environment variables that are going to be used while building the actual OS distribution:

```
$ cat poky-pyro-17.0.1/build/conf/local.conf
#
# This file is your local configuration file and is where all local user
# settings are placed. The comments in this file give some guide to the options
# a new user to the system might want to change but pretty much any
# configuration option can be set in this file. More adventurous users can look
# at local.conf.extended which contains other examples of configuration which
# can be placed in this file but new users likely won't need any of them
# initially.
#
# Lines starting with the '#' character are commented out and in some cases the
# default values are provided as comments to show people example syntax.
# Enabling the option is a question of removing the # character and making any
# change to the variable as required.
#
#
# Machine Selection
#
# You need to select a specific machine to target the build with. There are a
# selection of emulated machines available which can boot and run in the QEMU
# emulator:
#
#MACHINE ?= "qemuarm"
#MACHINE ?= "qemuarm64"
#MACHINE ?= "qemumips"
#MACHINE ?= "qemumips64"
#MACHINE ?= "qemuppc"
#MACHINE ?= "qemux86"
#MACHINE ?= "qemux86-64"
#
# There are also the following hardware board target machines included for
# demonstration purposes:
#
#MACHINE ?= "beaglebone"
#MACHINE ?= "genericx86"
#MACHINE ?= "genericx86-64"
```

```
#MACHINE ?= "mpc8315e-rdb"
#MACHINE ?= "edgerouter"
#
# This sets the default machine to be qemux86 if no other machine is selected:
MACHINE ??= "qemux86"
```

The very first environment variable to be set up by this file is the **MACHINE** variable; the reported output shows the possible architecture supported by the meta-yocto-bsp layer at the moment and points out how by default it is set up as **qemux86** architecture by default. **QEMU** is another of the powerful tools supported by the Open Embedded Project and the Yocto Project whose main functionality is to run a certain OS binary into a virtualized environment; this kind of tool can be useful when a certain OS could need to be tested while the destination hardware is not available.
Since this configuration file is intended as one of the customization entry point for the production of the OS we could also have needed to modify directly from this file the MACHINE architecture; This could have been the case if we had decided to use, for instance, a Beaglebone board instead of the UDOO Neo. To have a look at the recipes of the possible machine architectures simply list the available target architecture recipes included within the meta-layer-bsp by running:

```
/poky-pyro-17.0.1$ ls meta-yocto-bsp/conf/machine/
beaglebone.conf genericx86-64.conf include
edgerouter.conf genericx86.conf mpc8315e-rdb.conf
```

that as can be pointed out are the same machine architectures reported as an example within the local.conf file.

It is at this point already possible to run the build of the of the Pyro distribution and to test it using the QEMU environment simply by running the commands:

```
/poky-pyro-17.0.1/build$ ls bitbake <target_image>
```

and, after the process completion,

```
/poky-pyro-17.0.1/build$ runqemu qemux86
```

The QEMU virtualizer will take care of create a test bench for the OS built within the current Bitbake environment.

2.4 Building a custom OS for UDOO Neo

In order to create a working operating system for a custom board the Yocto Project was useful during these earliest phases of the thesis work. As pointed out in the previous chapter the three mentioned projects (Yocto Project, Open Embedded an Bitbake) are strictly related one with each other even if they coexists as different projects. To understand better the origin of this relationship may be interesting to spend some words about the Yocto Project birth.
As explained in the previous chapter the Poky version of Linux Embedded is the reference distribution maintained by Open Embedded and included within the Yocto Project reference package; in fact the origin of the project is related to this version of the OS. Thanks to his Open Source nature the Poky Linux OS became very popular within the embedded field at the point that in 2010 the Intel Company and the Linux Foundation started a collaboration gave origin to the Yocto Project.

The two projects share:

- Bitbake metadata layer
- Open Embedded Core metadata layers
- an aligned development

The great strength of the project is it's capability to create an OS from the scratch by simply adding to the Yocto basic structure the BSP for the architecture of the destination device.
The BSP or Board Support Package is an hardware-specific layer of software providing those functionalities that are strictly related to the hardware implementation like drivers and other routines that can grant full access to hardware. Since the structure of an hardware or a SoC like a development board (as in this case) is well clear to the hardware vendor it is typically in the interests of the vendor itself to develop and maintain its own hardware-related BSP metadata layer. Some of the most famous development boards have a huge community support and is thus quite easy to retrieve the BSP metadata layer for them. For instance if the choice for this project would have been a Raspberry Pi as development board we could have found the related BSP directly within the Yocto Project Source Repository index (`http://git.yoctoproject.org/cgit/cgit.cgi/?q=`) by quering it.

Since in this case the board used is a UDOO Neo full (the hardware specifications will be described in the next chapter) the first step was to retrieve the BSP related to this board. Order tog get the required packages i used the **repo** tool; repo is an utility intended to make easier the use of Git since it takes care of fetching the data form Git repositories as well as perform actions like revision control an versioning through data upload in a automatic fashion. Run the following commands to install *repo* within the home directory:

```
$ mkdir ./bin
$ curl http://commondatastorage.googleapis.com/git-repo-downloads/repo > ./bin
/repo
$ chmod a+x ./bin/repo
$ export PATH=$PATH: ./bin
```

Then let's set up a folder to contain the downloaded source

```
$ mkdir udoo-community-bsp
$ cd udoo-community-bsp
/udoo-community-bsp$ repo init -u https://github.com/graugans/udoo-community-b
sp-platform -b pyro
$ repo sync
```

As can be noticed the "repo init -u" command is setting up the folder directly connecting it with a specified git repository; this repository is a a forked one from a most general fsl-community-bsp-platform repository intended to collection the BSP for Freescale products. Eventually the "sync" command will start the download of the data, thus for those steps will be necessary both an Internet connection and some free space on the mass storage of the host PC.

During the synchronization process the required data to start a build is downloaded from the git repository; after the sync command the contents of the udoo-community-bsp folder is changed:

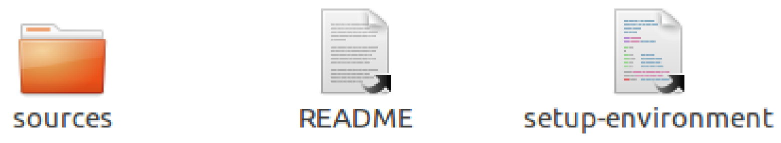


Figure 11: udoo-community-bsp folder contents

The folder now contains another folder named **sources** and a python script named **setup-environment**; the script basically takes care of setting up the environment to run the Bitbake command and starting a new OS build. This reminds the **oe-init-build-env** used to set up the environment for the Poky OS distribution. In fact not only the script itself it's called as wrapped within this new script, but also the metadata of the Poky distribution is present within the BSP (and can be found by reaching the directory /udoo-community-bsp/sources/poky). This can be easily spotted out by by running:

```
$ cat udoo-community-bsp/sources/poky/oe-init-build-env | grep oe-init-build-en
v
# Normally this is called as '../oe-init-build-env <builddir>'
THIS_SCRIPT="$(pwd)/oe-init-build-env"
```

To set up the environment it's now necessary to run the previously mentioned script; as can be noticed by having a look at the example reported inside the script

```
.
.
.
Examples:

- To create a new Yocto build directory:
$ MACHINE=imx6qsabresd DISTRO=fslc-framebuffer source $PROGNAME build

- To use an existing Yocto build directory:
$ source $PROGNAME build
.
.
.
```

in order to successfully run the script it's necessary to specify both the MACHINE architecture (as discussed for the previous Poky build) and the DISTRO version; moreover it's advisable to give a name to the build directory which by default will be again "build".
To star the build let's run the command:

```
/udoo-community-bsp$ MACHINE=udooneo DISTRO=poky source ./setup-environment udo
oneo_poky_udoo-image-full-cmdline
.
.
.
EULA has been accepted.

Welcome to Freescale Community BSP
```

```
The Yocto Project has extensive documentation about OE including a
reference manual which can be found at:
http://yoctoproject.org/documentation

For more information about OpenEmbedded see their website:
http://www.openembedded.org/

You can now run 'bitbake <target>'

Common targets are:
core-image-minimal
meta-toolchain
meta-toolchain-sdk
adt-installer
meta-ide-support

Your build environment has been configured with:

MACHINE=udooneo
SDKMACHINE=i686
DISTRO=poky
EULA=
```

Under the acceptance of the **EULA** (End-User License Agreements) an output like the one listed above will be produced; as can be noticed the variables MACHINE and DISTRO have been configured correctly, however this choice is only one of the possible choices. To check all the machines supported by the BSP just downloaded simply create a shell called **list-available_machines.sh** script in the **sources** folder running:

```
$ touch udoo.community-bsp/sources/list_available_machines.sh
```

then edit it with a text editor as *vi* or *gedit* and fill it with the following content

```
#!/bin/bash

for D in `find . -name 'machine'`
do
for F in `find "$D"`; do
if [[ $F == *.conf ]]; then
S=$(echo "${F%.*}" | sed "s/.*\///")
if [ "$S" != "machine" ] || [ "$S" != "xorg" ]; then
echo $S
fi
fi
done
done
```

At this point run the script from inside the **sources** directory with

```
$ cd udoo-community-bsp/sources/
/udoo-community-bsp/sources$ source ./list-available-machines.sh
udooneo
secosbca62
```

```
udooqdl
udoox86
qemux86copy
qemux86
qemuarm64
qemux86-64
qemuppc
qemumips
qemumips64
qemuarm
edgerouter
genericx86
beaglebone
genericx86-64
mpc8315e-rdb
cubox-i
pcm052
m28evk
tx6s-8035
cm-fx6
cfa10036
wandboard
ventana
colibri-imx6
.
.
.
```

The result is going to be a quite long list of possible MACHINE architecture choices included within the fetched BSP version.

The same has been done for the available target images; the image represents the configuration for the produced OS; creating a script named **list_available_images** within the **sources** directory with the following content:

```
$ touch udoo-community-bsp/sources/list_available_machines.sh
```

containing the following bash script:

```
#!/bin/bash

for D in ‘find . -name ’images’‘
do
for F in ‘find "$D"‘; do
if [[ $F == *.bb ]]; then
S=$(echo "${F%.*}" | sed "s/.*\///")
echo $S
fi
done
done
```

it becomes possible to list all the default target images included within the package:

```
$ cd udoo-community-bsp/sources/
```

```
/udoo-community-bsp/sources$ source ./list_available_images
udoo-image-qt5
udoo-image-full-cmdline
initramfs-debug-image
initramfs-kexecboot-image
initramfs-kexecboot-klibc-image
core-image-minimal-xfce
wic-image-minimal
error-image
oe-selftest-image
test-empty-image
core-image-minimal
.
.
.
```

For our purposes the image that has been at the first step used is the **udoo-image-full-cmdline.bb**; this image in particular offers a full functional console Linux system image well suited for the UDOO Neo boards; it's possible to check the main features that will be included by this image recipe by taking a look at it running:

```
/udoo-community-bsp$ cat sources/meta-udoo/recipes-udoo/images/udoo-image-ful
l-cmdline.bb
DESCRIPTION = "A console-only image with more full-featured Linux system \
functionality installed. Tailored for the UDOO boards"

IMAGE_FEATURES += "splash ssh-server-openssh package-management"

UDOO_EXTRA_INSTALL_arm = " \
imx-gpu-viv \
imx-gpu-viv-demos \
packagegroup-fsl-tools-gpu \
i2c-tools \
dtc \
${@base_conditional("ENABLE_CAN_BUS", "1", "canutils", "", d)} \
"
UDOO_EXTRA_INSTALL_x86-64 = " \
"

IMAGE_INSTALL = "\
packagegroup-core-boot \
packagegroup-core-full-cmdline \
packagegroup-base \
${CORE_IMAGE_EXTRA_INSTALL} \
${UDOO_EXTRA_INSTALL} \
resize-rootfs \
tmux \
binutils \
minicom \
mmc-utils \
"
```

```
inherit core-image


# Needed by resize-rootfs
IMAGE_DEPENDS_ext4 = "e2fsprogs-native"
IMAGE_CMD_ext4_arm_append () {
# Label the disk rootfs
e2label ${DEPLOY_DIR_IMAGE}/${IMAGE_NAME}.rootfs.ext4 rootfs
}
```

As can be seen some useful features are included within this image recipe, like the **ssh-server-openssh** service that will allow to the board remotely through the Secure Shell protocol or the **minicom** tool that will be used in earliest steps to start a serial communication between the board an the host PC used to develop the OS. The statement *inherit core image* instruct the Bitbake process on the inheritance relationship between this image and the **core-image** class; in particular this image inherits all the metadata which has already been included within the inherited image. Moreover there are not particular problems if adding the same package multiple times since the Bitbake process will take care o parse the list of the included features smartly.

One of the problems of the build process of an image is however the huge amount of time needed to complete the process; in fact, depending on both the number of processors of the PC allocated on this process and the download speed of the Internet connection, the process may last for hours. On the PC used for this thesis project for instance the the first build process lasted approximatively 6 - 8 hours, considering that no other huge processes were running at the same time (thus providing 4 threads to the Bitbake process) and having a quite slow Internet connection reaching peaks of **1.7 - 2 Mbits/s**. This limitation makes of course obvious the necessity of a much powerful machine and a better Internet connection in order to allow the completion of a new build in a minor time.

As underlined this limitation is managed perfectly by the Bitbake process; after the build process completion will in fact a significant amount of output will be produced within the build directory; a part of it will of course be the destination image but another output becomes useful when forecasting several build images. This output is called **State Cache** and basically represent an updating "checkpoint" within the build process; thanks to this particular output the Bitbake engine is able to decide weather a task needs to be done or if it is already been managed by a previous build process. In this way is possible to stop a build execution and re-starting it from the same point exploiting everything that was already been made.

In order to exploit this feature a more structured folder was set up, in order to save time and additional space consumption derived from future builds. Since the builds used within this project are based on the same image **udoo-image-full-cmdline.bb** the newly created build within the "udooneo_poky_udoo-image-full-cmdline" folder shall acts perfectly as the basic skeleton for future builds extending it with additional features but based on the same structure of metadata.

Since at this point the folder just mentioned were all moved into a containing folder named **thesis** the path are going to be different from now on.

To configure properly the build process the .conf files inside the **conf** directory of the build were modified. The **bblayers.conf** file was modified to look as follows:

```
$ cat thesis/udoo-community-bsp/udooneo_poky_udoo-image-full-cmdline/conf/bb
layers.conf
POKY_BBLAYERS_CONF_VERSION = "2"

BBPATH = "${TOPDIR}"
BSPDIR := "${@os.path.abspath(os.path.dirname(d.getVar('FILE', True)) + '/../
..')}"

BBFILES ?= ""
BBLAYERS = " \
${BSPDIR}/sources/poky/meta \
${BSPDIR}/sources/poky/meta-poky \
${BSPDIR}/sources/poky/meta-yocto \
${BSPDIR}/sources/poky/meta-yocto-bsp \
\
${BSPDIR}/sources/meta-openembedded/meta-oe \
${BSPDIR}/sources/meta-openembedded/meta-multimedia \
${BSPDIR}/sources/meta-openembedded/meta-python \
${BSPDIR}/sources/meta-openembedded/meta-networking \
\
${BSPDIR}/sources/meta-qt5 \
\
${BSPDIR}/sources/meta-freescale \
${BSPDIR}/sources/meta-freescale-3rdparty \
${BSPDIR}/sources/meta-freescale-distro \
${BSPDIR}/sources/meta-udoo \
"
```

Those settings inside the bblayers.conf configuration file will instruct Bitbake with the paths list of the layer that are going to be used during this build; as can be seen the current configuration is including the layers of the Poky reference distribution, the Open Embedded Core metadata layers, the Qt5 layer which include the Qt5 developer tools for user interface application design and the metadata layers from Freescale Company, including the specific one for the UDOO boards as well. It is important to comment the BSPDIR environment variable which is set up each time the **setup-environment** command is called; this variable represents the path of the folder containing the BSP downloaded from Github during the previous steps and thus it points to the **udoo-community-bsp** folder.

This variable is reported and used also in the other configuration file that has to be modified; the **local.conf** file storing the user build options for the current build are was modified to look as:

```
MACHINE ??= 'udooneo'
DISTRO ?= 'poky'
PACKAGE_CLASSES ?= "package_rpm"
USER_CLASSS ?= "buildstats image-mklibs image-prelink"
IMAGE_FEATURES = "debug-tweaks tools-debug eclipse-debug"
IMAGE_INSTALL_append = " tcf-agent openssh-sftp-server "
PATCHRESOLVE = "noop"
```

```
BB_DISKMON_DIRS = "\
STOPTASKS,${TMPDIR},1G,100K \
STOPTASKS,${DL_DIR},1G,100K \
STOPTASKS,${SSTATE_DIR},1G,100K \
STOPTASKS,/tmp,100M,100K \
ABORT,${TMPDIR},100M,1K \
ABORT,${DL_DIR},100M,1K \
ABORT,${SSTATE_DIR},100M,1K \
ABORT,/tmp,10M,1K"
PACKAGECONFIG_append_pn-qemu-native = " sdl"
PACKAGECONFIG_append_pn-nativesdk-qemu = " sdl"
CONF_VERSION = "1"

DL_DIR ?= "${BSPDIR}/downloads/"
ACCEPT_FSL_EULA = "1"
```

To have a look at all the differences between the original contents of the local.conf file and the contents listed above it is possible to write this version into a temporary file named tmp.conf and run the command:

```
 $ diff thesis/udoo-community-bsp/udooneo_poky_udoo-image-full-cmdline/conf/lo
cal.conf ./tmp.conf
4,5c4,6
< EXTRA_IMAGE_FEATURES ?= "debug-tweaks"
---
> IMAGE_FEATURES = "debug-tweaks tools-debug eclipse-debug"
> IMAGE_INSTALL_append = " tcf-agent openssh-sftp-server "
```

as shown by the command output several changes were made with respect to the original contents. In particular the **debug-tweaks** package has been moved from the original environment variable as well as some additional packages that are:

1. **tools-debug**, a package containing tools needed to perform a remote PC-board debug;
2. **tcf-agent**, a service that during the next steps will allow to establish a remote connection with the on-board file system directly from the Eclipse IDE;
3. **openssh-sftp-server**, a package containing the tools needed to perform file transfer between the host PC and the board through SFTP protocol

Now that the configuration files have been edited according to the project requirements the environment is eventually ready to start the build of the reference build for this thesis.
To start the build it is enough to run the command **bitbake target_image**, where the target can be any possible recipe within the sources used for this project.

```
/thesis/udoo-community-bsp/udooneo_poky_udoo-
image-full-cmdline$ bitbake udoo-image-full-cmdline
```

The Bitbake engine will start executing all the task needed to build the OS image in a well defined order; it is in fact important to point out that the Bitbake process is strictly related to a well defined structure of inter task dependencies. In other words in order to execute a certain task A it may happen that other two task B and C may be executed first. The dependencies between the task are defined within the recipes of the each software component. The command will produce the following output:

```
/thesis/udoo-community-bsp/build$ bitbake udoo-image-full-cmdline
Loading cache: 100% |###########################################| Time: 0:00:00
Loaded 450 entries from dependency cache.
Parsing recipes: 100% |#########################################| Time: 0:02:11
Parsing of 2178 .bb files complete (296 cached, 1882 parsed). 2961 targets, 288
 skipped, 0 masked, 0 errors.
NOTE: Resolving any missing task queue dependencies

Build Configuration:
BB_VERSION = "1.34.0"
BUILD_SYS = "x86_64-linux"
NATIVELSBSTRING = "ubuntu-16.04"
TARGET_SYS = "arm-poky-linux-gnueabi"
MACHINE = "udooneo"
DISTRO = "poky"
DISTRO_VERSION = "2.3.2"
TUNE_FEATURES = "arm armv7a vfp thumb neon callconvention-hard cortexa9"
TARGET_FPU = "hard"
meta
meta-poky = "HEAD:a75a2f4272226e924d8c9deb699a19ca9e606a5b"
meta-oe
meta-multimedia
meta-python
meta-networking = "HEAD:dfbdd28d206a74bf264c2f7ee0f7b3e5af587796"
meta-qt5 = "HEAD:c6aa602d0640040b470ee81de39726276ddc0ea3"
meta-freescale = "HEAD:06178400afbd641a6709473fd21d893dcd3cfbfa"
meta-freescale-3rdparty = "HEAD:9613dbc02ca970122a01c935bc9e5789ced73a9d"
meta-freescale-distro = "HEAD:cd5c7a2539f40004f74126e9fdf08254fd9a6390"
meta-udoo = "HEAD:79350fc9baf5b75e929fd2dbd59d3e3dbd8cc402"
```

From the previous output it is possible to spot how the Bitbake process try to load the ”current status of the build” from the cache folder; since this is the first build within the current environment Bitbake starts from an empty status.
After this step the process parses each recipe included within the build in order to manage correctly the task queue according to the inter task dependencies. If the recipes are parsed without any error the command will display the main building configurations that have been adopted:

- **BB_VERSION** is the Bitbake program version;
- **BUILD_SYS** is the variable storing the architecture of the PC used to start the build (this information becomes useful when cross-developing application from the host PC to the board);
- **NATIVELSBSTRING** represents the Linux version running o the PC;
- **TARGET_SYS** represents the target system specifications (ARM CPU architecture, Poky Linux OS, GNU Linux C Libraries).

What follows are some other variable and a recap of some already discussed information like MACHINE architecture, DISTRO version and the metadata layers included in the build.
Moreover the **bitbake** command will print on the terminal the completion status of the whole process in term of tasks still needed to be run and a percentage estimation of the work already

completed; however it is not advisable to make an estimation of the completion process only relying on these two informations. The main reason is that the times needed to execute two different tasks may result quite different according to the nature of the task. Before moving to see the output generated by the process it is advisable to give an hint on the possible kind of task that the bitbake command will eventually execute while parsing the recipe for a given package.

According to the structure of the Open Embedded build system the Bitbake build engine will eventually execute different kind of tasks; those types can be summarized as follows:

1. **FETCH**: obtains the source code; this kind of tasks retrieves the source code that can be a local resource or downloaded from the web through the different file transfer protocols supported by Bitbake (for instance HTTP, FTP and SFTP).
2. **EXTRACT**: unpacks the source code; during this task the system automatically detects the format of the fetched source code and extract it into the working directory.
3. **PATCH**: applies patches for bug fixes or added functionalities; this tasks basically updates files that have been modified form some reason as bug fixing or added functionalities.
4. **CONFIGURE**: prepares the build system according to the settings imposed by the local environment; this task also takes care of generating a makefile script for the target system through the Autotools build system.
5. **BUILD**: compiles and links the source code; this task run the script files generated at the previous step to compile the source code.
6. **INSTALL**: copies binaries and other files into the target directory; this task basically copies the produced output files into the right location of the file system.
7. **PACKAGE**: packs the produced binaries and needed preparing them to be installed on other systems; this task prepares the developed output for being distributed.

