ISSN 2281-4299

DIPARTIMENTO DI INGEGNERIA INFORMATICA
AUTOMATICA E GESTIONALE ANTONIO RUBERTI

SAPIENZA
UNIVERSITÀ DI ROMA

Analysing and Experimenting the Intel Galileo Board for the Internet-Of-Things

Paolo Cocchi

Technical Report n. 12, 2015

Analysing and Experimenting the Intel Galileo Board for the Internet-Of-Things

Paolo Cocchi

Sapienza, University of Rome

Abstract

This paper is a document about the Internet-Of-Things, or IoT. The motivations behind the present work can be found in the wide range of IoT devices appeared on the IT market during the last ten years. What is an IoT device? Is a home desktop PC connected to the Internet a IoT device? If we want to develop a IoT project, what platform is more convenient for us? What is the current state of art of IoT? The aim of our work is to answer these questions. The paper focuses on the Intel Galileo board for the IoT. It is both a microcontroller and an IoT device. It is an x86 Linux embedded system. We exploit capabilities of the board and provide a developing methodology on the platform.

Keywords: Internet-of-Things, IoT, microcontroller, remote controlling, x86, Linux, embedded device.

1 Introduction

1.1 What This Document Is About?

This paper is a document about the Internet-Of-Things, or IoT. What IoT precisely is will be described in the following section. The motivations behind the present work can be found in the wide range of IoT devices appeared on the IT market during the last ten years. But what is an IoT device? Is a home desktop PC connected to the Internet a IoT device? If we want to develop a IoT project, what platform is more convenient for us? What is the current state of art of IoT? The aim of our work is to answer these questions.

We begin in the next section with a description of what IoT means.

In Chapter 2 we make an overview of the current panorama of IoT devices available on the market. Due to the fact that tens of devices exist, we realized a comparison between five devices which can be taken as representatives of macro-types of IoT platforms. First, this comparison is useful to link the theory of IoT (which follows) to the real world and to the actual applications of the IoT concept. Second, this compare leads to the introduction of the IoT platform on which we focus and realize our experimentation, i.e. the Intel Galileo Gen 2 board for IoT.

Chapter 3 is a detailed analysis of the Intel Galileo board. Chapters 4 and 5 deal with the setup of the Intel Galileo platform, and are introductory to Chapter 6 and Chapter 7 in which we describe tests and results we achieved during our experimentation on the board.

Chapter 8 states our Thesis: what is our vision of Internet-Of-Things at the beginning of 2015 and what we can and what we cannot realize on a platform as the Intel Galileo board as result of our experimentation.

1.2 A Brief Overview About Internet-Of-Things

In telecommunications Internet of things (or, more properly, the Internet-Of-Things or IoT, acronym for the Internet-Of-Things) is a neologism referring to the extension of the Internet to the world of objects and concrete places. The purpose of IoT is to provide an advanced connectivity of systems, services and devices that overcomes the simple idea of communication between a machine and another one, including also a wide range of applications and domains. Reference may be made to a disparate panorama of devices such as the incorporated sensors of cars, implants able to check heartbeats or devices that can find application in a domestic context like smart thermostat systems which can be remotely controlled through the use of a LAN or Internet connection. All of these applications become a reality through the use of already existing technologies allowing the data flow between different devices.

Internet of things is seen as a possible evolution of the use of the Internet. The objects are made recognizable and they acquire intelligence thanks to the fact that they are able to communicate information about themselves and access to aggregated information from others. Alarm clocks go off before in case of traffic, plants can communicate to the watering system when it is the moment to be watered, sneakers transmit times, speed and distance to compete in real time with people on the other side of the globe. Every object can acquire a proactive role thanks to the connection to the Internet.

The aim of Internet-Of-Things is to ensure that the electronic world traces a map of the real one, giving an electronic identity to things and places in the physical environment.

Some history

The idea of creating a network of smart devices dates back to 1982. In that year, at Carnegie Mellon University, was set up a modified Coke machine which was the first Internet-connected device, capable to figure out if the newly loaded drinks were cold.