When the build process will be eventually finished the build folder will look different since the Bitbake process should have populated it as shown in the following image.

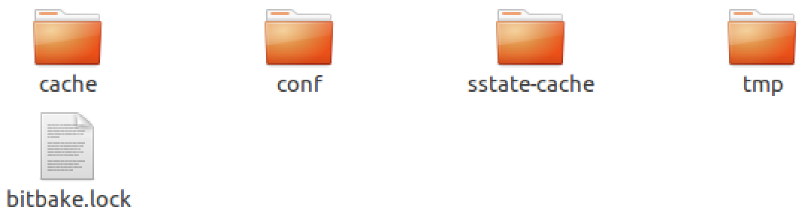


Figure 12: Build folder contents after bitbake completion

Two of the folder created at this point will be useful during next steps and it's thus useful to explain their contents. The **sstate-cache** folder contains the states of the build produced during the software process completion; sharing this folder between multiple builds can significantly decrease the completion time for future builds.

On the other side the **cache** folder represents a ”local” cache folder that is specific for the build run from the current environment; it is thus not possible to share this folder.
The third folder is named **tmp** and contains the output generated by the bitbake build process; this folder is likely to explode in dimensions after several builds since it contains extracted source code, the compilation outputs and the images of the Kernel and of the file system.
The build session can be resumed for any change by running the command:

```
/thesis/udoo-community-bsp$ source ./setup-environment udooneo_poky_udoo-image
-full-cmdline/

Welcome to Freescale Community BSP

The Yocto Project has extensive documentation about OE including a
reference manual which can be found at:
http://yoctoproject.org/documentation

For more information about OpenEmbedded see their website:
http://www.openembedded.org/

You can now run 'bitbake <target>'

Common targets are:
core-image-minimal
meta-toolchain
meta-toolchain-sdk
adt-installer
meta-ide-support

Your configuration files at udooneo_poky_udoo-image-full-cmdline/ have not been
 touched.
```

At this point the image of the OS to be deployed on the development board is eventually ready. It can be found within the **tmp** director at the path *tmp/deploy/images/udooneo/*: between all the output files contained within the folder the target file to be deployed on-board is the file named **udoo-image-full-cmdline-udooneo.wic.bz2**.

2.5 Booting the system

During the next step a peripheral device will be flashed with the OS binaries it is **important to be careful** during the choice of the right device because otherwise some disk contents may be lost; under Linux Ubuntu is possible to find the list of the connected devices under */dev* folder.
To check the name of the SD card that is going to be connected the easiest way is to prepare two files storing the *dmesg* command output before and after the SD card insertion and then check the two files for differences.
Run:

```
$ touch tmp_before.txt
$ dmesg >> tmp_before.txt
```

Now the micro SD card that is going be inserted within the board can be plugged into the PC SD card reader slot through a micro-SD to SD adapter. To check the name of the SD card run the commands:

```
$ touch tmp_after.txt
$ dmesg >> tmp_after.txt
$ diff tmp_before.txt tmp_after.txt
1090a1091,1095
> [ 7799.662352] mmc0: new high speed SDHC card at address 59b4
> [ 7799.690604] mmcblk0: mmc0:59b4 USD 7.51 GiB
> [ 7799.691592] mmcblk0: p1
> [ 7800.225224] EXT4-fs (mmcblk0p1): recovery complete
> [ 7800.229344] EXT4-fs (mmcblk0p1): mounted filesystem with ordered data mode
. Opts: (null)
```

From the output above it is now possible to get the correct name that di file system is using to identify the device that can in fact be found under the */dev* directory named (in this case) as **/dev/mmcblk0**. To flash the image on the micro-SD card it is sufficient to run the commands:

```
$ sudo bzcat thesis/udoo-community-bsp/udooneo_poky_udoo-image-full-cmdline/tm
p/deploy/images/udooneo/udoo-image-full-cmdline-udooneo.wic.bz2 |
 sudo dd of=/dev/mmcblk0 bs=32M
```

The previous command will decompress the file containing the image and will redirect the output towards the second command, that will eventually copy the received input into the SD card as memory blocks with a maximum block size of 32 MBytes each.

To boot the system now it's enough to insert the micro-ds card into the proper slot site on the back side of the board and connect the board to the host PC through a micro-USB to USB cable.

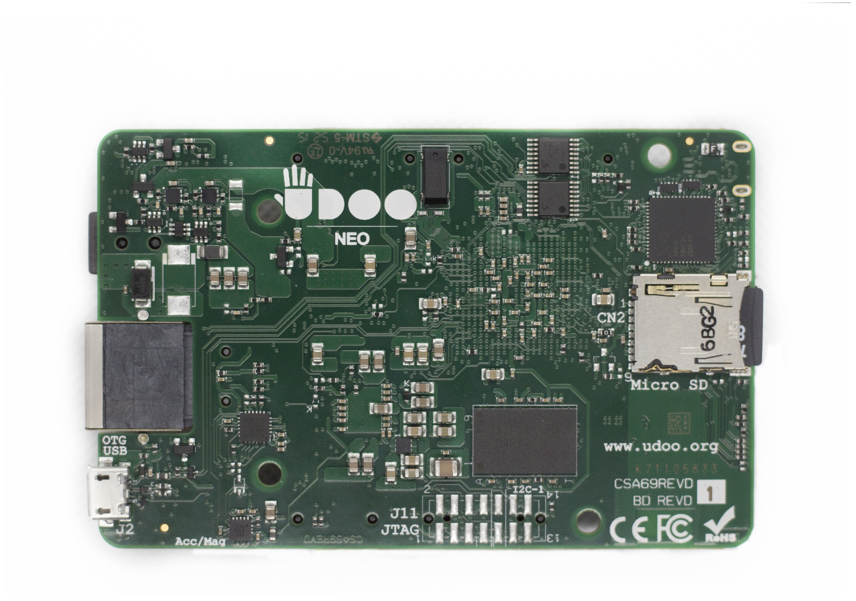


Figure 13: Back view of the UDOO Neo board

The board will start the bootstrap phase from the SD card and the bootloader embedded within the same memory will start loading the Kernel of Linux Embedded as explained in the previous sections of this chapter; eventually the bootloading phase will finish and this can be spotted by a change into the lightening behavior of the service led near the micro-USB input of the board.

After a successful bootstrap will be possible to connect directly with the board through di micro USB cable use to provide power to the system; to check weather the device has been correctly found by the OS running on the PC it will be enough to repeat the procedure adopted to find out the name of the name of the micro SD card within the /dev folder.
In this particular case the device name assigned to the board is **ttyACM0**; it is possible to connect with the board by using the terminal serial program **Minicom**.
In order to setup the Minicom configuration properly it has been modified the default configuration for the whole duration of the project; to setup the Minicom configuration it is necessary to run:

```
$ sudo minicom -s
           +-----[configuration]------+
           | Filenames and paths |
           | File transfer protocols |
           | Serial port setup |
           | Modem and dialing |
           | Screen and keyboard |
           | Save setup as dfl |
           | Save setup as.. |
           | Exit |
           | Exit from Minicom |
           +--------------------------+
```

and then to set the default configuration from the *Serial port setup* button of the menu, as:

```
    +-----------------------------------------------------------------------+
    | A - Serial Device : /dev/ttyACM0 |
    | B - Lockfile Location : /var/lock |
    | C - Callin Program : |
    | D - Callout Program : |
    | E - Bps/Par/Bits : 115200 8N1 |
    | F - Hardware Flow Control : No |
    | G - Software Flow Control : No |
    | |
    | Change which setting? |
    +-----------------------------------------------------------------------+
```

From now on it is possible to connect with the board and to enter as **root** by simply running:

```
Welcome to minicom 2.7

OPTIONS: I18n
Compiled on Feb 7 2016, 13:37:27.
Port /dev/ttyUSB0, 01:25:16

Press CTRL-A Z for help on special keys



Poky (Yocto Project Reference Distro) 2.3.2 udooneo /dev/ttyACM0

udooneo login: root
root@udooneo:#
```

From this terminal we can interact with the system with the bash commands just as a common Linux PC
As a prove of the correct behavior of the OS preliminary operations attended by the Linux Kernel during the Bootstrap it is possible to stream debug informations that are flushed out to the UART1 port of the device.
To perform this operation will be however necessary to use a TTL to USB adapter plugged into the board and into the PC; the board pinout and the used hardware will be however explained within the next chapter before moving onto the driver design.

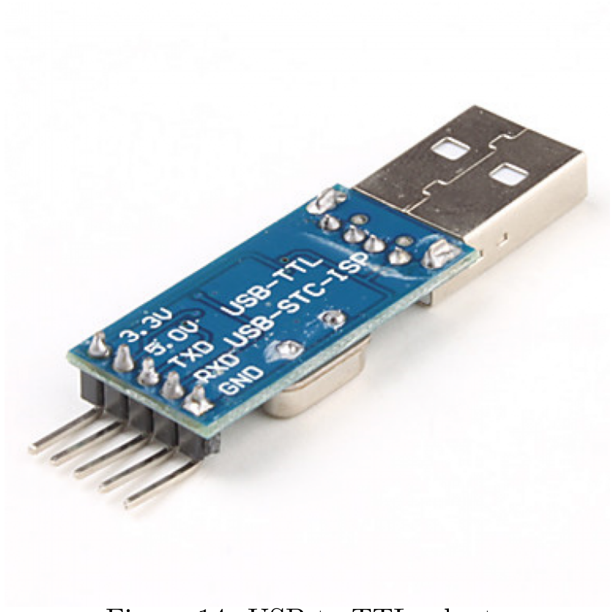


Figure 14: USB to TTL adapter

The proper connections are:

1. TXD pin of the adapter with GPIO 46 of the J7 header;
2. RXD pin of the adapter with GPIO 47 of the J7 header;
3. GND pin of the adapter with anyone of the pin labeled as GND of the internal banks;

If the proper connections have been established, a new device will be displayed within the /dev folder of the host PC (inside the same folder of the board Linux Embedded OS as well); the procedure to follow in order to retrieve easily the device names of the inserted modules remains the same: in this particular case the host PC sees a new **/dev/ttyUSB0** device while the board sees the PC as:

1. /dev/ttyGS0 on the serial micro USB line;
2. /dev/ttymxc0 on the debug connection;

To check out the boot process it is enough to access to the device /dev/ttyUSB0 with Minicom and launch from the first terminal the command **reboot**; on the debug terminal will be displayed the following output showing the power-off and reboot phases and thus the bootloading process:

```
Broadcast message from root@udooneo (pts/0) (Sun Dec 17 16:50:20 2017):

The system is going down for reboot NOW!
```

```
UIM SYSFS Node Found at /sys/./devices/soc0/kim/install
Stopping uim-sysfs daemon.
Stopping OpenBSD Secure Shell server: sshdstopped /usr/sbin/sshd (pid 345)
.
[ ok ]pping Avahi mDNS/DNS-SD Daemon: avahi-daemon
Stopping atd: OK
Stopping bluetooth
/usr/libexec/bluetooth/bluetoothd
Stopping system message bus: dbus.
[2017-12-17 16:50:20 UTC] (sys) Stopping
stopping mountd: done
stopping nfsd: [ 73.654776] nfsd: last server has exited, flushing export cache
done
Error opening /dev/fb0: No such file or directory
Stopping system log daemon...0
Stopping kernel log daemon...0
Stopping tcf-agent: OK
stopping statd: done
ALSA: Storing mixer settings...
/usr/sbin/alsactl: save_state:1595: No soundcards found...
Stopping crond: OK
Stopping rpcbind daemon...
done.
Deconfiguring network interfaces... ifdown: interface eth0 not configured
done.
Sending all processes the TERM signal...
logout
Sending all processes the KILL signal...
Unmounting remote filesystems...
Deactivating swap...
Unmounting local filesystems...
[ 79.154004] EXT4-fs (mmcblk0p1): re-mounted. Opts: (null)
Rebooting... [ 81.409852] reboot: Restarting system

U-Boot SPL 2017.03+fslc+gac3b20c (Dec 12 2017 - 22:55:32)
Trying to boot from MMC1



U-Boot 2017.03+fslc+gac3b20c (Dec 12 2017 - 22:55:32 +0100)

CPU: Freescale i.MX6SX rev1.2 996 MHz (running at 792 MHz)
CPU: Extended Commercial temperature grade (-20C to 105C) at 39C
Reset cause: WDOG
Board: UDOO Neo FULL
I2C: ready
DRAM: 1 GiB
PMIC: PFUZE3000 DEV_ID=0x30 REV_ID=0x11
MMC: FSL_SDHC: 0
*** Warning - bad CRC, using default environment
```

```
In: serial
Out: serial
Err: serial
Net: FEC0 [PRIME]
Hit any key to stop autoboot: 0
switch to partitions #0, OK
mmc0 is current device
Scanning mmc 0:1...
Found U-Boot script /boot/boot.scr
517 bytes read in 339 ms (1000 Bytes/s)
## Executing script at 82000000
45832 bytes read in 261 ms (170.9 KiB/s)
4538720 bytes read in 622 ms (7 MiB/s)
## Flattened Device Tree blob at 83000000
Booting using the fdt blob at 0x83000000
Using Device Tree in place at 83000000, end 8300e307

Starting kernel ...

[ 0.000000] Booting Linux on physical CPU 0x0
[ 0.000000] Linux version 4.1.15-2.0.x-udoo+g7773e46 (oe-user@oe-host) (gcc ver
sion 6.3.0 (GCC) ) #1 7
.
.
.
```

To briefly comment the output above it is possible to recognize most of the function that were expected as correct by the Bootloader; in the very first part it is possible to see how the *reboot* command causes the stop of the active services on the board. When these services are stopped the system auto-resets via software thus causing a new Bootstrap.

Since the first stage bootloader running on the board is configured to search for a bootable image within some specific memory addresses, the board is able to find the bootable image of Linux Embedded on the micro SD card slot inserted on the back side. At this point the U-Boot (Universal Bootloader) bootloader flashed into the SD card is loaded into the RAM memory by the first stage bootloader. Since U-boot is again a multi stage bootloader, after the preliminary initialization it loads the U-boot SPL found into the same micro-SD card. The *Secondary Program Loader* then takes care of loading within the RAM memory the largest part ot the software needed to ends up the bootstrap. After the Kernel software has been loaded the output of the command still continues reporting hardware configurations of network resources and other devices.

3 Hardware References

This section is intended to describe the most important properties of the components used during this project; since the full list of the characteristics can be found on the hardware vendors websites the next pages are giving a brief description of the most significant of them. Moreover a part will be dedicated at giving some hints on the communication protocols that have been adopted to achieve the final result.

3.1 UDOO Neo Full Development Board

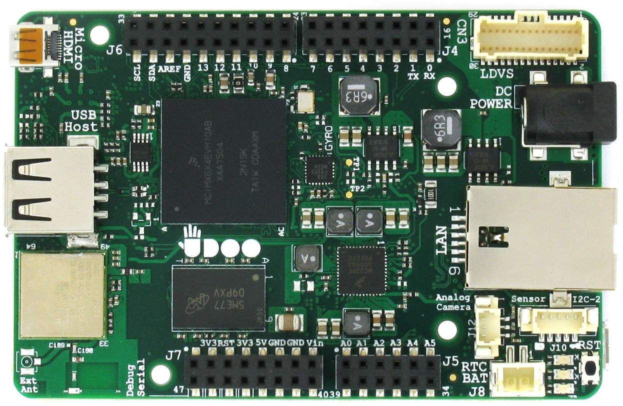


Figure 15: UDOO Neo Development board frontal view

The choice for the development board has been a UDOO Neo board; every reference about UDOO boards of the *Neo* line can be found at `https://www.udoo.org/udoo-neo/`.
The UDOO Neo family comes out with three different models of the same board; the choice for this project was the *Full* version which embeds the most hardware devices between the three models.
Summarizing the main hardware characteristics, the board embeds:

- i.MX 6SoloX processor by NXP®, embedding two different processors: the most performing is the ARM® Cortex-A9 core by Freescale, the second is an Arduino compatible Cortex-M4 core;
- 1GB RAM memory;
- Micro HDMI output to external displays;
- MicroSD card slot on the back;
- a USB 2.0 type A port;
- Wi-Fi module and Bluetooth module for wireless connectivity;
- 3 UART ports for serial connection;

- 5V DC Micro USB input to power the board and establish serial connections;
- Fast Ethernet connection;
- up to three I2C interfaces;

Moreover the boards embeds several sensors that have not been used during the project; in the next steps of development have been used some of the hardware interfaces listed above as the UART1 connection, the serial Micro USB connection, the Ethernet port and PINs from the pinout banks fo the board. Those connections will be specified step by step during the next chapters.
Here is for convenience the components locations of the UDOO Neo board (that can also be found on the website of the vendor)

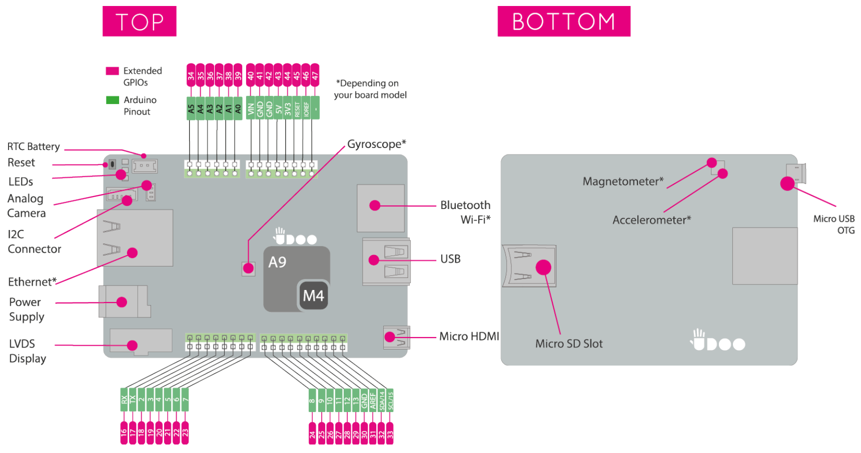


Figure 16: Components locations on the board

and the pinout map for the internal and external banks

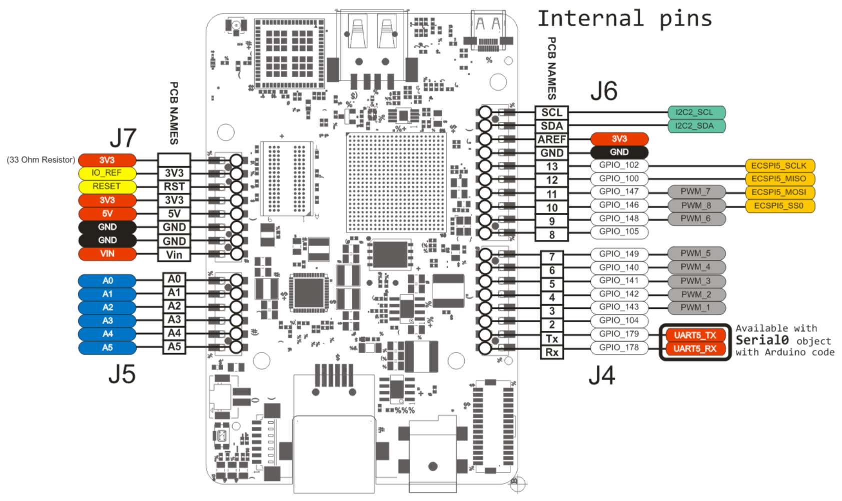


Figure 17: Pinout of the internal banks

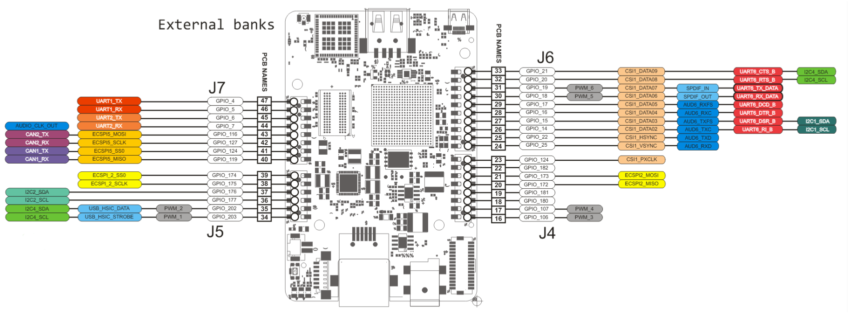

External banks
PCB NAMES
J7
UART1_TX GPIO_4 47
UART1_RX GPIO_5 46
UART2_TX GPIO_6 45
AUDIO_CLK_OUT UART2_RX GPIO_7 44
CAN2_TX ECSPI5_MOSI GPIO_116 43
CAN2_RX ECSPI5_SCLK GPIO_127 42
CAN1_TX ECSPI5_SS0 GPIO_124 41
CAN1_RX ECSPI5_MISO GPIO_119 40
ECSPI_2_SS0 GPIO_174 39
ECSPI_2_SCLK GPIO_175 38
I2C2_SDA GPIO_176 37
I2C2_SCL GPIO_177 36
I2C4_SDA USB_HSIC_DATA PWM_2 GPIO_202 35
I2C4_SCL USB_HSIC_STROBE PWM_1 GPIO_203 34
J5
PCB NAMES
J6
33 GPIO_21 CSI1_DATA09 UART6_CTS_B I2C4_SDA
32 GPIO_20 CSI1_DATA08 UART6_RTS_B I2C4_SCL
31 GPIO_19 PWM_6 CSI1_DATA07 SPDIF_IN UART6_TX_DATA
30 GPIO_18 PWM_5 CSI1_DATA06 SPDIF_OUT UART6_RX_DATA
29 GPIO_17 CSI1_DATA05 AUD6_RXFS UART6_DCD_B
28 GPIO_16 CSI1_DATA04 AUD6_RXC UART6_DTR_B
27 GPIO_15 CSI1_DATA03 AUD6_TXFS UART6_DSR_B I2C1_SDA
26 GPIO_14 CSI1_DATA02 AUD6_TXC UART6_RI_B I2C1_SCL
25 GPIO_22 CSI1_HSYNC AUD6_TXD
24 GPIO_25 CSI1_VSYNC AUD6_RXD
23 GPIO_124 CSI1_PXCLK
22 GPIO_182
21 GPIO_173 ECSPI2_MOSI
20 GPIO_172 ECSPI2_MISO
19 GPIO_181
18 GPIO_180
17 GPIO_107 PWM_4
16 GPIO_106 PWM_3
J4

Figure 18: Pinout of the external banks

3.2 MGC3130 Hillstar Development Kit

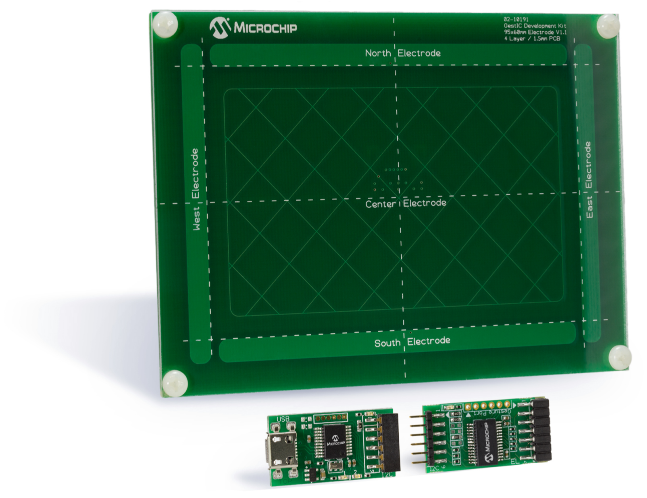


Figure 19: MGC3130 Hillstar Development Kit by Microchip

The whole project finds its bases onto the components that can be found within this Kit; any additional detail on this Kit can be found at `https://www.microchip.com/DevelopmentTools/ProductDetails.aspx?PartNO=DM160218`.
The package sold by Hillstar contains three fundamental components:

- the MGC3130 sensor for data tracking position and gesture recognition;
- a reference electrode PCB for signal sensing;
- an USB to I2C bridge to connect the sensor to a PC.

Moreover the package contains four foam hand bricks that will be used when configuring the controller through the Microchip Aurea evaluation software for GestIC sensors.
The same software has also been used to flash the firmware on any new MGC3130 Controller used during the project.

3.2.1 MGC3130 Controller

The **MGC3130 Unit** for position tracking and gesture recognition is the key element of the **MGC3130 Controller**, which is structured as follows:

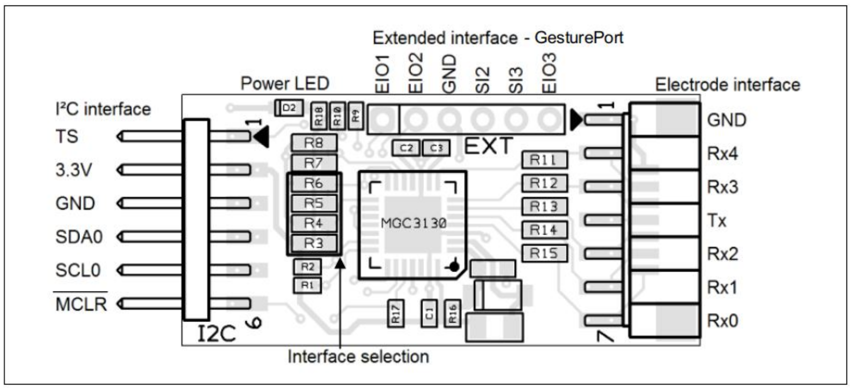


Figure 20: MGC3130 Controller Layout

As can be seen from the layout the unit is positioned at the center of the PCB and it can be connected with the external world through three interfaces:

1. The electrode interface on the right; this interface is a 7-pin 2mm connection to the electrode PCB and it carries the 6 signals coming from the electrode (one pin goes for the ground GND while the other five pins go on for each of the electrodes embedded into the panel) and one signal coming from the MGC3130 controller itself that is the TX signal. This signal is basically the power source that the controller gives to the electrode to make it work properly.
2. The I2C interface on the left; this interface is a connection to the external world through a 6-pins 2mm I2C connector: the pins intended as input are the **3.3V** and the **GND** pins that must provide power to both the controller and the electrode PCB; the other 4 pins will be used by both the MGC3130 (acting as a slave) and the Host PC (acting as the Master) during the data exchange process made through I2C protocol with two available slave addresses **0x42** and **0x43**.
3. The last interface is the one on the upper side of the controller and is intended to send gesture results directly to the external world but int this case this last interface was not used.

As can be easily spotted out by the number o the pins on the left interface, the I2C interface implemented for this sensor it's not a classical I2C one cause it adds the TS line as a synchronization signal. To better understand the difference between the two versions it's advisable to make a brief explanation of the standard I2C protocol and then to make a comparison with the two implementations.

The standard I2C protocol is intended for bidirectional communications between two devices; one the the two devices will act as the *Master* of the connection while the other one will act as the *Slave*. The behavior of the master is to rule the message flow on the connection; in fact the slave device cannot send any I2C message unless it has been asked by the master, while the master is free to send any message when needed. Since on the same bus may be connected multiple I2C devices, each of them must be identified by an identifier named **address**. The MGC3130 can be for example found on the same bus at two different addresses.

The standard signal used by this protocol are:

1. **SDA** which is the line used to transmit the data;
2. **SCL**, the line used to transmit the clock pulse;
3. **GND**, the pin going to ground;
4. **VCC**, bringing voltage (usually 3.3V) and connected to SDA and SCL line through pull-up resistors.

Since the master device is the one ruling the data flow on the connection it may need to do perform a write or a read operation. In order to begin each one of them the bus status must be in the **IDLE** condition i.e. both the SDA and the SCL must be in **high** status.
The *write cycle* operations are:

1. master sends a START condition to the slave;
2. master sends data to the slave;
3. master sends the STOP condition to the slave;

Once defined the START condition as a *high-to-low* transition of SDA with SCL high and the STOP condition as a *low-to-high* transition while the the SCL signal is high, it is possible to represent graphically the write cycle:

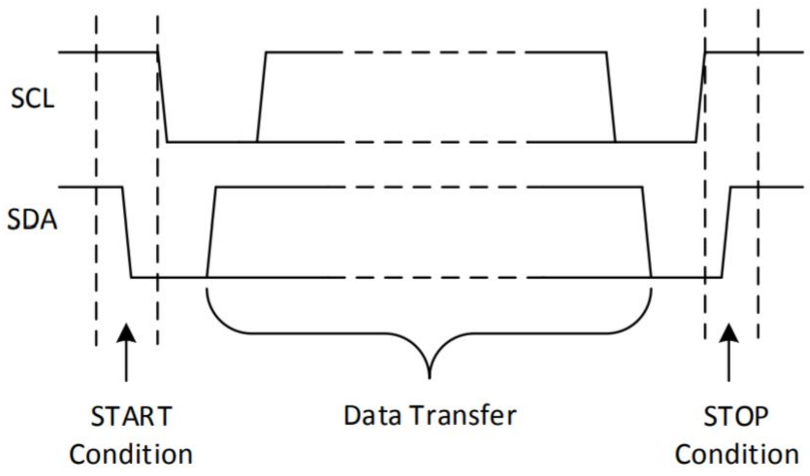


Figure 21: I2C write cycle

The read cycle instead is structured as follows:

1. master sends a START condition to the slave;
2. master sends to the slave the address of the register that it needs to read;
3. master receives data from the slave;
4. master sends the STOP condition to the slave.

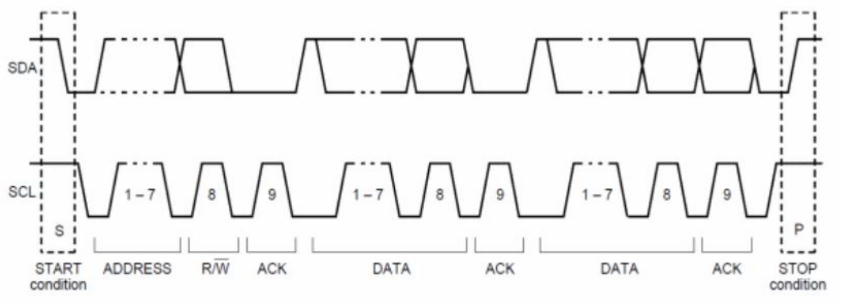


Figure 22: I2C read cycle

As can be noticed from the image each byte must be followed by a single bit sent by the receiver device; this bit named **ACK** is used to communicate to the sender that the byte has been received properly, thus after each byte sent the master has to release the bus in order to receive the ACK/NACK bit.

The MGC3130 I2C interface essentially extends the standard connection by adding the TS signal. In particular this line is used to synchronize the data transfer between the master and the slave controller in some particular cases; since the controller is provided with power saving mode it is necessary to use the TS line in order to communicate the master when new data is available. This allows:

- to inform the host when new data is available from the controller;
- to let the updates of the controller buffer finish before starting the communication.

This extension allows a very low percentage of lost or corrupted messages.
Since the TS line must be used to communicate a status it has to be shared between the master and the slave; the table defining the possible conditions on TS it's the following:

MGC3130	**Host PC**	**TS**	**Condition**
released H	released H	H	Host has finished reading and the MGC3130 has sent all the available data; the MGC3130 can update it's buffer.
asserted L	released H	L	New data available from the MGC3130 but he Host has not started reading; after a timeout of 5ms in this condition the controller releases the line high while updating the buffer.
asserted L	asserted L	L	Host started reading; the MGC3130 won't update the buffer until the host has finished
released H	asserted L	L	MGC3130 needs to update the data but the Host is still reading.

Table 1: MGC3130 possible TS line conditions

According to the conditions defined by the MGC3130 I2C protocol the cycles will be a little bit different; the read cycle will be structured as:

1. master waits until TS line is L;
2. when TS line is 0 the master asserts L the TS line as well;
3. master sends a START condition to the slave;
4. master sends to the slave the address of the register that it needs to read;

5. master receives data from the slave;
6. master sends the STOP condition to the slave;
7. master releases the TS line.

The write cycle instead remains the same because during a write operation there's no need to manage the TS line.

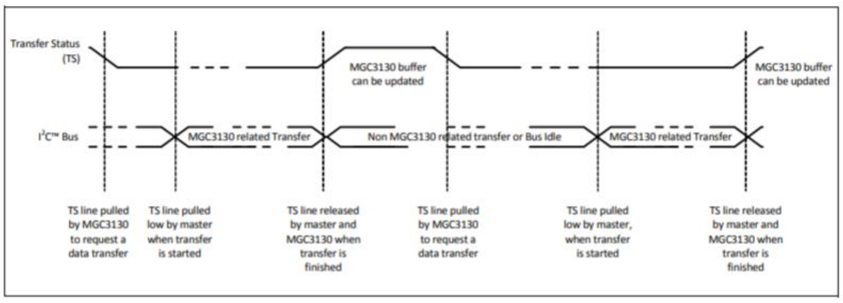

Transfer Status (TS)
MGC3130 buffer can be updated
MGC3130 buffer can be updated
I²C™ Bus
MGC3130 related Transfer
Non MGC3130 related transfer or Bus Idle
MGC3130 related Transfer
TS line pulled by MGC3130 to request a data transfer
TS line pulled low by master when transfer is started
TS line released by master and MGC3130 when transfer is finished
TS line pulled by MGC3130 to request a data transfer
TS line pulled low by master, when transfer is started
TS line released by master and MGC3130 when transfer is finished

Figure 23: GestIC I2C protocol

The last pin is the $\overline{MCLR}$; this signal basically acts like a negated reset thus the a High level will have no effects on the controller while a Low value on this line will reset the MGC3130 Unit.

3.2.2 Reference Electrode

The Kit is also provided with a reference electrode panel actually represents the sensor within the system; this component is responsible to sense moving object position variation and accordingly send those informations to the MGC3130 controller in the form of electrical signals. This is achieved exploiting the **Electrical Near-Field Distortion** phenomenon.
Basically the GestIC controller, intended as the union of the MGC3130 and a reference electrode designed correctly, exploits the E-Field (electric field) variation due to the presence of an object to compute the estimated position of that object in a 3D space around the reference electrode itself.

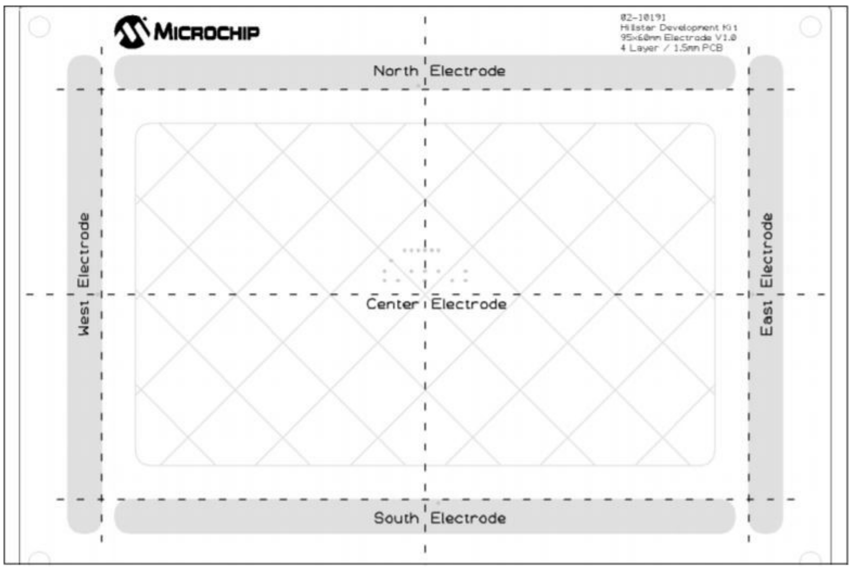


Figure 24: Reference Electrode layout

When powering the electrode with a DC voltage the E-Field generated by the panel will be static; on the other hand if and AC voltage is applied the the charges onto the reference panel will not be constant but will vary in time instead. The variation of the electrical charge will generate electromagnetic waves that, in vacuum, will propagate with a velocity equal to the speed light:

$$\lambda = f * c$$

being λ the wavelength of the generated wave, f the frequency and c the speed of light.
However when the wavelength of the electromagnetic wave generated is much larger than the geometry of the generating electrode the magnetic component will be so small to not being considered. the MGC3130 can be configured to use one between five different transmission frequencies corresponding to 115KHz, 103KHz, 88KHz, 67KHz and 44KHz from the highest to the lowest. When in the worst case the controller uses a TX signals varying with the highest frequency of 115KHz the resulting wavelength will be a value greater than 2.5Km; it is thus possible to assert that the electromagnetic component of this device will be almost null by the fact that the reference electrode , measures in the order of centimeters. As a result the E-Field near the panel is almost static and can be monitored to sense the presence of a conductive objective.

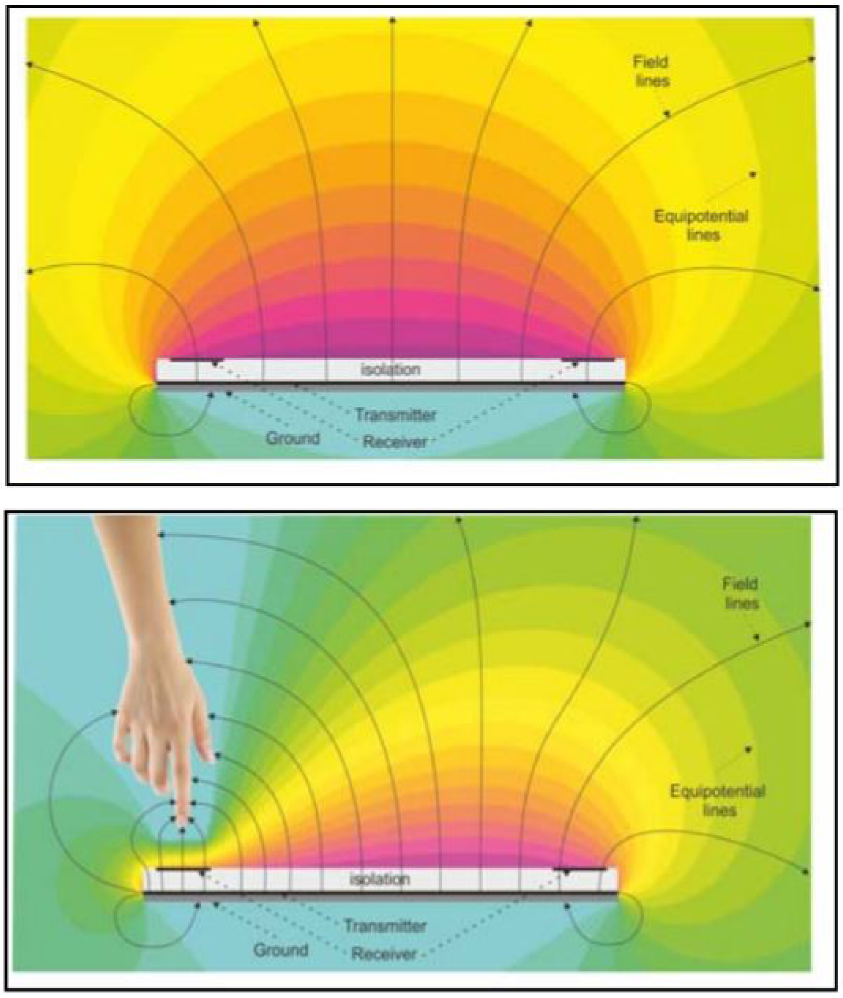


Figure 25: E-Field Distortion caused by human hand

In particular the human body represents a conductive object; thus when a hand will pass

through the near E-field of the electrode the filed itself will be variated since a part of the electric field will be drawn by the hand. In the position of the space where the object is located the E-field will be locally smaller then everywhere else within the electrode space.
The different variations of the E-field for the four electrodes disposed as the cardinal points on the board will allow the controller to compute the position of the object as three coordinates x, y, z.
In order to achieve the proposed result the reference electrode has to be design compliant in the shape and component position; the panel shall in fact be characterized by a multi layer structure alternating conductive and not conductive material and should embed at least for electrodes positioned as the cardinal points (the central electrode is optional and can be included to perform a more precise estimation of the position). The reference electrode should be based on the following structure:

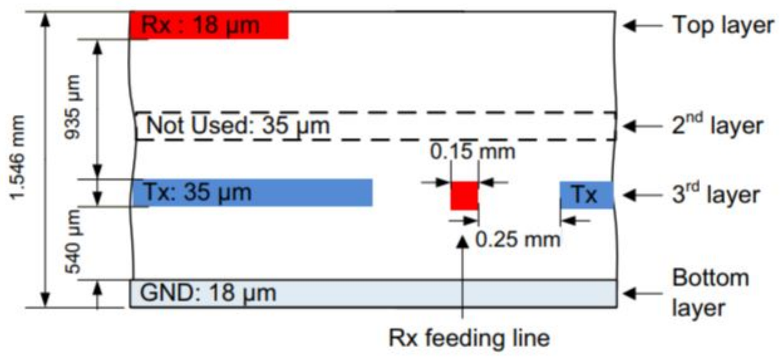


Figure 26: PCB layers disposition

The five receiving electrodes Rx are positioned on the upper face of the PCB while the Tx signal passes beneath the surface isolated by the first layer and designed as cross-hatched to reduce capacitive effect between separate layers; the layer at the bottom can optionally be connected to ground to isolate the upper layers and thus the sensing area from the components that may lay under the GestIC.

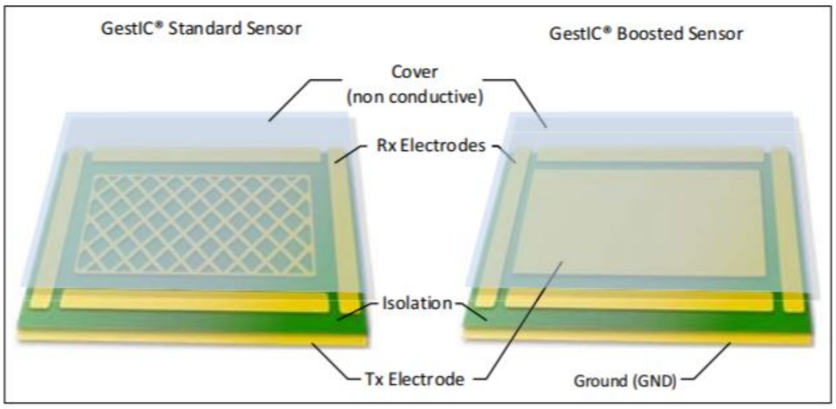


Figure 27: PCB different versions

The panel provided with the Kit has a rectangular shape, however other shapes can be chosen during the electrode design phase; on of them could for instance be a circular shape. The measures of the the PCB are provided by the vendor and will be useful while setting up the controller configuration during the calibration phase.

The measures are summarized within the following figure:

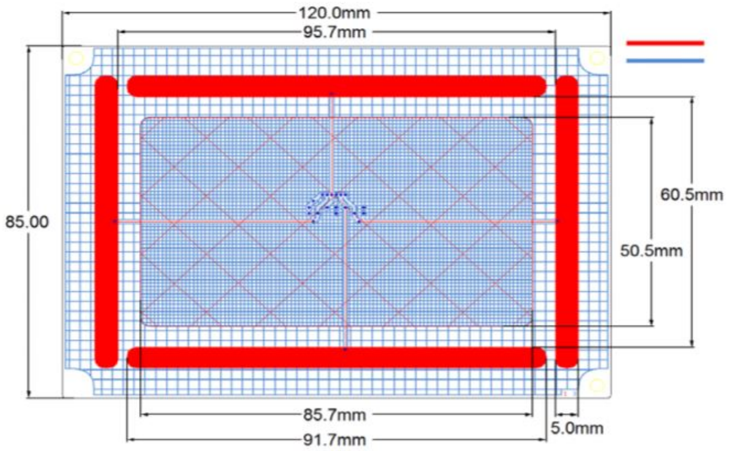


Figure 28: Hillstar reference electrode geometry

where the receiving electrodes Rx are indicated in red while the center electrode is represented by the cross-hatched area.

3.2.3 USB to I2C Bridge

The last component that can be found within the package is the Micro USB type B to I2C adapter; this bridge embed a micro controller that is intended to retrieve data from the MGC3130 and send the data to the Host PC. Moreover it regulates the 5V voltage incoming from the PC USB port changing its value to a 3.3V MGC3130 compliant input voltage.