Nevertheless, the notion of Internet of Things became popular only in 1999 through the Auto-ID Center at MIT and the related market-analysis publications. The MIT Auto-ID Center was created to develop the Electronic Product Code, a global RFID-based item identification system. Radio-frequency identification (RFID) was considered as a necessary condition for the further development of Internet of Things in modern day. If all objects and people in daily life were equipped with identifiers, computers could manage and inventory them. Besides using RFID, at the beginning of 2015 the tagging of things may be achieved through technologies as near field communication, barcodes and QR codes.

From 2014, the concept of the Internet-Of-Things has evolved as a result of a convergence of multiple technologies ranging from wireless communications to micro-electromechanical systems.

Enabling technologies

In the vision of the Internet of things, objects create a pervasive and interconnected network using multiple communication technologies. RFID tags have been one of the first examples in this area. However, over time, new technologies born that could improve the efficiency of communication between objects. Between them stands the IEEE 802.15.4 and, above all, its recent improvement IEEE 802.15.4e which can greatly increase the reliability of wireless links and the efficiency of radio frequency energy, thanks to the adoption of the access mechanism of Time Slotted Channel Hopping. These technologies of lowest level, when integrated to the IP-based protocol architectures, may give rise to a concrete vision of the Internet of things, being able to communicate with the nodes of the Internet. In this sense, it is important to mention the IETF 6LoWPAN, RPL, and CoAP, which are protocols able to operatively create an IP network of objects that can communicate with the Internet to create new services in multiple application domains.

Objects and places provided with a Radio-frequency (RFID) or QR codes ID can also communicate information on mobile devices such as mobile phones.

Applications

The Internet-Of-Things concept finds application in almost every area. In fact, such systems could be responsible of the collection of information in environments ranging from natural ecosystems to cities, thus finding applications in the fields of environmental sensing and urban planning.

We can also identify other many examples of sensing and actuating which are possible as result of this technology: we are talking about applications concerning electricity, energy and heat management.

The main application domains and operational areas affected by the development of IoT are summarized in the following list:

1. Home Automation and Smart Home
2. Robotics
3. Avionics
4. Automotive industry
5. Biomedical
6. Monitoring in factories
7. Telemetry
8. Wireless sensor networks
9. Surveillance
10. Forecasting and Detecting of adverse events