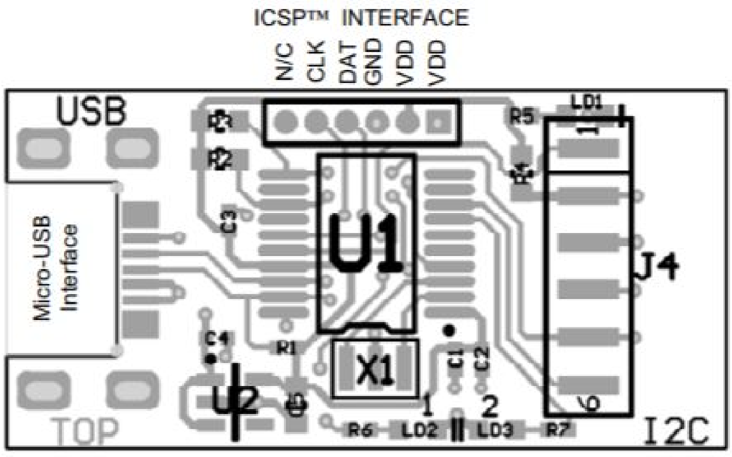


Figure 29: USB to I2C bridge

This device has been useful during the preliminary phases of the project when a parametrized firmware and calibrated parameters had to be flashed into the stock MGC3130 controller. To fulfill this purpose the Kit comes out with an evaluation software named **Aurea** that can be found at `http://www.microchip.com/DevelopmentTools/ProductDetails.aspx?PartNO=dm160218` within the *Documentation and Software* section; the guided initialization of the controller can be made quite easily through this application thus no particular words will be expended on the parametrization field. Once the initialization of the controller will be finished a test bench will be offered by the same application in order to test the correct functionalities of the controller.

3.2.4 Other components

Other devices used to develop the project are

- a **USB to TTL adapter**; this device can be used to convert a USB ports into a TX-RX serial line that can be used to be connected directly with the board UART serial ports on the pinout banks;
- a **Ethernet cable;**
- a **micro-USB cable;**
- a **micro SD card** with a storage capacity of at least 8GB;
- wires and connectors.

4 Driver Development

This section shows the process that brought to design a Linux Embedded driver for the GestIC controller in the form of a loadable Kernel module. Thus in this chapter will be explained the theory behind Linux Kernel modules and devices abstraction, the source code snippets of the driver itself, the methods used to debug the software and the integration tools offered again by the Yocto Project.

4.1 Linux Drivers Generalities

In general it is possible to define a **driver** as the piece of software allowing to make use of a certain device; devices can be in general of different natures, the most common during everyday life are devices connected through bluetooth modules or USB devices. Under each of the devices connected to a PC there's a certain driver that make possible the communication between the OS and the device itself.
Linux OS offers two different ways of developing and maintaining a driver; the driver code can be in fact embedded within the kernel itself or developed as a user space module. The two approaches have of course their own advantages and drawbacks.

4.1.1 Kernel Drivers

The **Kernel drivers**, are directly embedded within the Kernel; those drivers can also be referred as *in-tree* drivers since they are embedded within the **Device Tree** of the system. The Device Trees are essentially data structures that describe the hardware configuration for a certain machine so that the Kernel can manage a correct usage of those components; the *tree* contains information about each piece of hardware composing the board spanning from the CPU to the memory, from the I/O devices to the network components. This data structure presents the most typical tree layout in computer science being composed by nodes having properties, representing the system components and their characteristics.
The device tree composition must be stored somewhere within the system since at any time it must be a resource reachable by the Linux OS Kernel; there are essentially two possibilities to store the Device Tree. The first one is to statically hard code the hardware composition tree into the (or PC) firmware being flashed on the board; this approach is however intended for a system whose hardware won't be modified often since every change within the hardware composition will cause the necessity for an new firmware with an updated Device Tree binary. The second more flexible approach, that is the one adopted by Linux Embedded OS, is to compile an updated version of the Device Tree called **Device Tree Blob** (**DTB**), a binary file representing the hardware components of the system that is passed by the bootloader to the Kernel during the booting phase. The generation of the starts from the compilation of Device Tree Sources (**DTS**); upon any changes of the hardware will be thus possible to recompile just a new DTS files instead of the full Kernel software allowing to save a lot of time.
The Device Tree stores some useful informations; from any system running Linux those informations can be accessed from the contents of the */proc/device-tree* folder. For example it is possible to obtain the model of the board by running:

```
root@udooneo:# echo $(cat /proc/device-tree/model)
UDOO Neo Full
```

or

```
root@udooneo:# find /proc/device-tree/
```

to print out the full list of the components within the device tree.
The following figure summarizes the basic concepts on the creation od a device tree:

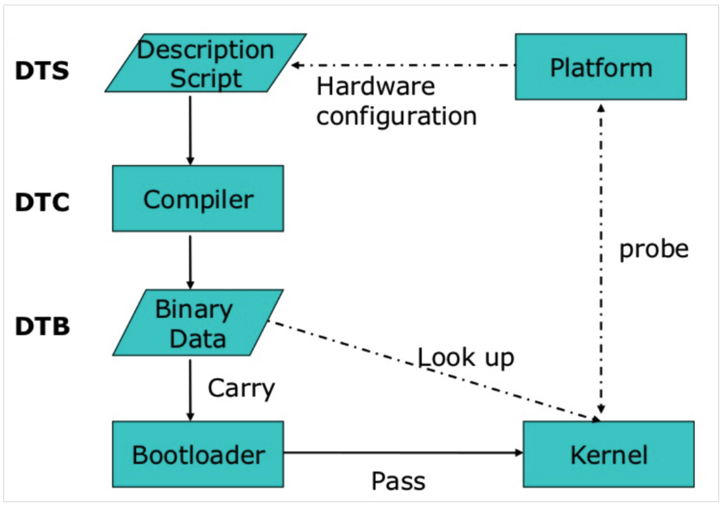


Figure 30: Device Tree work-flow

The overhaul advantages of choosing a Kernel device drivers will be related to the great support that the community gives to improve and maintain the Linux Kernel; moreover the distribution will obviously be easier by choosing the community channel.

4.1.2 User Space Drivers

The other possibility offered by the Linux Embedded OS is to develop a device driver at user space level and to plug or unplug it when needed; this kind of device drivers is defined as **out-of-tree device drivers** since this software is not directly embedded within the Kernel and thus the device for which the driver is intended is not listed into the device tree.
Since this has been the strategy adopted to develop the MGC3130 driver it is interesting to point out all the advantages and the drawbacks related to this particular choice. The main advantages w are:

- the driver can be written in any language (in this case the choice has been C language);
- the driver won't become part of the community and thus it could sold;
- no risks of Kernel crash if something is wrong with the driver since it runs within the user space;
- if the driver crashes it can be simply removed and re-installed;

On the other hand the main drawbacks are related to performance.

- lower performance in terms of speeds with respect to the Kernel drivers due to the necessity of more system calls;
- greater interrupt latency.

4.1.3 Virtual File System

One of the most useful properties of Linux and in general of Unix-based OSs is that they allow an easy access to the devices through a *device abstraction* known as **Virtual File System** or VFS; thanks to this feature it is possible for any application running at the user space level to access different type of devices with the same file-access API functions (like open, read, write, close...) thus not considering the real nature of the device in case. This feature is achievable because the VFS takes also in charge to establish an interface between the user space call to a file-accessing function and the actual implementation of that function within the driver, weather it is a Kernel Driver or a Device Module Driver.
Within the next image is shown a representation of the VFS layer embedded into the multi-layer structure of the Linux Embedded OS:

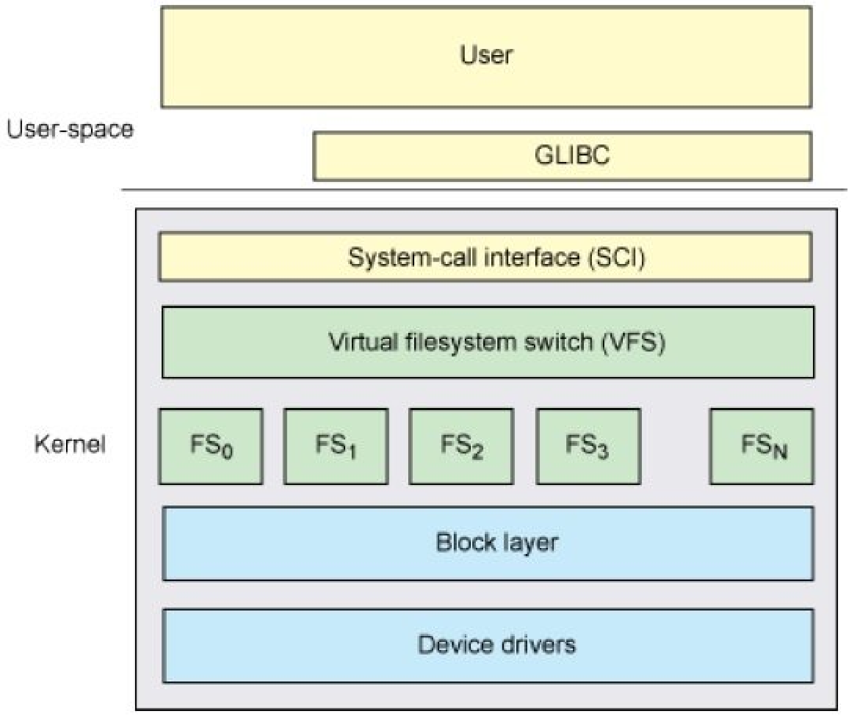


Figure 31: VFS layer logical position

4.2 Preparing the Yocto environment

Before moving on to the real driver implementation it is necessary to setup a new build environment for the Yocto Project in order to leave the original base-build produced in the previous chapter untouched; thus a new build project has to be created; inside this build it will be possible to find the device modules that will be installed on the UDOO Neo Development Board. Even if it's not necessary to use the Yocto Project tools in order to produce a device module, the process necessary to achieve the same result may not be as much straightforward as the one offered by the Yocto Project tools and Bitbake environment. To prepare a new build run the following commands:

```
$ cd thesis/udoo-community-bsp
/thesis/udoo-community-bsp$ MACHINE=udooneo DISTRO=poky source ./setup-environm
ent build-thesis
```

These command will create a new build folder named **build-thesis** alongside the folders created during the previous chapters; as done for the first build folder it is necessary to edit the configuration files inseide the main build directory. In order to configure the build environment correctly the **local.conf** file inside the conf folder should look as follows:

```
/thesis/udoo-community-bsp$ cat build_thesis/conf/local.conf
MACHINE ??= 'udooneo'
DISTRO ?= 'poky'
PACKAGE_CLASSES ?= "package_rpm"
USER_CLASSES ?= "buildstats image-mklibs image-prelink"
IMAGE_FEATURES = "debug-tweaks tools-debug eclipse-debug"
IMAGE_INSTALL_append = " tcf-agent openssh-sftp-server "
PATCHRESOLVE = "noop"
BB_DISKMON_DIRS = "\
STOPTASKS,${TMPDIR},1G,100K \
STOPTASKS,${DL_DIR},1G,100K \
STOPTASKS,${SSTATE_DIR},1G,100K \
STOPTASKS,/tmp,100M,100K \
ABORT,${TMPDIR},100M,1K \
ABORT,${DL_DIR},100M,1K \
ABORT,${SSTATE_DIR},100M,1K \
ABORT,/tmp,10M,1K"
PACKAGECONFIG_append_pn-qemu-native = " sdl"
PACKAGECONFIG_append_pn-nativesdk-qemu = " sdl"
CONF_VERSION = "1"

DL_DIR ?= "${BSPDIR}/downloads/"
SSTATE_DIR ?= "${BSPDIR}/udooneo_poky_udoo-image-full-cmdline/sstate-cache/"
ACCEPT_FSL_EULA = "1"
```

As can be spotted out by running the command

```
/thesis/udoo-community-bsp$ diff udooneo_poky_udoo-image-full-cmdline/conf/loc
al.conf build_thesis/conf/local.conf
21a22
> SSTATE_DIR ?= "${BSPDIR}/udooneo_poky_udoo-image-full-cmdline/sstate-cache/"
```

the output of the *diff* command shows how the only change that has been made between the previous version of the file local.conf and the ne one is simply the presence of a new line. Since the **SSTATE_DIR** was not explicitly declared in the first build version the default path for the **sstate-cache** directory is within the build directory itself; in this case the path ha been set to point a the previous version of the sstate-cache folder and thus the sstate-cache within the the "udooneo_poky_udoo-image-full-cmdline" folder. This choice has been made to speed-up any future build based on the previous one; in fact thanks to the files found within the sstate-cache folder the process will skip all those processes that don't need to be re-executed.
The following step has been the creation of a custom metadata layer intended to contain the driver source code; keeping things separated within different recipes and layers allows te build being very modular and easy to customize. Moreover including the recipe for the driver within a Yocto compatible layer has allowed to use the Bitbake program to compile the driver source code when needed. Even if it is possible to create a new layer by manually copying the standard Yocto layer structure from another layer, it is possible to create a new one using one of the commands that the Yocto environment has predisposed. To create new layer resume, if needed, the new build and create the folder containing the custom resources:

```
/thesis/udoo-community-bsp$ mkdir mysources
```

Now from inside the *mysources* folder create a new layer by typing:

```
/thesis/udoo-community-bsp/mysources/$ yocto-layer create thesis
Please enter the layer priority you'd like to use for the layer: [default: 6]
Would you like to have an example recipe created? (y/n) [default: n] y
Please enter the name you'd like to use for your example recipe: [default: exam
ple]
Would you like to have an example bbappend file created? (y/n) [default: n] n

New layer created in meta-thesis.

Don't forget to add it to your BBLAYERS (for details see meta-thesis/README).
```

The output shows how during the command has been created a new layer with a default priority of 6, with an example recipe default named *example* and with no example bbappend files.
As can be seen in the following figure

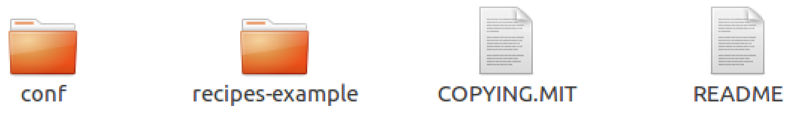


Figure 32: mysources folder after creation

the folder *mysources* has now been set up as a metadata layer with its own configuration folder and an example recipe. After renaming the *recipes-example* directory in **recipes-kernel**, it is possible to configure the layer.conf configuration file that can be found into the *conf* folder to look as follows.

```
# We have a conf and classes directory, add to BBPATH
BBPATH .= ":${LAYERDIR}"

# We have recipes-* directories, add to BBFILES
BBFILES += "${LAYERDIR}/recipes-*/*/*.bb \
${LAYERDIR}/recipes-*/*/*.bbappend"

BBFILE_COLLECTIONS += "meta-thesis"
BBFILE_PATTERN_meta-thesis = "^${LAYERDIR}/"
BBFILE_PRIORITY_meta-thesis = "6"

#Added the kernel modules for the out-of-tree extra devices
MACHINE_EXTRA_RRECOMMENDS += " gestic-mod"

# Adding a run-time recommendation for the out-of-tree modules (the absence sho
uld not be blocking)
MACHINE_ESSENTIAL_EXTRA_RRECOMMENDS += " kernel-module-gestic-mod"
```

The layer.conf file has essentially been modified by adding the between the machine source file the gestic-mod module device that it'going to be illustrated within the following sections of this paper.
The next step is tho change the name of the *example* folder inside the recipes-kernel directory as gestic-mod, being compatible with the name assigned within the layer.conf file. After renaming

the folder get inside it and run the following commands to rename the folder and the recipe found within the gestic-mod folder:

```
mv example-0.1/ files
```

to rename the folder, and

```
mv example_0.1.bb gestic-mod.bb
```

to rename the recipe for the device module. This recipe is the one that it's going to be run as a Bitbake target when compiling the driver source code.
The **gestic-mod.bb** has been edited to look as follows:

```
DESCRIPTION = "Linux kernel module for the gestic module"
LICENSE = "GPLv2"
LIC_FILES_CHKSUM = "file://COPYING;md5=12f884d2ae1ff87c09e5b7ccc2c4ca7e"

inherit module

COMPATIBLE_MACHINE = "udooneo"

SRC_URI = "file://Makefile \
file://gestic.c \
file://COPYING \
"
S = "${WORKDIR}"
```

As can be noticed the recipe inherits the module class and thus will be executed accordingly by Bitbake; moreover the *SRC_URI* variable contains the file names of the files that will be contained inside the renamed **files** folder. To complete this step create a **Makefile** looking as follows:

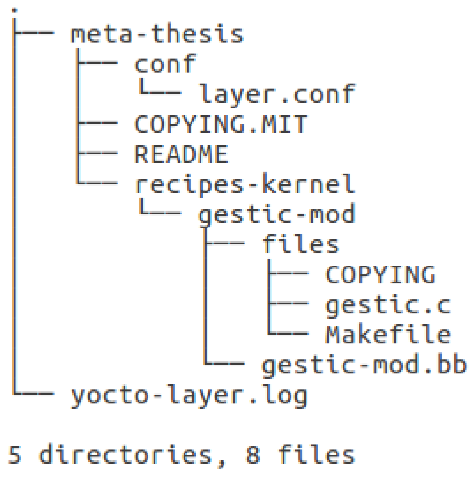
```
obj-m := gestic.o

SRC := $(shell pwd)

all:
$(MAKE) -C $(KERNEL_SRC) M=$(SRC)

modules_install:
$(MAKE) -C $(KERNEL_SRC) M=$(SRC) modules_install

clean:
rm -f *.o * core .depend .*.cmd *.ko *.mod.c
rm -f Module.markers Module.symvers modules.order
rm -rf .tmp_versions Modules.symvers
```

This Makefile will instruct the process on how to compile the driver source code and about the target location for the module installation.
Now a license should be added to this module; for this project has been used a GPLv2 license (General Public License) for free software sharing. The license file to be included can be copied from the *sources* and renamed by running the following command from the *files* folder:

```
cp /home/user/thesis/udoo-community-bsp/sources/poky/meta/COPYING.GPLv2 ./
mv COPYING.GPLv2 COPYING
```

To terminate the setup of the folder *files* it's eventually necessary to add the file that will contain the source code to compile the driver; since the chosen language to write the driver has been C language create a file named **gestic.c**. The file just created will contain the source code for the driver and the VFS interface functions and will be the file to be edited during the following section.

Before moving onto the driver design the directory tree of *mysources* folder looks as follows:



```
.
├── meta-thesis
│   ├── conf
│   │   └── layer.conf
│   ├── COPYING.MIT
│   ├── README
│   └── recipes-kernel
│       └── gestic-mod
│           ├── files
│           │   ├── COPYING
│           │   ├── gestic.c
│           │   └── Makefile
│           └── gestic-mod.bb
└── yocto-layer.log

5 directories, 8 files
```

Figure 33: mysources tree after layer setup

To conclude the setup it's necessary to add the newly created layer between the build layers of the build; thus the **build_thesis/conf/bblayers.conf** file was modified to look as follows:

```
POKY_BBLAYERS_CONF_VERSION = "2"

BBPATH = "${TOPDIR}"
BSPDIR := "${@os.path.abspath(os.path.dirname(d.getVar('FILE', True)) + '/../..
')}"
EXTDIR := "/home/fabio/thesis/udoo-community-bsp"

BBFILES ?= ""
BBLAYERS = " \
${BSPDIR}/sources/poky/meta \
${BSPDIR}/sources/poky/meta-poky \
${BSPDIR}/sources/poky/meta-yocto \
${BSPDIR}/sources/poky/meta-yocto-bsp \
\
${BSPDIR}/sources/meta-openembedded/meta-oe \
${BSPDIR}/sources/meta-openembedded/meta-multimedia \
${BSPDIR}/sources/meta-openembedded/meta-python \
${BSPDIR}/sources/meta-openembedded/meta-networking \
\
${BSPDIR}/sources/meta-qt5 \
\
${BSPDIR}/sources/meta-freescale \
${BSPDIR}/sources/meta-freescale-3rdparty \
${BSPDIR}/sources/meta-freescale-distro \
${BSPDIR}/sources/meta-udoo \
\
${EXTDIR}/mysources/meta-thesis \
```

As can be noticed a new variable named **EXTDIR** has been created; even if in this case the BSPDIR and the EXTDIR variables bring to the same path, the variable has been added in order to maintain a *modular* approach a forecasting that, in case of a folder reposition only one path should have been changed inside the whole configuration file. As the name may suggest the EXTDIR variable has been intended as the folder containing custom source code to be added to the build process.

4.3 Module Device Driver Implementation

Once the Yocto environment has been configured correctly to include within the build process the metadata layer created to host the module device driver for GestIC controller it is possible to comment the driver implementation contained in the source file **gestic.c** created during the previous section. The driver has been, as suggested by the extension of the file, been written in C; the choice of this language has been motivated essentially by the huge existent documentation regarding driver development in C.
The driver will be explained with code snippets of the different parts of the file, that can be in fact divided in three parts:

1. *definitions and inclusions*;
2. *initialization and disposition* of the resources;
3. *file operations* implementation;

It also needed to point out the all the libraries that have been included within the driver can be found under the *tmp* folder containing Bitbake output files since those libraries are the result of source code fetching of the Bitbake process itself; in particular the most of them can be found and consulted within the path *udoo-community-bsp/build_thesis/tmp/work-shared/udooneo/kernel-source/include/linux/* that will be referred as **linux/** folder from now on. The following code snippets will show the most important parts of the code.

4.3.1 Initializzation and Disposition functions

The driver development started with the implementation of the initialization and disposition functions; those two functions named **gestic_init** and **gestic_exit** respectively, are the function executed by the Kernel when each time the module is installed or removed from system. They essentially must take care of initializing all the required hardware resources, as for instance the I2C port needed to communicate with the controller or an interrupt, and the structures (whose implementation can be found within the included libraries) that are used by the module functions.
The initialization function is the following:

```
/*The function which allocates all the needed resources for the device driver*/
static int __init gestic_init(void)
{
if (DEBUG) printk(KERN_INFO "\n\nGestIC: the module is in DEBUG mode\n");
...
```

As can be noticed by the looking at the function definition, the *gestic_init* function (and the *gestic_exit* function as well) are marked through the *__init* and *__exit* macros as an initialization and a disposition function so that the Kernel will know which function to execute during each one of the two operations. In this particular case the gestic_init function has been marked as the entry point for the Kernel execution of the module. Moreover the snippets shows the use of the **DEBUG** parameter that was used to debug the module; since the module execution is taken in charge by the Kernel itself the most immediate way, that was adopted to test the gestic module, is to print some important information during the code execution in order to understand if the driver is working properly or not. Linux offers through the module parameters feature, a way to extend the the device module installation with options; the module parameter has been initialized as:

```
...
//The macro used to check the boolean parameter of this module; this parameter allows to enable or not
//a very verbose debug mode that prints on the kernel standard output a walkthrough log of the
//module functions
#define DEBUG (gestic_debug)
...
/*
* MODULE PARAMETERS
*/

//The parameter used to enable a debug mode
static bool gestic_debug = 0;
module_param(gestic_debug, bool, 0);
MODULE_PARM_DESC(gestic_debug, "Enables the debug mode on kernel output");
...
```

It is important to underline that the *DEBUG* parameter, that is basically a boolean value, is initialized as *false* by default; thus installing the module without specifying the DEBUG option will result in a non-debug code execution; this choice has been made in order to let the code as clean as possible in case of non-debug execution since, even if in a small percentage, even the most basic print operation may vary the module behavior in cases of fast real-time application as this one. In case of DEBUG mode the output will be printed on the Kernel log with a low priority (KERN_INFO); it is advisable in fact to leave to system errors greater priorities as *KERN_ERR*.

The *gestic_init* function has been structured as a single sequence of nested conditional statements; in this way is possible to stop the initialization of the module whenever one the operations required to complete this phase does not succeed and to immediately clean up the resources allocated until the failure happened. The structure looks as follows:

```
...
//Initializing the queue for reading processes
init_waitqueue_head(&read_queue);

//Registering the new character device numbers
//Parameters: device, first minor number for the moudule, count of needed
//minor numbers, name of the associated device
//Returns a negative value in case of error
if (alloc_chrdev_region(&gestic_dev, 0, 1, "gestic") >= 0)
{
if (DEBUG) printk(KERN_INFO "GestIC: character device numbers registration succeded\n");
//Creating the class structure for the device; needed to register the device in
//sysfs
//parameters: module owner of the class, name of the class
if ((cls = class_create(THIS_MODULE, "chardrv")) != NULL)
{
...
}
else if (DEBUG) printk(KERN_INFO "GestIC: class initialization failed\n");
unregister_chrdev_region(gestic_dev, 1);
}
else if (DEBUG) printk(KERN_INFO "GestIC: character device numbers registration failed\n");
return KERNEL_ERROR;
}
```

The previous snippet points out the design just described; for instance if the *class_create* function returns an error code the function won't nest within the successive *if-statements* but will stop instead unregistering the allocated character device (since the alloc_chrdev_region was the only function executed correctly until the failure time). If everything goes well and all the resources are initialized correctly the inner *if-statement* is reached:

```
...
if (DEBUG) printk(KERN_INFO "GestIC: initialization completed\n");
printk(KERN_INFO "GestIC device: driver installed\n");

return NO_ERROR;
...
```

The main resources allocated and initialized by the gestic_init function are:

- a *waitqueue* used to manage the read requests from the user space according to the availability of new data from the sensor;
- a *class* structure for the device; the device can be configured as a character device oa block device. The class choice will modify way the file is accessed: if the file is *character-based* the access to the corresponding memory will be a sequential iterative access to the characters composing the file; on the other hand if the file is *block-based* the access will be granted for portions of the file memory that are multiple of the block size (usually **512Kbyte**). In this particular case the module has been classified as **character-based**;
- a *device* structure containing some information about the device: a structure containing information on this module will be added inside the **sysfs**, a pseudo-file system containing user space information about kernel subsystems that is mounted under */sys* folder;
- the *file operation* structure (**fops**) is connected to the device;
- a *I2C data structure* that will be used to establish a communication with the controller on the I2C bus of the board;
- two *GPIOs* used to manage the *TS* and the $\overline{MCLR}$ line. The TS line has been initialized as an input GPIO while the reset line has been initialized as output as it will always be a line ruled by the master;
- an *interrupt* used to manage the TS line changes;

On the other hand the *gestic_exit* function takes in charge of freeing the allocated resources when the module device is removed:

```
/*The function which releases all the resources used by the device driver*/
static int __exit gestic_exit(void)
{
free_irq(ts_irq_num, NULL);
gpio_free(GESTIC_GPIO_MCLR);
gpio_free(GESTIC_GPIO_TS);
i2c_unregister_device(gestic_client);
cdev_del(&cdev);
device_destroy(cls, gestic_dev);
class_destroy(cls);
unregister_chrdev_region(gestic_dev, 1);

printk(KERN_INFO "GestIC device: driver removed\n");

return 0;
}
```

In order to define the two described functions as respectively an entry and an exit point for the module it necessary to add the following directives within the module:

```
...
module_init(gestic_init);
module_exit(gestic_exit);
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Fabio Garcea <fabio.garcea@gmail.com>");
MODULE_DESCRIPTION("GESTIC Driver for UDOO-Neo");
```

As can be noticed the *module.h* library that can be found under the *linux*
folder offers the possibility to add some other module informations as a DESCRIPTION or the AUTHOR contacts.

4.3.2 File Operations

After the initialization and disposition phases management the successive part to be developed was the one containing the *file operations*; as explained these operations act as translation for the *VFS* that makes an association between the high level file-access abstraction and the low-level driver implementation. The structure containing the file-operations looks as follows:

```
/*The struct defining the correspondance between low-level and high-level functions used
to interac twith the device*/
struct file_operations gestic_fops =
{
.owner = THIS_MODULE,
.open = gestic_open,
.release = gestic_close,
.read = gestic_read,
.write = gestic_write,
};
```

The driver provides a particular implementation for each of the most basic operations that can be made on a file. The operations that have been implemented are:

- the *open* function, used to open the device;
- the *close* function, used to close the device;
- the *read* function, used to read from the device;
- the *open* function, used to write to the device;

These function are sufficient in this driver implementation to give full access to the device. The following figure tries to represent in a graphical way the behavior of the driver during both a read or a write operation. If a write is needed from the user space the operation will be executed immediately since there's no need to wait for the TS line edge; if a read shall be performed however the operation shall be tried only upon a falling edge of the TS line (the GestIC module assert Low the line to signal a data buffer update).
The first file operations that have been developed are the *open()* and *close()* functions.

```
...
/*Opens the GestIC device: this function is usually called once by the application level; this function
is basically used to configure the module in a known status and make it ready to be written or read*/
static int gestic_open(struct inode *i, struct file *f)
{
//Resetting the module and initializing the parameters each time the module is opened
gestic_reset();

if (DEBUG) printk(KERN_INFO "GestIC: module opened correctly\n");

return NO_ERROR;
}
...
```

The *gestic_open()* function basically takes only care of resetting the controller by calling another function named *gestic_reset()*:

```
...
/*Resets the MGC3130 controller*/
static void gestic_reset(void)
{
//Resetting initial values
write_length = 0;
read_length = 0;
//Releasing the MCLR line resets the controller; from the datasheet a 5ms period has to be waited
ASSERT_MCLR;
msleep(reset_asserted_delay);

//Asserting the MCLR line resets the controller; from the datasheet a 20ms period has to be waited
RELEASE_MCLR;
msleep(reset_released_delay);
...
```

The *gestic_reset()* simply manages the $\overline{MCLR}$ line through the two implemented macros *ASSERT_MCLR* and *RELEASE_MCLR*; the assert macro will pull the GPIO connected to reset signal low thus giving a reset command while the release will set the pin high to let the controller exit from the reset phase. Moreover after each of the two cases a delay time has been waited to be sure that any possible controller transient could be escaped. Since the resets represents a blank status of the device within the same function some variable re-initialization have also been performed.

The *gestic_close()* function on the other hand has nothing to do since no resources have to be closed neither cleaning operations have to be performed upon the device file closure.

Before moving on and start talking about the implementation of the write and read function it is necessary to point out the decision made in terms of hardware pinout: the UDOO Neo pins that have to be chosen are *six* just as the number of the pins of the MGC3130 controller. The choice for the GPIOs can be summarized with the following table:

PIN	TS	3.3V	GND	SDA	SCL	$\overline{MCLR}$
GPIO	105	/	/	/	/	148
PIN NAME	/	/	/	I2C2_SDA	I2C2_SCL	/
HEADER	J6, 8	J7, 3V3	J7, GND	J6, SDA	J6, SCL	J6, 9

Table 2: Summary of the pinout-controller connections

Thus some in-code definitions have been made accordingly:

```
...
//The two pins used for the the TS and MCLR signals
#define GESTIC_GPIO_TS 105
#define GESTIC_GPIO_MCLR 148

//The exadecimal number used to identify one of the two possibile addresses to establish a I2C connection
//with the board. The other one is 0x43 from the MGC3130 controller datasheet
#define GESTIC_I2C_ADDRESS 0x42

//From the board datasheet the bus number for the selected line is "2" but since the enumeration starts
//with "0" the corresponding line number is "1"
#define GESTIC_I2C_BUS_NUM 1
...
```

As can be noticed a constant storing the GESTIC_I2C_BUS_NUM has been declared; in fact the UDOO Neo board has, as pointed out within the *Hardware Reference* section, multiple I2C connections and thus multiple buses. The chosen I2C bus has been the bus number 2 but since the enumeration of the buses starts from "0" the I2C bus that the driver uses can be identified with the bus number "1".
At his point it is possible to explain the design adopted for the *write* and *read* file operations. The following figure gives a graphical representation to the *gestic_write()* function:

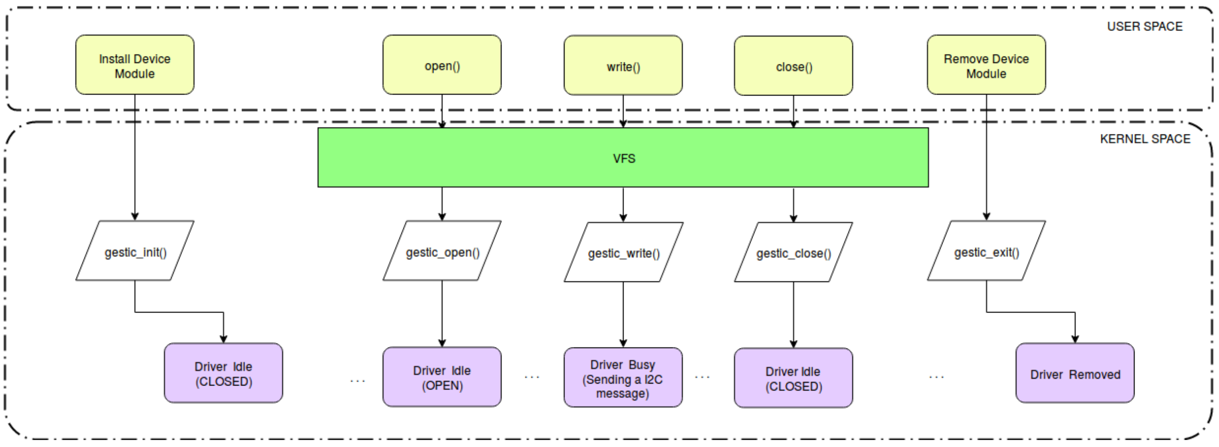


Figure 34: gestic_write() function scheme

An example of the module installation and disposition phases has been included; in fact all the operation on the device driver can only be executed by the Kernel if the module has been previously installed and opened.
The *gestic_write()* function has thus been developed taking into account the required design:

```
...
/*This function includes all the common operations to be done before sending the message
The message can be actually send through I2C
Future implementation: serial*/
ssize_t gestic_write(struct file *f, char __user *buf, size_t len, loff_t *off)
{
//The number of bytes succesfully written to the device
size_t bytes_written = 0;

//Setting the global variables
write_length = len;
if (write_length > MAX_MSG_LENGTH)
//Cropping the message to maximum allowed length
write_length = MAX_MSG_LENGTH;
//Resetting the whole write buffer before starting the transmission
memset(write_buffer, 0, MAX_MSG_LENGTH);
memcpy(write_buffer, buf, write_length);

bytes_written = gestic_I2C_write();

return bytes_written;
}
...
```

The function takes care to perform all the preliminary operations needed to prepare the variables before sending a new I2C message; the message coming from the user level is cropped to fit the maximum message length expected by the controller, it is copied inside a globally declare *write_buffer* (not risking to compromise the user buffer contents) and finally it calls the gestic_I2C_write() to send an I2C message to the controller working on the contents of the previously described buffer.
This subdivision of the jobs has been made forecasting a future extension of driver on other protocols different by the I2C protocol and selectable with more module parameters allowing a customization at installation time. The function is intended to interpret the contents of the buffer copied from the user space and understand weather it contains a command to be executed on the module or a message compatible with the GestIC controller message structure.
The driver allows in fact to execute two basic commands of *reset* and *sleep* on the device module by simply writing on it two predefined pattern (respectively **0x11** and **0x22**):

```
...
//Checking if the message received is a command; actually two commands are implemented:
//RESET, 0x11
//SLEEP, 0x22
if (CMD_RECEIVED)
{
if (DEBUG) printk(KERN_INFO "GestIC: command received\n");
//A command has been received; no message has to be sent, a command shall actually be applied
switch (CMD_ID)
{
case RESET_ID:
if (DEBUG) printk(KERN_INFO "GestIC: resetting...\n");
//Resetting the device
gestic_reset();
break;
case SLEEP_ID:
if (DEBUG) printk(KERN_INFO "GestIC: sleeping...\n");
//Making the module sleep
msleep(*sleep_time);
break;
default:
if (DEBUG) printk(KERN_INFO "GestIC: invalid command\n");
break;
}
bytes_written = write_length;
}
...
```

If the buffer doesn't contain any command the function sends, if the contents represent a valid message, the data inside the buffer to the controller returning the number of bytes successfully sent.
In order to understand how the driver implements the message validity check the GestIC controller message structure has to be briefly described first. Each message exchanged with the MGC3130 controller shall respect the following structure:

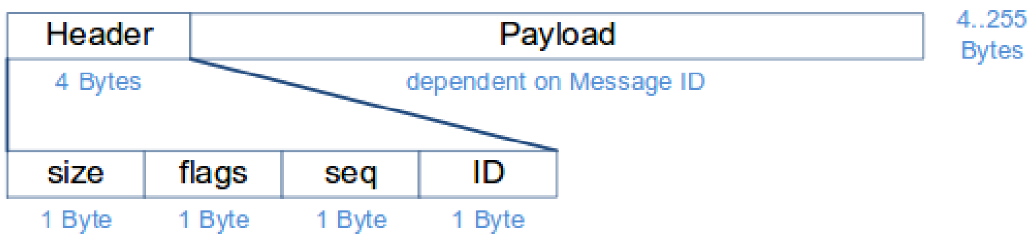


Figure 35: MGC3130 message layout

The first part of the message carries information about the message itself:

- a **size** byte, the size of the full message (header and payload);
- a reserved **flags** byte;
- a **sequence** byte that can be used to perform a check on the message loss;
- a **ID** byte that identifies the type of message; some of them will be explained during the next section when exploring the GestIC API.

The structure of the message header has been hard-coded within the driver and it is used to check the message validity during write operations through a macro named **VALID_MESSAGE** before eventually send the buffer contents to the controller:

```
...

//The macro used to check weather the message to be sent respects the format
//expected by the device
#define VALID_MSG ((write_header−>size >= MSG_HEADER_LENGTH) && (write_header−>size ==
      write_length))

...

/*The struct representing the message header
−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
SIZE:   Complete size of the message in bytes including the header
FLAGS:  Reserved for future use
SEQ:    Sequence number increased for each message sent out by the controller
ID:     ID of the message
−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−

Using the attribute 'packed' forces the compiler not to pad the structure
thus avoiding to add meaningless bytes in order to allign data. This choice is
made to maintain a space saving policy against a fast access one*/
struct gestic_message_header{
uint8_t size;
uint8_t flags;
uint8_t seq;
uint8_t id;
}__attribute__ ( ( packed ) );

...
```

On the other hand, as discussed in the section 3 "Hardware References", in case of a *read* operation on the device a proper management of the TS line is required.
The following figure gives a graphical representation of the of read process implemented by the driver:

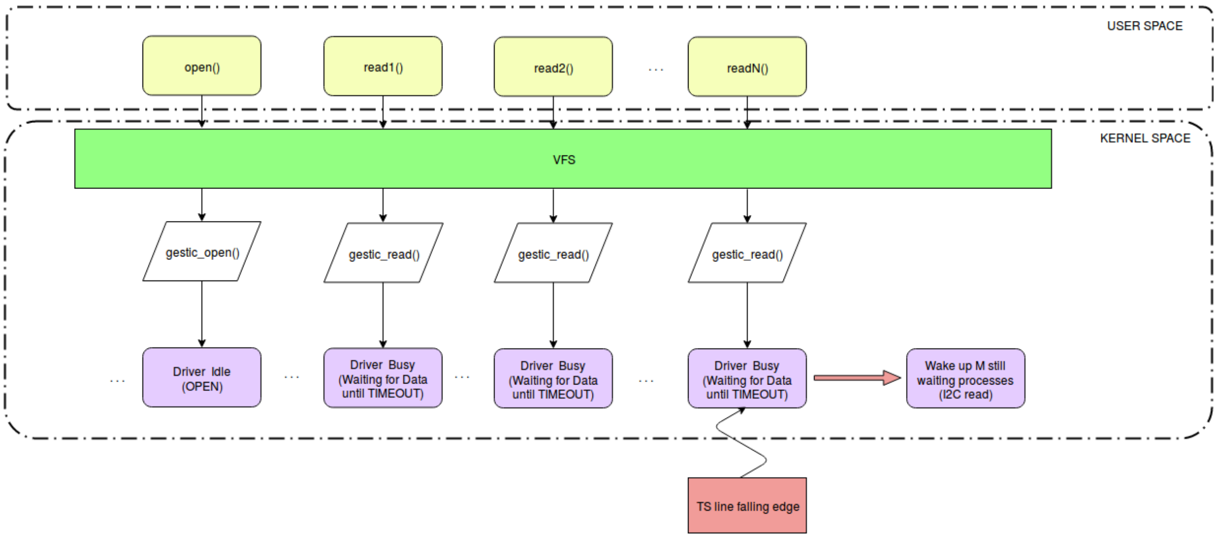

USER SPACE
open()
read1()
read2()
...
readN()
KERNEL SPACE
VFS
gestic_open()
gestic_read()
gestic_read()
gestic_read()
...
Driver Idle (OPEN)
...
Driver Busy (Waiting for Data until TIMEOUT)
...
Driver Busy (Waiting for Data until TIMEOUT)
...
Driver Busy (Waiting for Data until TIMEOUT)
Wake up M still waiting processes (I2C read)
TS line falling edge

Figure 36: gestic_read() function scheme

A call to the *read()* function from the user space level will result in a call of the *gestic_read()* driver function; this function has been designed according to TS line signal management required to read correctly from the MGC3130 controller.

```
...

/*This function includes all the common operations to be done before reading a message
The message can be actually read through I2C
Future implementation: serial*/
static ssize_t gestic_read(struct file *f, char __user *buf, size_t len, loff_t *off)
{
int error = NO_ERROR;
//The number of bytes succesfully read from the device
size_t bytes_read = 0;

//Setting the global variables
read_length = MAX_MSG_LENGTH;

//Resetting the whole read buffer before starting the transmission
memset(read_buffer, 0, read_length);

//Blocking the read process until the TS line is asserted
if (TS_CURRENT_VALUE != 0)
{
if (DEBUG) printk(KERN_INFO "GestIC: waiting for data...\n");

//The TS line is not asserted yet; the process will sleep until a signal won't wake it up.
//Right after the specified condition will be evaluated
error = wait_event_interruptible_timeout(read_queue,
(TS_CURRENT_VALUE == 0),
msecs_to_jiffies(READ_TIMEOUT));

...
```

Upon a *gestic_read()* call the process pushes itself within *waitqueue* that was previously initialized by the *gestic_init()* function; the read process will eventually be awaken by the waitqueue when one the following conditions will be verified:

- an external thread wakes him up by the waitqueue;

- a timeout elapses;

The thread in charge of waking up the read processes that are sleeping inside the queue a is the *interrupt service routine* associated with a falling edge on the TS line; if a TS falling edge happens this will mean that new data is available within the MGC3130 data buffer. thus a thread will raise and will wake up all the processes still waiting on the the waitqueue. Has been in fact verified that this approach is the most stable in terms of process management.

```
...

/*This function includes all the common operations to be done before reading a message
The message can be actually read through I2C
Future implementation: serial*/
static ssize_t gestic_read(struct file *f, char __user *buf, size_t len, loff_t *off)
{
int error = NO_ERROR;
//The number of bytes succesfully read from the device
size_t bytes_read = 0;

//Setting the global variables
read_length = MAX_MSG_LENGTH;

//Resetting the whole read buffer before starting the transmission
memset(read_buffer, 0, read_length);

//Blocking the read process until the TS line is asserted
if (TS_CURRENT_VALUE != 0)
{
if (DEBUG) printk(KERN_INFO "GestIC: waiting for data...\n");

//The TS line is not asserted yet; the process will sleep until a signal won't wake it up.
//Right after the specified condition will be evaluated
error = wait_event_interruptible_timeout(read_queue,
(TS_CURRENT_VALUE == 0),
msecs_to_jiffies(READ_TIMEOUT));

if (error == 0)
//Timeout expired
error = -ETIMELAPSED;
else
error = NO_ERROR;

}

...
```

The variable storing the timeout value is named **READ_TIMEOUT** and has been defined as constant value of 20 milliseconds; in order to use the *wait_event_interruptible_timeout* function declared inside the *linux/wait.h* library it's however necessary to convert the millisecond value in the corresponding number of *jiffies*. The jiffy conversion coefficient depends of the system and it represents the time interval that lasts between two ticks of the system timer interrupt; it can be however computed through the conversion function *msecs_to_jiffies()* that is declared within *linux/jiffies.h*. When the process awakes it sets the error local variable according to the event that caused him to wake up from the sleep; if no error has occurred the read process ends up with the *gestic_I2C_read()* function that implements the actual I2C data caption.

```
...
```

```
ASSERT_TS;
if (DEBUG) printk(KERN_INFO "GestIC: re-asserting TS\n");
//Reading the message through I2C
bytes_read = gestic_I2C_read();
if (bytes_read >= 0)
memcpy(buf, read_buffer, bytes_read);

//Releasing the TS line
RELEASE_TS;
msleep(i2c_delay_read);

...
```

Following the MGC3130 protocol for message receive the master asserts the TS line low to ensure the data buffer won't update while reading and then it releases TS line GPIO high after the I2C read function has been executed; the interrupt service routine that starts after a falling edge caption of the TS line is initialized within the *init* function and looks as follows:

```
...

/*The interrupt which handles the TS line value changes*/
static irqreturn_t ts_changed(int irq, void *dev_id)
{
static int ts_last_value = 1;
int ts_new_value = !!TS_CURRENT_VALUE;

if (DEBUG) printk(KERN_INFO "GestIC: value TS is : %d\n", ts_new_value);

//Checking if the irq of the interrupt is the one associated with the TS line
if (irq != ts_irq_num)
{
if (DEBUG) printk(KERN_INFO "GestIC: no need to handle this interrupt\n");
return IRQ_NONE;
}

//If the TS line was asserted by the controller some new data is available; thus the read process has
//to be woke-up
if (ts_new_value != ts_last_value)
{
if (!!ts_new_value == 0)
{
wake_up_all(&read_queue);
}
}

ts_last_value = ts_new_value;
if (DEBUG) printk(KERN_INFO "GestIC: interrupt handled\n");

return IRQ_HANDLED;
}

...
```

As explained the read processes waiting on the read queue are awaken by the *wake_up_all()* function call whenever the TS value stored within the *ts_new_value* variable is equal to 0.