11. Smart Grid and Smart City
12. Embedded Systems
13. Telematics

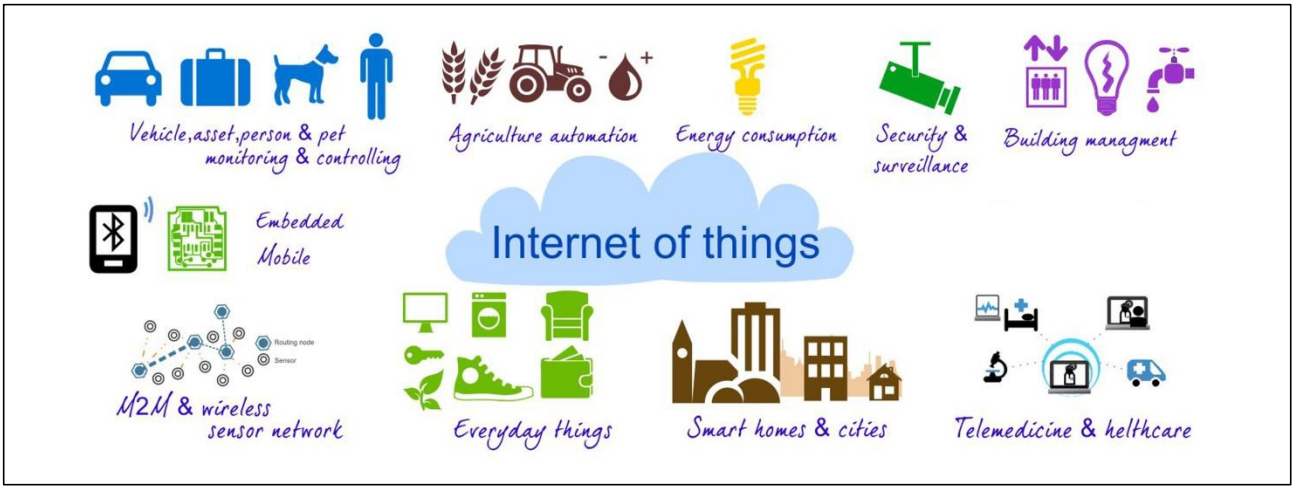

Figure 1-1. *IoT application areas.*

Figure 1.1 shows the wide range of Internet-Of-Things application areas.

Future expectations

According to Gartner Inc. (a multinational world leader in strategic consulting, research and analysis in the field of Information Technology), in 2020 there will be 26 billion connected objects globally. ABI Research (a market research and market intelligence company) estimates that will be more than 30 billion. Other institutions speak of 100 billion.

The expectations of the experts are that the Internet-Of-Things will change the way we live in a radical manner. Intelligent items, with decision-making capacity, will enable energy savings both at local and personal level (Home Automation and Smart Home) and at the macro level (Smart City and Smart Grid). Figure 1.2 provides a simple idea of what Home Automation and Smart Home mean. The link between these "local" concepts and the "global" ideas of Smart City and Smart Grid is represented in figure 1.3. The "Smart Grid" label at the top right corner of picture 1.2 is conceptually linked to the centralized green Smart Grid at the center of figure 1.3.

Figure 1-2. *IoT implemented as Home Automation and Smart Home.*

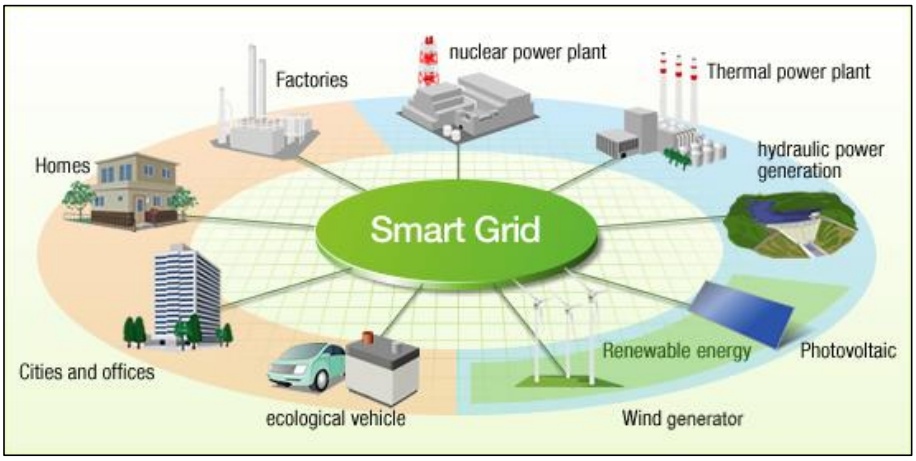

Figure 1-3. *The "global" view of Smart Grid.*

Integration with Internet involves using unique IP. IPv4 allows for 4.3 billion unique addresses, which is why the developers of IoT devices are adopting the IPv6 standard, which allows to reach many more addresses.

Another step in the Internet-Of-Things evolution is supposed to be the development of existing IoT systems in order to achieve a greater speed of data diffusion and a greater capability of elaboration and recollection of this data. Enabling Internet on billions of devices and collecting the big amount of data retrieved from these devices is not a problem with a trivial solution. This concept links the world of IoT to the topics of Big Data and Distributed Systems management.

Criticism and controversies

The biggest criticisms to be highlighted when we talk about Internet-Of-Things concerns two aspects: security and privacy.