4.4 Deploying and Debugging the Module

Once the driver has been completed the Yocto environment has everything that is needed and is thus ready to produce the new image for the UDOO Neo board; the produced image will store the device module just created inside the OS file system under the path */lib/modules/4.1.15-2.0.x-udoo+g7773e46/extra/* of the board with the name **gestic.ko**. To compile the image it is necessary to resume the build and then run the *bitbake* command as:

```
bitbake udoo_image_full_cmdline
```

The process will take care of compiling the driver source code as the rest of the image. Moreover since a recipe was created for the developed driver it is possible to recompile just the its source code upon any patch application on the source code with the commands:

```
/thesis/udoo-community-bsp/build_thesis$ bitbake -c clean gestic-mod
```

The command will execute the *clean* target of the Makefile created during the last section thus removing each output of the driver compilation of any previous build. In order to recompile the module won0t be necessary the re-compilation of all the image source code but only the one related to the driver recipe. To re-compile the driver the following command shall be executed after the clean command:

```
/thesis/udoo-community-bsp/build_thesis$ bitbake gestic-mod
```

The command will recompile only the device module with the updated version of the source code.
Since a recompilation of the driver has been made the version previously deployed on the board inside the file system is no more up to date; it will be thus necessary to upload the new version of the module from the PC directly into the board. The last version of the gestic.ko mod will be always found within the *tmp/sysroots-components/udooneo/gestic-mod/lib/modules/4.1.15-2.0.x-udoo+g7773e46/extra/* folder of the current build.
There are several ways to send the file to the board and the process may as simple as loading it from a portable USB device used to transfer file between the PC and the host. However during the driver development and test phase the choice has been for a **SFTP** connection to deploy file and an **SSH** connection to open a terminal on the board.
To use the board in such a method it is necessary to connect it to the Host PC network; the network interface on the board is initially configured to acquire an IP address when assigned externally from a **DHCP** device. This can be spotted out by running on a minicom terminal connected with the board the following command:

```
root@udooneo:# cat /etc/network/interfaces
# /etc/network/interfaces -- configuration file for ifup(8),

...

# Wired or wireless interfaces
auto eth0
iface eth0 inet dhcp

...
```

While the *interfaces* file is configured in such a way the OS will try to acquire an IP address assigned by the DHCP; this would be the right choice when working with a board remotely connected to the Host PC network through a DHCP device as for instance a router.
However during this project the board has almost always been kept directly connected with the host PC; thus has been necessary the reconfiguration of the *interfaces* file on the UDOO Neo board and the creation of a new wired connection on the Host PC.
To setup a static IP address for the UDOO Neo board the *interfaces* file under */etc/network/* directory has been modified as follows:

```
root@udooneo:# cat /etc/network/interfaces
# /etc/network/interfaces -- configuration file for ifup(8),

...

# Wired or wireless interfaces
auto eth0
iface eth0 inet static
address 192.168.1.2
netmask 255.255.255.0
gateway 192.168.1.255

...
```

With this configuration basically the UDOO Neo board will always have the same IP address **192.168.1.2**; moreover it has been necessary to configure a new cabled LAN connection for the Host PC. The new PC connection has been created through the "Network" application offered by the Linux Ubuntu OSM; the properties to set for a proper connection are:

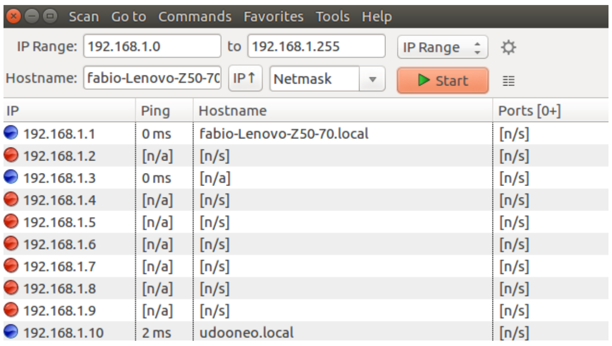
- IPv4 IP protocol;
- the Netmask is 24;
- the Gateway address is 192.168.1.1;
- the IP address for the the PC is 192.168.1.1;

While on the host PC the changes to the IP address are fulfilled at run, to make true the changes on the board is however necessary to synchronize the file system and optionally perform a *rebooot* o the OS; this can be done through the following commands:

```
root@udooneo:# sync
root@udooneo:# reboot
```

After the reboot the board will be connected to th PC network; the result can be monitored through any IP management tool. To attend this job the "Angry IP Scanner" application has been used several times; moreover the board can now be found under the name of *udooneo.local* on the PC network.

even if it is just an example any IP scanner tool should produce a result looking more or less as follows:



IP Range: 192.168.1.0 to 192.168.1.255
Hostname: fabio-Lenovo-Z50-7C IP↑ Netmask

IP	Ping	Hostname	Ports [0+]
🔵 192.168.1.1	0 ms	fabio-Lenovo-Z50-70.local	[n/s]
🔴 192.168.1.2	[n/a]	[n/s]	[n/s]
🔵 192.168.1.3	0 ms	[n/a]	[n/s]
🔴 192.168.1.4	[n/a]	[n/s]	[n/s]
🔴 192.168.1.5	[n/a]	[n/s]	[n/s]
🔴 192.168.1.6	[n/a]	[n/s]	[n/s]
🔴 192.168.1.7	[n/a]	[n/s]	[n/s]
🔴 192.168.1.8	[n/a]	[n/s]	[n/s]
🔴 192.168.1.9	[n/a]	[n/s]	[n/s]
🔵 192.168.1.10	2 ms	udooneo.local	[n/s]

Figure 37: Angry IP Scanner results

Since now the system is available within the system it is possible to transfer a file on the Linux file system through the Ethernet connection. In order to transfer any new version of the device module it is possible to run (for a visual convenience from the folder containing the gestic.ko module) the following commands :

```
$ cd thesis/udoo-community-bsp/build_thesis/tmp/sysroots-components/udooneo/ge
stic-mod/lib/modules/4.1.15-2.0.x-udoo+g7773e46/extra/
$ sftp root@udooneo.local
Connected to udooneo.local.
sftp> put gestic.ko
Uploading gestic.ko to /home/root/gestic.ko
gestic.ko 100% 17KB 17.1KB/s 00:00
sftp> exit
```

Now the device module has been sent to the UDOO Neo through the LAN connection and can be found, as the output states, within */home/root/* directory.
as now the board can be found inside the PC network is possible to connect with the remote system by opening an SSH connection with it; this can be done by running the following command:

```
$ ssh root@udooneo.local
Last login: Sun Dec 17 16:54:29 2017 from 192.168.1.1
root@udooneo:#
```

From the terminal opened it is now possible to install and debug the module device deployed before; to install the module in debug mode it is sufficient to run:

```
root@udooneo:# insmod gestic.ko gestic_debug
```

The install command in fact is implicitly setting the *gestic_debug* boolean value, referred inside the driver with the macro *DEBUG*, as true. To check the log printed on kernel output by the device driver will be enough to run the *dmesg* command:

```
root@udooneo:# dmesg

...

[ 3206.847244]

GestIC: the module is in DEBUG mode
[ 3206.856741] GestIC: character device numbers registration succeded
[ 3206.864598] GestIC: class initialization succeded
[ 3206.870765] GestIC: associated device initialization succeded
[ 3206.877056] GestIC: Initializing the character device data structure
[ 3206.883529] GestIC: character device data structure initialization succeded
[ 3206.890494] GestIC: I2C adapter structure initialization succeded
[ 3206.899225] GestIC: I2C client initialization succeded
[ 3206.904457] GestIC: TS line initialization succeded
[ 3206.909342] GestIC: TS line direction configuration succeded
[ 3206.915614] GestIC: MCLR line initialization succeded
[ 3206.920681] GestIC: MCLR line direction configuration succeded
[ 3206.926982] GestIC: interrupt number detection succeded
[ 3206.932266] GestIC: TS associated interrupt initialization succeded
[ 3206.939001] GestIC: initialization completed
[ 3206.943345] GestIC device: driver installed
```

In order to monitor in a more ”real-time” mode the kernel log it is instead possible to run:

```
root@udooneo:# watch -n 0.1 "dmesg | tail -50"
```

This command will basically refresh the *dmesg* command output **each 100ms** giving the idea of time elapsing while debugging the kernel log.
When the driver is installed a new character device will appear in the file system; in fact a new device will be registered under the */dev* folder,

```
root@udooneo:# ls -al /dev/gestic
crw------- 1 root root 247, 0 Dec 17 17:43 /dev/gestic
```

The module device driver can be installed and removed at any time; the OS won’t however allow to remove the module if any process using the */dev/gestic* file is active. To remove the module it will be sufficient to run:

```
root@udooneo:# rmmod gestic
```

The module will be removed by the Linux Embedded OS; to check the correct disposition of the module it is possible to check again the kernel log,

```
root@udooneo:# dmesg

...

[ 6772.776550] GestIC device: driver removed
```

or check for the character device within the */dev* folder as done before.

```
root@udooneo:~# ls -la /dev/gestic
ls: cannot access '/dev/gestic': No such file or directory
```

5 GestIC Control Application

Within this section will be discussed the design and the development phases that brought to the realization of the *gestic_control* application exploiting the driver created during the section 4 "Driver Development" and running on the UDOO Neo board.
The main steps that occurred into the realization of the application were:

- the setup of the Eclipse IDE to cross-compile an application running on Linux Embedded;
- the customization of the Microchip GestIC API for making it compatible with the driver implementation;
- the design of a multi-thread slave application able to receive data from the controller, forward the data to the host PC and interpreting commands on serial.

Before moving on the actual developing phase a brief comparison between compilation and cross-compilation will be made.

5.1 Compilers and Cross-Compiler

The *compiler* is a the software whose purpose is to *translate* a source-code written with some high-level programming language into *object code* or, in general a lower-level code. The **object code** is thus the product of the compile process; it basically represents a sequence of instructions written in a computer-understandable language (machine code). In order to produce a *executable* file the object codes that may be the result of multiple compilations will be combined by the **linker** software to produce an executable file i.e. a file that can be executed by the machine.

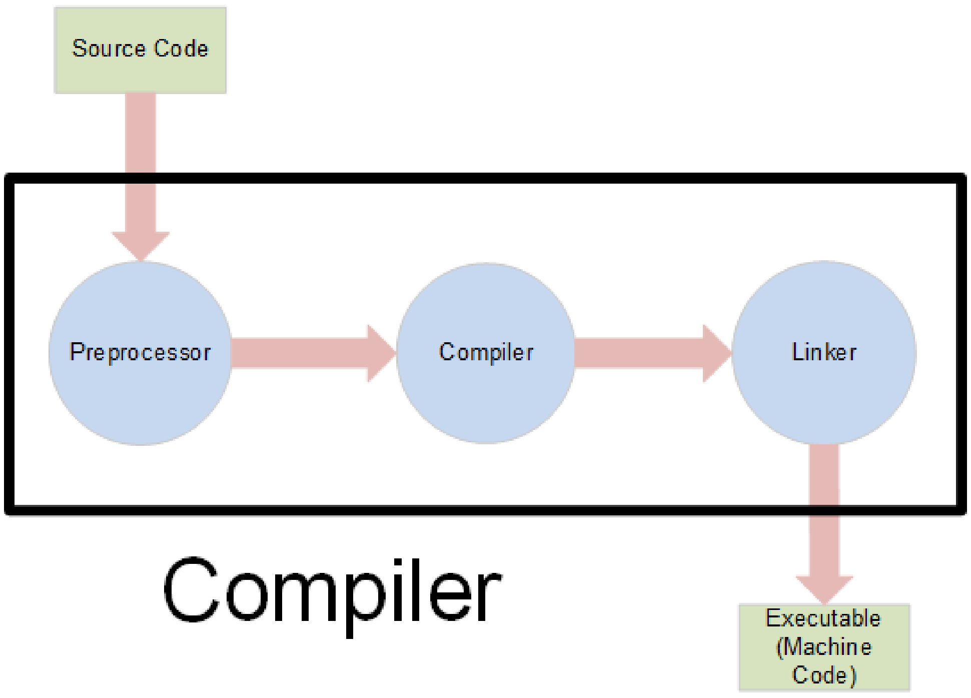


Figure 38: Executable file production

As can be seen by the figure the compilation process may also invoke a *preprocessor*, even if this phase may occur in different moments according to te specific compiler design. The preprocessor is basically designed to manipulate the source code during the compilation process and to perform actions as for instance macros expansions.

On the other hand the *cross-compiler* is a software intended to compile source that will run on a *target platform* that will be different by the host platform; this is typically the case of embedded programming where the target system has not an OS or enough computing resources to handle the compilation. The free **GCC** compilers collection (GNU Compilers Collection) can be used to cross-compile source-code for a target system; this will however require that the C libraries of the target system are available on the *host system*.
In order to produce an application from an host pc for a target machine other useful tool are the cross-linker, that will act as a standard linker but will produce an executable for a target system, and a cross-debugger used to analyze and control the application execution from the build pc.
The collection of the tools that are needed to handle a complex software development as application cross-development, can be referred as **toolchain**; in order to setup the development environment is thus needed to get a toolchain feasible with the target system properties and the host machine as well. In order to achieve this goal the Yocto Project tools have been used once more.

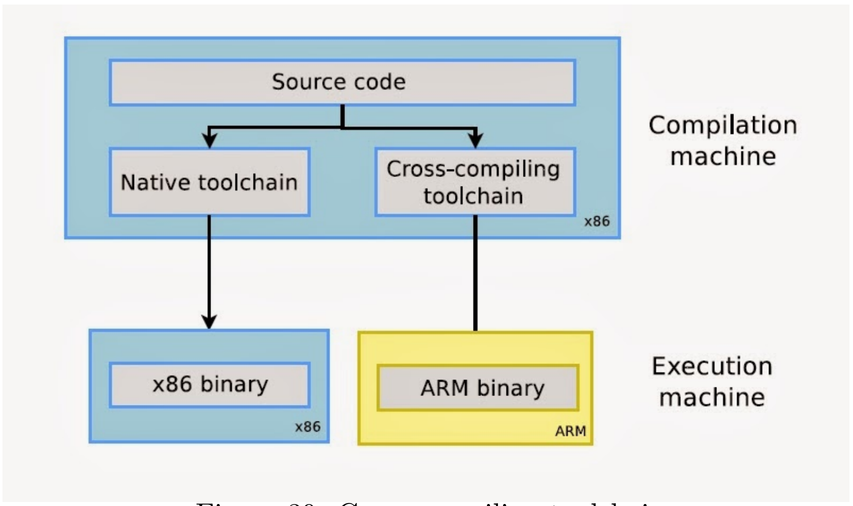


Figure 39: Cross-compiling toolchain

The Yocto Project allows in fact the user to build a custom toolchain installer that depends on the build environment and the used image recipe; so after the build environment has been resumed it is possible to produce an installer for the correct toolchain:

```
/thesis/udoo-community-bsp$ source ./setup-environment build_thesis/
/thesis/udoo-community-bsp/build_thesis$ bitbake -c populate_sdk udoo-image-ful
l-cmdline
```

The *populate_sdk* will produce the toolchain installer that will be useful when setting up the IDE for cross-development; the executable file will be put inside the folder *tmp/deploy/sdk* of the build folder itself.
Moreover the toolchain produced by the Yocto Project tools will be a self-contained folder, in fact everything that will be required during the cross-development phase will be found inside that folder.
The executable file produced by Yocto can be run to install the toolchain inside any folder; in this case the folder has been left the default on */opt/poky/2.3.2/*:

```
/thesis/udoo-community-bsp/build_thesis$ cd tmp/deploy/sdk/
/thesis/udoo-community-bsp/build_thesis/tmp/deploy/sdk/ $ ./poky-glibc-x86_64-u
doo-image-full-cmdline-cortexa9hf-neon-toolchain-2.3.2
```

```
Poky (Yocto Project Reference Distro) SDK installer version 2.3.2
=================================================================
Enter target directory for SDK (default: /opt/poky/2.3.2):
You are about to install the SDK to "/opt/poky/2.3.2". Proceed[Y/n]? y
Extracting SDK.............................................................done
Setting it up...done
SDK has been successfully set up and is ready to be used.
Each time you wish to use the SDK in a new shell session, you need to source th
e environment setup script e.g.
$ . /home/fabio/toolchain/environment-setup-cortexa9hf-neon-poky-linux-gnueabi
```

As can be spotted by the output of the executable the toolchain for cross-development has been installed within the requested folder; since we are integrating that folder inside the IDE for application development it won't be necessary to source the setup script manually.
As can be seen by listing the contents of the */opt/poky/2.3.2/sysroots/* folder that will be installed on the file system by the script

```
$ ls /opt/poky/2.3.2/sysroots/
cortexa9hf-neon-poky-linux-gnueabi x86_64-pokysdk-linux
```

two folder containing system roots for both the system architecture have been included.

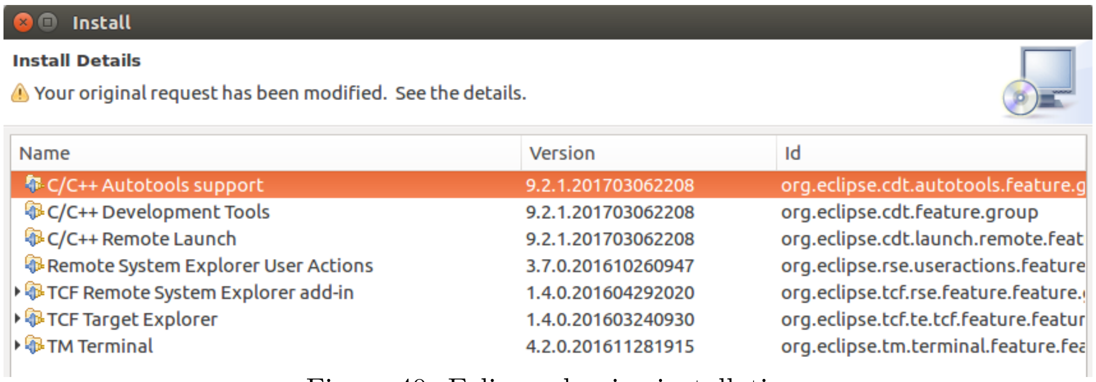
5.2 Configuring Eclipse and the Yocto SDK

Once the toolchain has been installed is possible to configure the IDE to perform cross-development; The choice for the IDE has been the Eclipse for three essential reasons:

- it is free and thus widely used, thus many tutorials and instructions can be found on the web to make a fast setup of the environment;
- it allows an highly customization and supports a waste number of programming languages; in fact, even if it's origins can be historically found in Java development, it now support most of the commonly used languages as C/C++ or Python. Moreover to add some capability to the IDE it's sufficient to install a specific plug-in to extend the Eclipse IDE functionalities;
- an Eclipse plug in named **Yocto SDK** (Yocto Software Development Kit) for cross-development is directly maintained by the Yocto Project, allowing an easy way to design applications for a target embedded system.

The version of the Eclipse IDE used during the thesis project is Eclipse Neon; even if an older version of the IDE may work well the following instructions have been tested on the Neon version only.
The IDE can be downloaded from the official Eclipse Project web site at `http://www.eclipse.org/downloads/download.php?file=/technology/epp/downloads/release/neon/3/eclipse-cpp-neon-3-linux-gtk-x86_64.tar.gz`; once the archive has been downloaded its contents can be extracted inside any folder. After this step no other actions are needed to install the IDE since it can be run directly from the extraction folder.

To configure the Eclipse IDE some extensions have to be downloaded; before downloading those packages is however suggested to launch a check for update: this operation will in fact update some of the packages used during the development. To launch update it is possible to run *Help - Check for Updates*. To add the remaining plug-ins to the IDE it is sufficient to search for *Help - Install New Software*; a summary of the needed extensions can be seen in the following figure:



Install

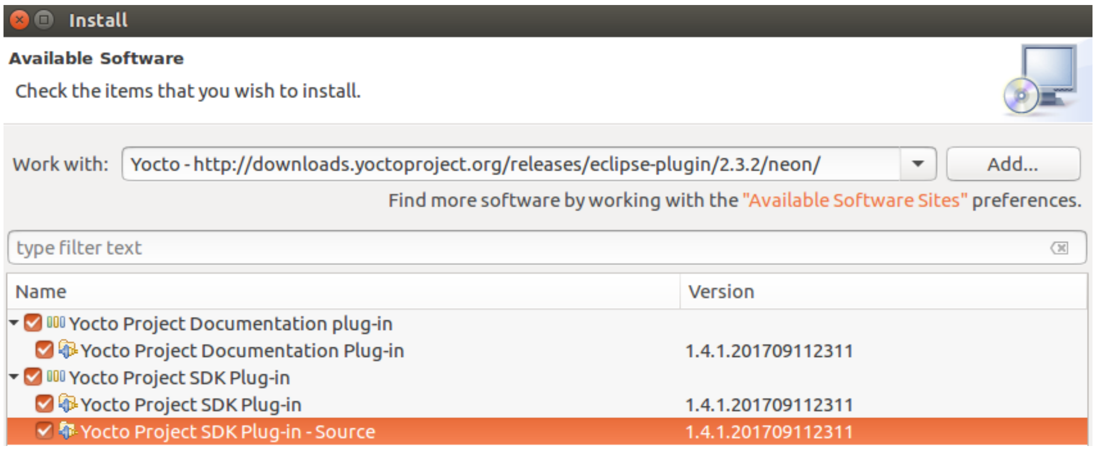
Install Details

⚠ Your original request has been modified. See the details.