Referring to the first aspect, most of the companies are working on solutions which will improve devices safety from hacker attacks, even if we can expect that antivirus or other protection systems will be necessary in the future not only for computers but also for objects of different types.

As we know, the unique one hundred percent secure IT system is the offline system. When we connect a wide range of objects to the Internet, we have to ponder about the possible drawbacks. If we have a remote controlled garage door installed at home, we know that if the system is compromised by an hacker we could not get out our car. This could be a thought as an unreal example: why an hacker could compromise our home automation system? Who could gain something from such action? The answer to these questions is that actually the Internet is the ground on which the so called "viruses" grow exponentially day by day (this is reported in the 2014 annual documents of the most important antivirus producers), often in the form of botnets (complex distributed systems of "warms") which spread to generic Internet connected systems.

Furthermore, and we will detail the concept in the next chapter when we will talk about hardware and software specification of the IoT platform, the most part of the IoT devices is equipped with general purpose (Linux) or general purpose derived systems (like Android).

So, when connected to the Internet, our home objects and devices suffers the same IT security problems of our old desktop PC. Figure 1.4 shows the Verizon detected most widespread IT system threats during the last year (2014).

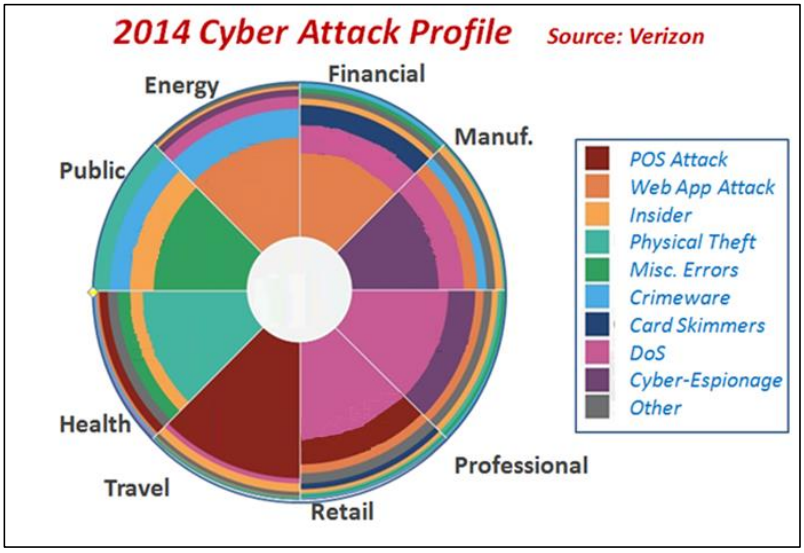

Figure 1-4. *Cyber attacks during 2014.*

The problem of objects security is closely linked with the issue of privacy. Many privacy challenges linked to the Internet-Of-Things have been identified by researchers to preserve user privacy in all circumstances. These challenges are mostly the assurance and guarantee of anonymity, user consent and freedom of choice.

Criticism and controversies are normal aspects of every technological revolution. And we believe that the Internet-Of-Things is the future in our everyday life experience.

At this point we have only an initial theoretical idea of what is the Internet-Of-Things. In the following Chapter 2 we begin with a technological overview of those devices which make possible the complete realization of IoT in our real life.

2 The Boards World

2.1 Introduction

As said at the end of the previous chapter after a brief description of the Internet-Of-Things, the second step to our learning of the matter is an analysis of the physical devices which make possible the realization of IoT in the real life.

This chapter is an overview of the current state of the world of programmable cards.

The first goal of the section is to clarify the differences between the tens of platform which are on the market. To simplify the description, we make a first reduction of the big set of platforms to two macro-types: the microcontroller and the mini PC. The second simplification we make is to choose five platforms which cover the features of the whole board set. Each of this five devices has characteristics more less near to the microcontroller or to the mini PC macro-types, and we highlight this features.

The second aim of our comparison is to place the platform on which we focus in the following of this paper (i.e., the Intel Galileo board) in the right position into the board panorama.