Name	Version	Id
C/C++ Autotools support	9.2.1.201703062208	org.eclipse.cdt.autotools.feature.g
C/C++ Development Tools	9.2.1.201703062208	org.eclipse.cdt.feature.group
C/C++ Remote Launch	9.2.1.201703062208	org.eclipse.cdt.launch.remote.feat
Remote System Explorer User Actions	3.7.0.201610260947	org.eclipse.rse.useractions.feature
TCF Remote System Explorer add-in	1.4.0.201604292020	org.eclipse.tcf.rse.feature.feature.
TCF Target Explorer	1.4.0.201603240930	org.eclipse.tcf.te.tcf.feature.featur
TM Terminal	4.2.0.201611281915	org.eclipse.tm.terminal.feature.fea

Figure 40: Eclipse plug-ins installation

The installed extensions basically provide Eclipse some useful tool to remotely manipulate the target system root file system and some C/C++ writing tools since the language chosen for the application is C.
At Eclipse restart the last element, the Yocto ADT plug-in has to be installed in the IDE. To do this is necessary to reach again the *Help - Install New Software* option; a new source website not listed between the several choices has to be added, by pressing the *Add* button, with name *Yocto* and location `http://downloads.yoctoproject.org/releases/eclipse-plugin/2.3.2/neon/`. From this source shall be fetched two packages, one for documentation and one for the Yocto SDK.



Install

Available Software

Check the items that you wish to install.

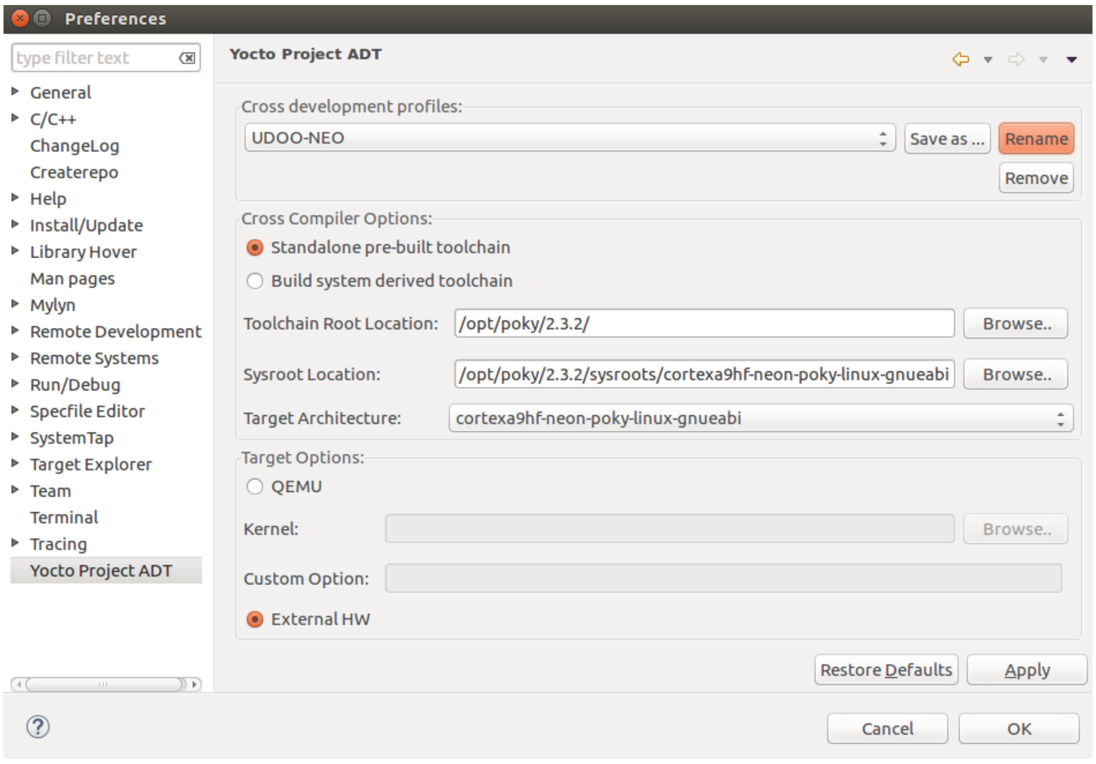
Work with: Yocto - http://downloads.yoctoproject.org/releases/eclipse-plugin/2.3.2/neon/

Find more software by working with the "Available Software Sites" preferences.

Name	Version
✓ Yocto Project Documentation plug-in	
✓ Yocto Project Documentation Plug-in	1.4.1.201709112311
✓ Yocto Project SDK Plug-in	
✓ Yocto Project SDK Plug-in	1.4.1.201709112311
✓ Yocto Project SDK Plug-in - Source	1.4.1.201709112311

Figure 41: Eclipse Yocto SDK installation

After the IDE has restarted the plug-in can be configured; in order to do this correctly the location of the toolchain root and the location of the sysroots folder has to be set under the *Window - Preferences - Yocto ADT* option as can be seen in *Figure 38*. The current settings has been saved under the *UDOO-NEO* name and set as default.


Preferences

Yocto Project ADT

Cross development profiles: UDOO-NEO

Cross Compiler Options:
- [✓ Standalone pre-built toolchain]
- Build system derived toolchain

Toolchain Root Location: /opt/poky/2.3.2/

Sysroot Location: /opt/poky/2.3.2/sysroots/cortexa9hf-neon-poky-linux-gnueabi

Target Architecture: cortexa9hf-neon-poky-linux-gnueabi

Target Options:
- QEMU
- Kernel:
- Custom Option:
- [✓ External HW]

Figure 42: Eclipse Yocto SDK configuration

5.3 GestIC API Customization

Since the setup is now finished it is eventually possible to create a new application; to do this it is sufficient to search for *File - New - C Project* to start the project creation wizard. The application designed to show the driver behavior has been created as a "Hello World ANSI C Autotools Project", a simple "Hello World!" application, and modified during the different steps of the design. While the current subsection will underline the most important part of the API and the major customizations to the existing code, the following subsection will see more in detail the structure of the application.
The first step to integrate the GestIC API written by Microchip is to fetch the source code from the Hillstar KIT web page under the download panel; the archive which contains the GestIC API is the one named *Aurea Software Package - Aurea GUI and GestIC Library*. After the SDK inside the archive has been installed the folder containing the GestIC API will be available under the *api* folder of the extraction path while the documentation on the API features can be found under the *doc* folder.
The API contents are shown in *Figure 39*:

Figure 43: GestIC API folder contents

The API is divided into four folders:

1. the *include* folder contains the library that shall be ever included inside an application in order to exploit the API capabilities; there are three main headers to be included, the *gestic_api.h* is the most important between them since it represents the major customization point and the interface with the rest of the libraries;

2. the lib folder contains the all API source code in the form of a *.dll* file (dynamic library version of the API);

3. the *lib-static* folder contains the the API source code in the form of a *.lib* static library;

4. the *src* contains the source code written in C in the form of source files and headers;

During the project the two folder whose contents have been included inside the application are the *include* and the *src* folders. However before including the API files inside the Eclipse newly created project it is advisable to launch at least once the build process in order to

1. make sure that everything is working fine for the most basic project possible, the one with just the hello world source file;

2. allow the autotools to create the project configuration files for the basic version of the project;

The application in this state may be seen as in his "0" state, an initial working condition.

In order to exploit the API it is necessary to include the source files and the header files needed inside the Eclipse project before starting any modification to them; first build will produce several configuration files generated by autotools. One of them is the *makefile.am* file that (even if this can be handled by the autotools) has been manually updated each time a new source file or header file has been included within the project.
To include a copy of the API sources inside the build it is necessary to double-click on the *src* folder of the project (in the project explorer view) and search for import option; when the wizard opens it's necessary to browse for the two folders previously discussed and include from them the necessary libraries and source files. Moreover it would be necessary in order to specify the needed files within the project to add them in the makefile.am file.
The makefile.am file, that can be found within the *src* folder of the Eclipse Project, looks like:

```
bin_PROGRAMS = gestic_control
gestic_control_SOURCES = gui.h gui.c serial.h serial.c gestic_control.c gestic_
api.h arch.h core.c gestic_custom.h gestic_custom.c gestic_static.h impl.h rtc.
c stream.c x86_linux.h fw_version.c flash.c

AM_CFLAGS = @gestic_control_CFLAGS@
AM_LDFLAGS = @gestic_control_LIBS@ -lpthread

CLEANFILES = *~
```

As can be noticed all the sources are listed inside the *gestic_control_SOURCES* as a concatenation of names; keeping this string updated will prevent the IDE to raise an "Undefined Reference" error when launching the build process. As can be noticed most of the included files belong to reference API, make an exception the files from *gui.h* to *serial.c* that don't come form the API folder and the *gestic_control.c* that is the main file of the project. Eventually a reference to the *pthread.h* library has been made in order to include the thread functionalities into the

application.
As already said the gestic_api.h file is the entry point for the GestIC API; it is organized as a sequence of conditional inclusions and declaration based on defined values. In other words whenever a value with a certain name is defined within one of the files of the project, the api will include some libraries and declare some function instead of others; the code snippet reported below represents an example of this particular design of the API:

```
...

#define GESTIC_CUSTOM
#ifdef GESTIC_CUSTOM
#include "gestic_custom.h"
#endif

...
```

This snipperìt shows how in this particular case has been decided to define the *GESTIC_CUSTOM* value; this allows, even if the value is defined without value, to interact with the API that, according to this portion of code, will include the *gestic_custom.h* file and at the same time not include other possible implementations in order to avoid conflicts. In order to avoid mutual exclusion violations it is recommended to modify only the API file.
Since this API has been intended to work on a system that receives data through a serial connection by the I2C to USB cable, the inclusion of a *gestic custom library* has been the first modification that has been done to the gestic_api.h file. Through this file, it is possible to set different properties with respect to the default one:

```
...

//#define GESTIC_NO_FW_VERSION
//#define GESTIC_NO_FLASH
//#define GESTIC_NO_DATA_RETRIEVAL
//#define GESTIC_NO_RTC
#define GESTIC_NO_LOGGING


/* Notify GestIC-API that we provide a custom IO-implementation */

#ifdef GESTIC_IO
#undef GESTIC_IO
#endif
#define GESTIC_IO GESTIC_IO_CUSTOM

...
```

The file it's used, for instance, to exclude the logging functionalities of the API (no log is maintained using the API functions), and to set the I/O type of the application as custom; by default it would have been a serial I/O type since this software has been intended to make a host PC communicate with the host by serial. Defining GESTIC_IO as GESTIC_IO_CUSTOM the API interface won't integrate the serial implementation of the Input/Output routines that int this case will interface the file operation developed into the driver.
The consequence on the *gestic_api.h* file:

```
...

#if GESTIC_IO == GESTIC_IO_CUSTOM
/* Nothing to define here */
#elif GESTIC_IO == GESTIC_IO_CDC_SERIAL
#   define GESTIC_HAS_SERIAL_IO
#   define GESTIC_USE_IO_CDC_SERIAL
#   define GESTIC_USE_MSG_EXTRACT
#else
#   error "Unknown IO implementation selected"
#endif

...
```

As said the functionalities implemented for the serial I/O version of the interface have not been included.
Since a customized version has been declared inside the *gestic_api.h* file it has been necessary to provide the custom implementation source file **gestic_custom.c**; within this file have been implemented all the basics interaction between the user level application and the character device that, as explained in section 3 "Driver Development", is going to model the GestIC controller when the device module will be installed on the UDOO Neo root file system. The most useful functions implemented in this source file are those functions used to open or close the device and the two functions used to exchange messages with the module. Here follows a brief summary on the most important part of these functions; the function used to open the device is the *gestic_open()* reported in the following snippet:

```
...

//Opening the device
if((fd = open(GESTIC_DEV_NAME, O_RDWR))<0)
{
puts("Error while opening the device.\n");
error = GESTIC_IO_OPEN_ERROR;
}

if (error == GESTIC_NO_ERROR)
{
gestic->io.fd = fd;
gestic->io.connected = 1;

/* Sleeping 300 ms to be sure the device is ready to be polled
* after reset.
*/
usleep(POST_OPEN_USLEEP);


//Emptying the buffer from any initial and possibly corrupted message
for(i=0; i<5; i++)
{
read(gestic->io.fd, buf, sizeof(buf));
}
}

...
```

As can be pointed out it is quite simple to open the driver module from the user space; in fact a simple call to the *open()* function has to be performed as for a simple text file. Moreover a time transient of 300 ms will be awaited on startup in order to avoid undesired messages from the device, like for instance the firmware version message sent on startup by the controller; since after the startup some unexpected messages (whose structure was not reported on the documentation version provided with the kit) are sent by the controller a sort of initial flush has been implemented by trying to rawly read up to 5 messages from the MGC3130 output buffer.
The close function instead won't do nothing more then closing a file with a call to the *close()* function:

```
...
```

```
close(gestic−>io.fd);
gestic−>io.connected = 0;

...
```

It is important to underline that both the file descriptor *fd* and the *connected* boolean value are part of a structure that stores the main information on the GestIC device such as firmware informations and the results of the received messages interpretation.
The function that can be used to receive messages from the device is the *gestic_message_receive* and has been implemented as follows:

```
...

for(i=0; i<3; i++) {
msg_size = read(gestic−>io.fd, buf, sizeof(buf));
if(buf) {
gestic_message_handle(gestic, buf, msg_size);
break;
}

if(!timeout || (∗timeout <= 0)) {
result = GESTIC_NO_DATA;
break;
}

...
```

This function will try to read a message up to three times from the device before performing a check on the timeout expiration; in this case the response will be a GESTIC_NO_DATA error to the calling function, otherwise the message will be handled through the *gestic_message_handle* function to determine the nature of the received data frame. As can be noticed also in this case the bufer has been filled by calling the *read()* function as can be done for a simple file.
The last basic function is the *gestic_message_write()*:

```
...

bytes = write(gestic−>io.fd, (char ∗)msg, size);

if (bytes != size)
error = GESTIC_IO_ERROR;

...
```

In this case no retries are performed in case of unsuccessful write at this level since this control is done by the functions that calls this one; this allowed to keep the message_write function simple as possible.
The last function that should be described at this level is the *gestic_reset()* function:

```
...

  // Setting up the message as a RESET_COMMAND
  SET_U8(msg, 0);
  SET_U16(msg + 1, 0);
  SET_U8(msg + 3, 17);
  SET_U32(msg + 4, 0);

  gestic_message_write(gestic, msg, sizeof(msg));

  return GESTIC_NO_ERROR;

...
```

This is achieved by simply sending a predefined message exploiting the previously discussed function; the driver has been in fact implemented, as explained in subsection 4.3.2 "File Operations", to identify this particular command and send a reset signal to the controller by manipulating the physical pin. The same idea has been applied for the *gestic_sleep()* function that allows the user space to set a sleep time for the device module.

5.4 Application Design

Once the basic bricks has been implemented the application design phase can start; the application includes all the sources previously discussed.
The *gestic_control()* application has been designed to run on UDOO Neo board after the gestic device module has been installed; it fulfills several jobs at the same time since it has been designed as a multi-thread application. These jobs are basically three:

1. sending data to the host PC through serial connection;
2. receiving commands from the host PC that within the applications acts as master;
3. receiving new data from the device module and keeping updated the infrastructure used to store informations from the MGC3130;

In order to develop this application it is needed to include in the project the *lpthread* library as explained in the previous subsection when showing the makefile.am file. The following figure gives a representation of the application basic structure:

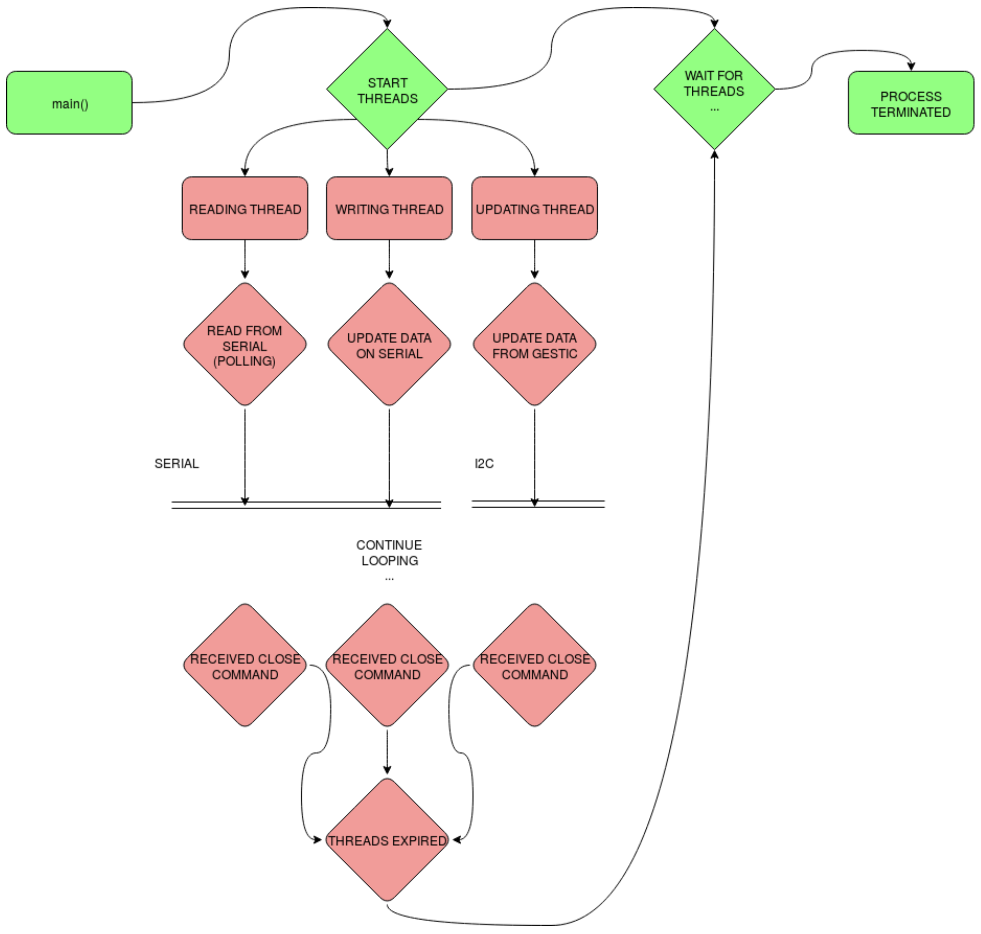


Figure 44: Application basic structure

As can be seen in Figure 40 the main process launches the three threads and wait until they finish; on the other hand the threads are structured as loop functions that will end only when the receiving thread is getting a close command. When this is happening the main process will wake up and terminate. This has been done to make sure that all the threads expire correctly by checking the status of the main process.

```
...

 pthread_create(&pth_gesticUpdater, NULL, (void*)gestic_stream_update, &gestic);
 pthread_create(&pth_serialWriter, NULL, (void*)serial_writer_thread, &arguments);
 pthread_create(&pth_serialReader, NULL, (void*)serial_reader_thread, &serial);

 pthread_join(pth_serialReader, NULL);
 pthread_join(pth_serialWriter, NULL);
 pthread_join(pth_gesticUpdater, NULL);

 free_gestic_device(&gestic);

 exit(EXIT_SUCCESS);
 }

...
```

It is possible to recognized the described structure; the call are made by the main process after:

1. having initialized the GestIC data structure;
2. having declared three *mutex* used to synchronize the threads;
3. having opened the serial connection to the the ttyGS0 port;

It is moreover important to notice that will be, in some cases, problems related to mutual exclusion on variables. In fact the reading and the writing threads have to share the serial line while the writing and the updating threads have to share the data structure used to store informations sent by MGC3130. This is the basic reason which explains the use of mutexes to synchronize the three threads.
The first of the three threads being analyzed is the *serial_reader_thread*:

```
...

while(!exitFlag)
{
if (serial_poll(serial, SERIAL_POLL_TIMEOUT) > 0)
{
pthread_mutex_lock(&serialLock);
memset(cmd, 0, sizeof(cmd));
/* Read up to buffer size or timeout */
if (serial_read(serial, cmd, sizeof(cmd), 0) < 0)
{
fprintf(stderr, "serial_reader_thread: %s\n",
serial_errmsg(serial));
pthread_mutex_unlock(&serialLock);
return;
}
pthread_mutex_unlock(&serialLock);

cmd_id = command_handle(cmd);
valid_cmd = command_is_valid(cmd_id);

if (valid_cmd)
{
```

```
if (cmd_id == EXIT)
exitFlag = true;
else
{
pthread_mutex_lock(&gesticLock);
...
pthread_mutex_unlock(&gesticLock);
}
}

cmd_id = NO_COMMAND;
valid_cmd = false;
}

...
```

As can be seen the serial reader thread has the following structure:

1. it polls the serial line for received messages for 5ms;
2. if the thread finds that new data is available on the serial line it locks the connection through the *serialLock* mutex to preempt the resource from other processes;
3. if the data received represents a valid command it locks the *gestic* resource and execute the corresponding task;
4. when the *EXIT* command is received the loop terminates and the thread expires.

The *serial_writer_thread* shall be synchronized with the other two in order to let the reading thread preempt the serial line and the updating thread store new values within the GestIC structure.

```
...

while (!exitFlag)
{
pthread_mutex_lock(&serialLock);
pthread_mutex_lock(&gesticLock);
if (dataUpdated)
{
//Writing the data to the host
set_data_frame(gestic, &data_frame);
serial_flush(serial);
if((bytes = serial_write(serial, (uint8_t*)&data_frame, sizeof(data_frame))) < 0)
{
fprintf(stderr, "serial_write(): %s\n", serial_errmsg(serial));
pthread_mutex_unlock(&gesticLock);
pthread_mutex_unlock(&serialLock);
exit(GESTIC_IO_ERROR);
}

dataUpdated = false;
}
pthread_mutex_unlock(&gesticLock);
pthread_mutex_unlock(&serialLock);
}

...
```

This second thread has been designed as follows:

1. when both the serial line and the data structure resources are available it preempts them and check weather new data has to be sent through the *dataUpdated* flag;
2. it prepares the *data_frame* to be sent on the serial connection;
3. upon a successful write it frees the resources preempted and wait for new data to be available;
4. when the *EXIT* command is received by the *serial_reader_thread* the loop terminates and the thread expires.

The data_frame for the current version of the application packages the most important data only even if it could be easily extended to carry more informations about the status of the controller. The *data_frame* brings the following informations:

- the object position in (x, y, z) coordinates (the center of mass in case of extended surfaces);
- a recognized gesture if a valid movement has been caught by the controller or a *GESTURE_NONE* otherwise;
- a mask of bits bringing the information of the electrodes that have been touched (multi-touch is allowed);
- a counter incremented if a clockwise *air_wheel* gesture is in progress and decremented in case of a counter-clockwise progression;

To make data interpretation easy for any client application willing to retrieve those informations from the UDOO Neo board it has been decided to implement the frame interpreting functions in different libraries that can act as stand-alone and thus don't need the full API for being used; this allows an easy porting of the operations that have to be done on the data_frame and to hide any information about the API implementation to the client application.
The last thread that has been implemented is the *gestic_stream_update* thread:

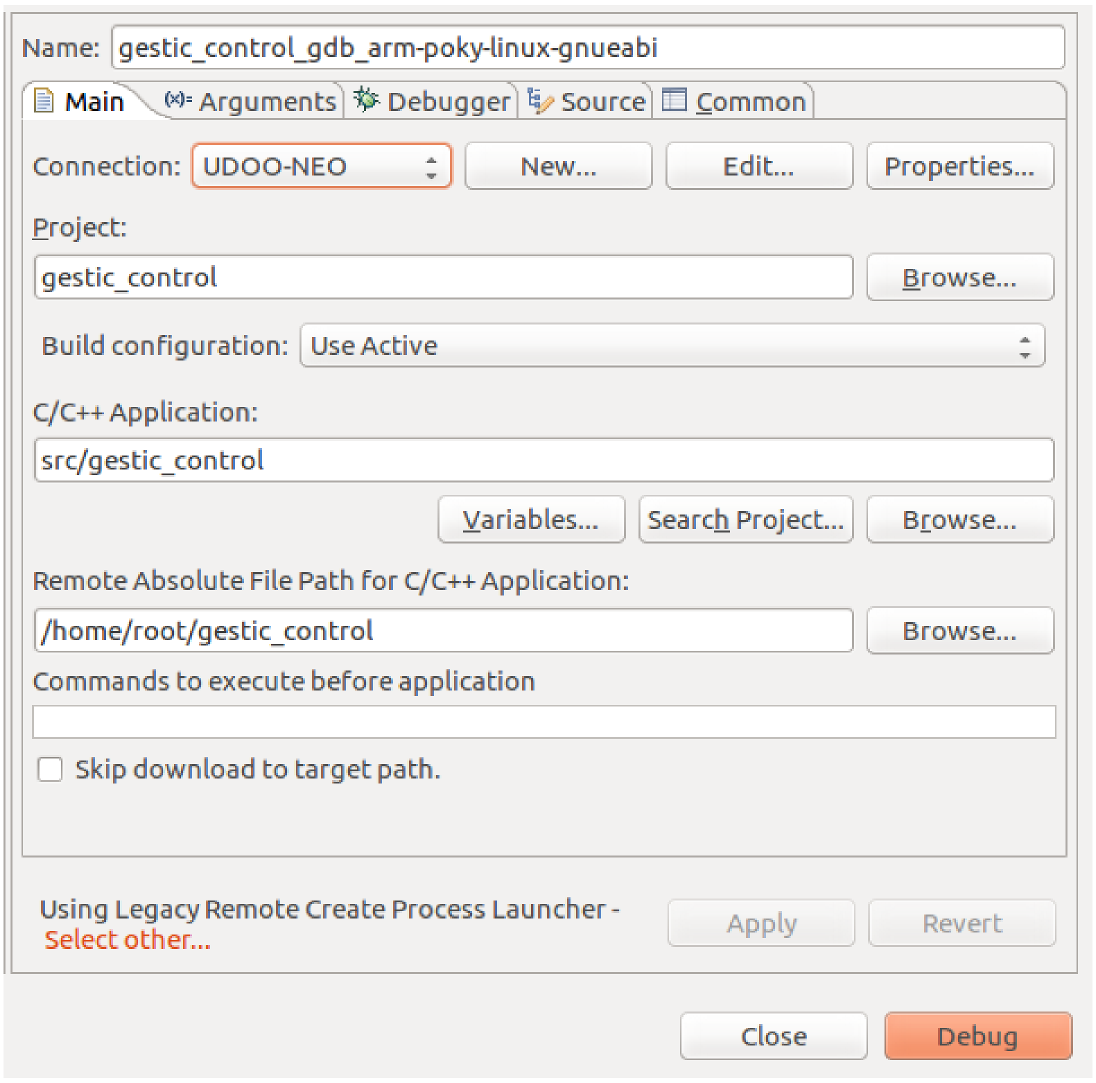
```
...

while(!exitFlag)
{
pthread_mutex_lock(&gesticLock);
if (i > decimation_factor)
i = 0;
//Updating the data
if (gestic_data_stream_update(gestic, NULL) == 0)
{
if(i == 0)
dataUpdated = true;
i++;
}
pthread_mutex_unlock(&gesticLock);
}

...
```

The updating thread is basically intended to attend the following tasks:

1. it tries to get updated values from the GestIC device;
2. it applies a decimation of the values: in case of a fast serial line this is not necessary, in fact in the current version the *decimation_factor* has been set to 0;
3. when the *EXIT* command is received by the *serial_reader_thread* the loop terminates and the thread expires.

5.5 Application Deployment and Debug

When te application is ready the last steps will be to deploy it exploiting the Eclipse IDE extensions downloaded during the previous subsection "Configuring Eclipse and the Yocto SDK"; in order to deploy the application on the board it will be necessary to setup the debug configuration for the *gestic_control* project. This can be done by right clicking on the project name and choose the *Debug As - Debug configurations...*; this will open the debug configuration wizard:


Name: gestic_control_gdb_arm-poky-linux-gnueabi

Connection: UDOO-NEO

Project: gestic_control

Build configuration: Use Active

C/C++ Application: src/gestic_control

Remote Absolute File Path for C/C++ Application: /home/root/gestic_control

Commands to execute before application

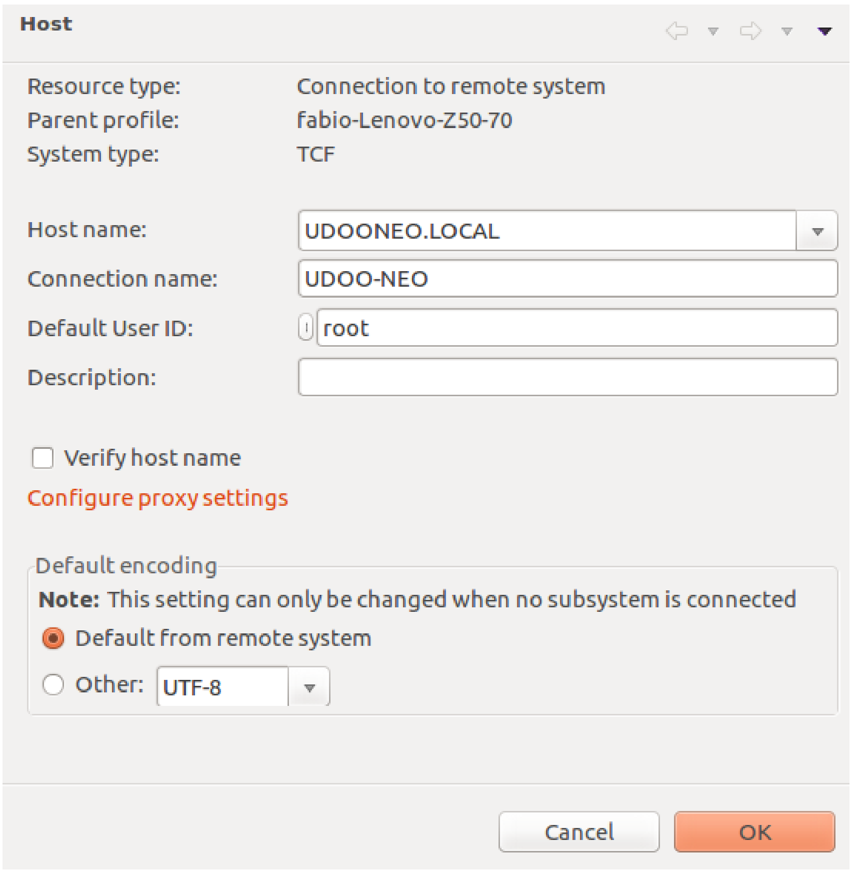
Skip download to target path.

Using Legacy Remote Create Process Launcher - Select other...

Figure 45: Debug Configurations

Within the main configurations tab it is necessary to specify the C/C++ source application file, in this case it can be found at the path *src/gestic_control* of the Eclipse project; the "Remote

Absolute File Path for C/C++ Application" represents the path of application deployment. The choice for the launcher has gone for the Legacy Remote Create Process Luncher embedded within the Eclipse IDE. The last step will be to connect the board to the PC through the Ethernet cable and setup a connection with the UDOO Neo board. This can be done by clicking on the *new...* button near the *Connection...* label and configure the new connection as follows:



Resource type:	Connection to remote system
Parent profile:	fabio-Lenovo-Z50-70
System type:	TCF
Host name:	UDOONEO.LOCAL
Connection name:	UDOO-NEO
Default User ID:	root
Description:	

☐ Verify host name

Configure proxy settings

Default encoding

Note: This setting can only be changed when no subsystem is connected

● Default from remote system

○ Other: UTF-8

Figure 46: Connection Configurations

By pressing the *Debug...* button of the wizard a new cross-debug session will be remotely started on the UDOO Neo Board. Moreover the application executable will be found at the specified path f the UDOO Neo board file system.
Once the application has been deployed on the target system it will be also possible to debug the remaining parts of the device driver. To do this it will be necessary to setup any proper connection between the sensor and the board; then from the host PC the following commands are to be run:

```
root@udooneo:# insmod gestic.ko gestic_debug
root@udooneo:# ./gestic_control & disown
```

By running the application with the *disown* option it will be possible to run it in background and thus the kernel log can be scoped by running the *dmesg* command.

```
root@udooneo:# dmesg


 GestIC: the module is in DEBUG mode
 [ 439.782224] GestIC: character device numbers registration succeded
 [ 439.790126] GestIC: class initialization succeded
 [ 439.796454] GestIC: associated device initialization succeded
 [ 439.802213] GestIC: Initializing the character device data structure
 [ 439.809254] GestIC: character device data structure initialization succeded
 [ 439.816336] GestIC: I2C adapter structure initialization succeded
 [ 439.824961] GestIC: I2C client initialization succeded
 [ 439.830117] GestIC: TS line initialization succeded
 [ 439.835502] GestIC: TS line direction configuration succeded
 [ 439.841174] GestIC: MCLR line initialization succeded
 [ 439.846668] GestIC: MCLR line direction configuration succeded
 [ 439.852513] GestIC: interrupt number detection succeded
 [ 439.858234] GestIC: TS associated interrupt initialization succeded
 [ 439.864542] GestIC: initialization completed
 [ 439.868857] GestIC device: driver installed
 [ 448.047230] GestIC: value TS is : 0
 [ 448.050791] GestIC: interrupt handled
 [ 448.153151] GestIC: reset completed
 [ 448.156818] GestIC: module opened correctly
 [ 448.461541] GestIC: TS in state 0
 [ 448.465047] GestIC: re-asserting TS
 [ 448.507577] GestIC: read succeded
 [ 448.511092]   bytes read: 132 message length: 255
 [ 448.517457] GestIC: read msg[size=132, flags=0, seq=0, id=131]
 [ 448.523474] i2c_master_recv <<< 00000000: 84 00 00 83 aa 63 80 e6 13 64 15 2
 0 32 2e 31 2e .....c...d. 2.1.
 [ 448.534654] i2c_master_recv <<< 00000010: 30 3b 70 3a 48 69 6c 6c 73 74 61 7
 2 56 30 31 3b 0;p:HillstarV01;
 [ 448.544448] i2c_master_recv <<< 00000020: 78 3a 48 69 6c 6c 73 74 61 72 3b 4
 4 53 50 3a 49 x:Hillstar;DSP:I
 [ 448.554606] i2c_master_recv <<< 00000030: 44 39 30 30 31 72 33 36 38 36 3b 6
 9 3a 42 3b 66 D9001r3686;i:B;f
 [ 448.564430] i2c_master_recv <<< 00000040: 3a 32 32 35 30 30 3b 6e 4d 73 67 3
 b 73 3a 74 72 :22500;nMsg;s:tr
 [ 448.574626] i2c_master_recv <<< 00000050: 75 6e 6b 72 32 31 36 33 3a 4d 4f 3
 b 00 00 00 00 unkr2163:MO;....
 [ 448.584413] i2c_master_recv <<< 00000060: 00 00 00 00 00 00 00 00 00 00 00 0
 0 00 00 00 00
 [ 448.594563] i2c_master_recv <<< 00000070: 00 0e 00 00 55 aa 90 65 20 20 80 0
 f 00 00 00 00 ....U..e ......
 [ 448.604348] i2c_master_recv <<< 00000080: 00 00 00 00 ....
 [ 448.633025] GestIC: TS in state 0
 [ 448.636351] GestIC: re-asserting TS
 [ 448.666755] GestIC: read succeded
 [ 448.670080]   bytes read: 26 message length: 255
 [ 448.675118] GestIC: read msg[size=26, flags=0, seq=0, id=145]
 [ 448.680878] i2c_master_recv <<< 00000000: 1a 00 00 91 1f 01 01 81 00 73 00 0
 0 00 00 00 00 .........s......
 [ 448.691026] i2c_master_recv <<< 00000010: 00 00 00 00 00 00 00 00 00 00
 [ 448.713025] GestIC: TS in state 0
 [ 448.716393] GestIC: re-asserting TS
 [ 448.746756] GestIC: read succeded
 [ 448.750080]   bytes read: 26 message length: 255
 [ 448.755189] GestIC: read msg[size=26, flags=0, seq=1, id=145]
 [ 448.760950] i2c_master_recv <<< 00000000: 1a 00 01 91 1f 01 1d 81 00 73 00 0
 0 00 00 00 00 .........s......
 [ 448.771168] i2c_master_recv <<< 00000010: 00 00 00 00 00 00 00 00 00 00
 [ 448.793031] GestIC: TS in state 0
 [ 448.796357] GestIC: re-asserting TS
 [ 448.826656] GestIC: read succeded
 [ 448.829981]   bytes read: 26 message length: 255
 [ 448.835008] GestIC: read msg[size=26, flags=0, seq=2, id=145]
 [ 448.840767] i2c_master_recv <<< 00000000: 1a 00 02 91 1f 01 2c 81 00 73 00 0
 0 00 00 00 00 ......,..s......
 [ 448.850912] i2c_master_recv <<< 00000010: 00 00 00 00 00 00 00 00 00 00
 [ 448.873022] GestIC: TS in state 0
 [ 448.876349] GestIC: re-asserting TS
 [ 448.906756] GestIC: read succeded
 [ 448.910082]   bytes read: 26 message length: 255
 [ 448.915108] GestIC: read msg[size=26, flags=0, seq=3, id=145]
 [ 448.920951] i2c_master_recv <<< 00000000: 1a 00 03 91 1f 01 3c 81 00 73 00 0
 0 00 00 00 00 ......<..s......
 [ 448.931891] i2c_master_recv <<< 00000010: 00 00 00 00 00 00 00 00 00 00
 [ 448.953049] GestIC: sending a message through I2C
 [ 448.959569] GestIC: write succeded
 [ 448.963405]   bytes written: 16 message length: 16
 [ 448.968208]   messaeg header[size=16, flags=0, seq=0, id=162]
 [ 448.974375] i2c_master_send <<< 00000000: 10 00 00 a2 a1 00 00 00 1e 00 00 0
 0 00 00 00 00
```

As can be seen by the Kernel log of the first messages exchanged by the board and the controller the communication works properly; moreover the really first message received by the UDOO Neo after the controller reset is always the firmware version flashed on the MGC3130 controller: this message was useful in the very first tests of the communication since it was used to check weather te messages exchange protocol was respected by the device driver.

6 Conclusions

This document has gone through the steps that brought to the production of a primitive gesture recognition interface based on gesture controller sold by Microchip company. The most of the focus has gone on the software development phase for such an application, spanning from the OS production until the application level design.

An extensive study has been made on the Yocto Project since it represents one of the most useful set of tools in the Embedded field. Thanks to this study has been possible to understand the most basic parts of a complex Operating System as Linux Embedded and to tailor a custom version of this OS for the used hardware. This achievement could obviously be useful when, in future, it could be necessary to setup a unanimated device with a fully functional operating system.

After the study of hardware specifications imposed by the choice for the controller this document showed the implementation of a driver for the MGC3130 controller; thanks to the nature of the environment inside which the driver has born this software is highly configurable and easy to be embedded into another project with a minimum part of re-design mostly related to choice of the GPIO connection with a different development board.

At last the project evolved into the development of an application used to show the functionalities offered by the driver; this application could be re-used in future as a starting point for a complex client application exploiting data received by the controller.

The overhaul result is a set of software sources that can be easily re-used in any Linux based environment willing to offer a gesture recognition like te one here presented; it should be however taken into account improvements on the precision of the position tracking and in the design of the electrode before passing onto a project that may need high level performances or willing to respect safety requirements.

References

[1] Embedded Linux Course, Massimo Violante, DAUIN, Politecnico di Torino.

[2] Embedded Linux Systems with the Yocto Project, Rudolf J. Streif, December 22, 2015, Prentice Hall.

[3] Embedded Linux Development with Yocto Project, Otavio Salvador, Daiane Angolini, Packt Publishing, July 9, 2014

[4] Linux Device Drivers, by Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman, Third Edition, O'Reilly, February 18, 2005.

[5] Linux Kernel In A Nutshell, by Greg Kroah-Hartman, O'Reilly, December 1, 2006.

[6] Acceleglove: A Programmable $500 Hand Sensor, July 10, 2009. Available at `http://www.coolthings.com/acceleglove-a-programmable-500-hand-sensor/`

[7] Gesture Recognition, Wikipedia contributors, Wikipedia, The Free Encyclopedia. Available at `https://en.wikipedia.org/wiki/Gesture_recognition`

[8] Kinect, Wikipedia contributors, Wikipedia, The Free Encyclopedia. Available at `https://en.wikipedia.org/wiki/Kinect`

[9] Real-time operating system, Wikipedia contributors, Wikipedia, The Free Encyclopedia. Available at `https://en.wikipedia.org/wiki/Real-time_operating_system`

[10] What is An RTOS? Available at `https://www.freertos.org/about-RTOS.html`

[11] Linux Boot Process - Step by Step, by P. Kumar, October 19, 2010. Available at `http://blog.adminnote.com/2010/10/linux-boot-process-step-by-step.html`

[12] Das U-Boot, Wikipedia contributors, Wikipedia, The Free Encyclopedia. Available at `https://en.wikipedia.org/wiki/Das_U-Boot`

[13] Booting, Wikipedia contributors, Wikipedia, The Free Encyclopedia. Available at `https://en.wikipedia.org/wiki/Booting#BOOT-LOADER`

[14] The Linux kernel driver - kernel first, Christian, March 26, 2014. Available at `http://www.programering.com/a/MjNzcTMwATA.html`

[15] Microkernel, Wikipedia contributors, Wikipedia, The Free Encyclopedia. Available at `https://en.wikipedia.org/wiki/Microkernel`

[16] USB to TTL adapter. Available at `https://www.techtonics.in/products/usb-to-serial-ttl-converter`

[17] Linux Kernel and Driver Development Training. Availble at `https://bootlin.com/doc/training/linux-kernel/linux-kernel-slides.pdf`

[18] Yocto Project and OpenEmbedded Training. Availble at `https://bootlin.com/doc/training/yocto/yocto-slides.pdf`

[19] Embedded Linux system development. Availble at `https://bootlin.com/doc/training/embedded-linux/embedded-linux-slides.pdf`

[20] UDOO Neo Full documentation. Available at `https://www.udoo.org/docs-neo/Introduction/Introduction.html`

[21] Hillstar Kit Reference Documentation. Available at `http://www.microchip.com/DevelopmentTools/ProductDetails.aspx?PartNO=dm160218`

[22] GestIC® Design Guide, GestIC Design Manual by Microchip. Available at `http://ww1.microchip.com/downloads/en/DeviceDoc/40001716C.pdf`

[23] MGC3130 Hillstar Development Kit Users Guide, Hillstar Kit Reference Manual by Microchip. Available at `http://ww1.microchip.com/downloads/en/DeviceDoc/40001721B.pdf`

[24] MGC3030/3130 GestIC® Library Interface Description Users Guide, MGC3130 Reference Manual by Microchip. Available at `http://ww1.microchip.com/downloads/en/DeviceDoc/40001718E.pdf`

[25] Understanding the I2C Bus, Jonathan Valdez, Jared Becker, Texas Instruments Application Report n. SLVA704, June, 2015. Available at `http://www.ti.com/lit/an/slva704/slva704.pdf`

[26] I2C Bus Specification. Available at `http://i2c.info/i2c-bus-specification`

[27] I2C, Wikipedia contributors, Wikipedia, The Free Encyclopedia. Available at `https://en.wikipedia.org/wiki/I2C`

[28] Linux Device Tree, GitHub contributors. Available at `https://github.com/robbie-cao/kb-openwrt/blob/master/Linux-DT.md`

[29] Anatomy of the Linux virtual file system switch, M. Tim Jones, August 31, 2009. Available at `https://www.ibm.com/developerworks/library/l-virtual-filesystem-switch/`

[30] Virtual file system, Wikipedia contributors, Wikipedia, The Free Encyclopedia. Available at `https://en.wikipedia.org/wiki/Virtual_file_system`

[31] Device Tree for Dummies, Thomas Petazzoni for Free-Electrons, 2013. Available at `https://events.static.linuxfound.org/sites/events/files/slides/petazzoni-device-tree-dummies.pdf`

[32] C++ Compiler Operation. Available at `http://icarus.cs.weber.edu/~dab/cs1410/textbook/1.Basics/compiler_op.html`

[33] Cross-compiling Toolchain, Abhishek Mourya, February 6, 2014. Available at `http://abhishekmourya.blogspot.it/2014/02/cross-compiling-toolchain.html`

[34] Yocto Project Mega-Manual, Scott Rifenbark, 2017-2018 Revision. Available at `https://www.yoctoproject.org/docs/2.4.2/mega-manual/mega-manual.html`