2.2 Boards Overview

Since 2005 with the first outcome of the Arduino board, many embedded systems for the Internet-Of-Things appeared on the market. The big set of these boards is really heterogeneous.

Some of them, such as Arduino, are programmable cards precisely designed to handle input from sensors and output to actuators (both analog and digital). These are programmable devices designed to deal with the external environment. We put them into the macro-type of microcontroller.

Others (like the BeagleBoard we describe in the following) are real mini PC with good performance. But the major part of the board set is represented by platforms which have characteristics of both types. So, we felt a bit confusing the first time we explored the world of the IoT devices. If a mini PC is a platform for the Internet-Of-Things, also a small size desktop PC could be considered at the same manner?

As we see in the following, actually all the platform we analyse are IoT devices in some way. Some of them are more IoT microcontrollers than mini PCs, others are more useful and powered to be applied as mini PC (for example as server for remote controlling). Some, as our Intel Galileo, try to be both kind at the same time.

Despite a lot of differences between one platform to another, there are some characteristics shared by all these devices and which make the difference of these IoT platform from a standard PC: the low size, weight and power consumption. And, not least because it is less important, they are all featured with some pin interface for attaching external devices (sensor or actuator) to deal with the environment they are placed.

We describe our five selected IoT platform from the hardware to the software side. Our candidates are the Arduino platform, the BeagleBoard, the Raspberry Pi, the UDOO board and the Intel Galileo platform.

We present Arduino as representative of the programmable microcontroller macro-type; BeagleBoard and Raspberry Pi for the mini PC macro-type (we actually see as the Raspberry Pi comes as a mini PC but approaching the world of microcontrollers); UDOO as the perfect mix of the two families of boards.

Finally we see where the new Intel Galileo fits, as the only x86 Internet-Of-Things embedded platform.

All sections 2.2.x which follows deal with a detailed description of the features of these five boards. The reader not interested in these details could skip to section 2.3 for a detailed comparison between them on the basis of their technical skills and to section 2.4 where we summarize what is the current state of the art of the IoT boards and we place the Intel Galileo platform into its natural place.

2.2.1 ARDUINO

Arduino is an open-source hardware designed and manufactured as standard for building digital devices that can sense and control the physical world.

The first Arduino was introduced in 2005. The project target was to provide an inexpensive and easy way for hobbyists, students, and professionals to create devices that interact with the environment using sensors and actuators. Common examples for beginner hobbyists include simple robots, thermostats and motion detectors.

Arduino boards may be purchased preassembled, or as do-it-yourself kits. The hardware design information is available for those who would like to assemble an Arduino by hand. Adafruit Industries estimated in mid-2011 that over 300,000 official Arduinos had been commercially produced, and in 2013 that 700,000 official boards were in users' hands.

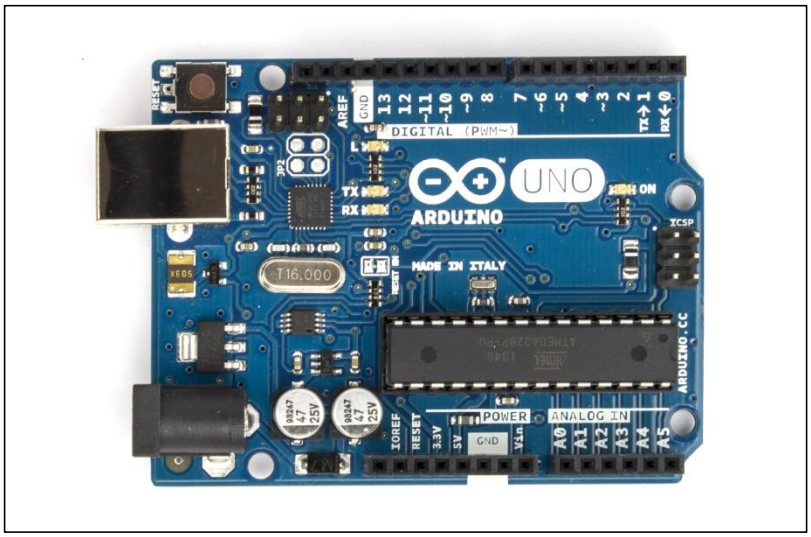

Figure 2-1. *The Arduino UNO*

Hardware

An Arduino board consists of an Atmel 8-bit AVR microcontroller with complementary components that facilitate programming and incorporation into other circuits. An important aspect of the Arduino is its standard connectors, which lets users connect the board to a variety of interchangeable add-on modules called “shields”. Some shields communicate with the Arduino board directly over various pins, but many shields are individually addressable via an I²C serial bus - so many shields can be stacked and used in parallel. Official Arduinos use the megaAVR series of chips, specifically the ATmega8, ATmega168, ATmega328, ATmega1280, and ATmega2560. A handful of other processors have been used by Arduino compatibles. Most boards include a 5 volt linear regulator and a 16 MHz crystal oscillator (or ceramic resonator in some variants), although some designs such as the LilyPad run at 8 MHz and dispense with the onboard voltage regulator due to specific form-factor restrictions. An Arduino's microcontroller is also pre-programmed with a boot loader that simplifies uploading of programs to the on-chip flash memory, compared with other devices that typically need an external programmer. This makes using an Arduino more straightforward by allowing the use of an ordinary computer as the programmer.

At a conceptual level, when using the Arduino software stack, all boards are programmed over an RS-232 serial connection, but the way this is implemented varies by hardware version. Serial Arduino boards contain a level shifter circuit to convert between RS-232-level and TTL-level signals. Current Arduino boards are

programmed via USB, implemented using USB-to-serial adapter chips such as the FTDI FT232. Some variants, such as the Arduino Mini and the unofficial Boarduino, use a detachable USB-to-serial adapter board or cable, Bluetooth or other methods. (When used with traditional microcontroller tools instead of the Arduino IDE, standard AVR ISP programming is used.)

The Arduino board exposes most of the microcontroller's I/O pins for use by other circuits. The Diecimila, Duemilanove, and current Uno provide 14 digital I/O pins, six of which can produce pulse-width modulated signals, and six analog inputs, which can also be used as six digital I/O pins. These pins are on the top of the board, via female 0.10-inch (2.5 mm) headers. Several plug-in application shields are also commercially available. The Arduino Nano, and Arduino-compatible Bare Bones Board and Boarduino boards may provide male header pins on the underside of the board that can plug into solderless breadboards.

There are many Arduino-compatible and Arduino-derived boards. Some are functionally equivalent to an Arduino and can be used interchangeably. Many enhance the basic Arduino by adding output drivers, often for use in school-level education to simplify the construction of buggies and small robots. Others are electrically equivalent but change the form factor - sometimes retaining compatibility with shields, sometimes not. Some variants use completely different processors, with varying levels of compatibility.

The first Arduino was designed as pure microcontroller for sensor driving, without any interface for net connections. During the years, the most part of the Arduino boards began to mount Ethernet and/or Wireless interfaces for Internet connection and network expansion shields appeared for those Arduinos which had no on-board network adapters. This is how the Arduino entered the world of the Internet-Of-Things.

Software

The Arduino integrated development environment (IDE) is a cross-platform application written in Java, and derives from the IDE for the Processing programming language and the Wiring projects. It is designed to introduce programming to artists and other newcomers unfamiliar with software development. It includes a code editor with features such as syntax highlighting, brace matching, and automatic indentation, and is also capable of compiling and uploading programs to the board with a single click. A program or code written for Arduino is called “sketch”.

Arduino programs are written in C or C++. The Arduino IDE comes with a software library called "Wiring" from the original Wiring project, which makes many common input/output operations much easier. Users only need define two functions to make a runnable cyclic executive program:

- setup(): a function run once at the start of a program that can initialize settings
- loop(): a function called repeatedly until the board powers off

A typical first program for a microcontroller simply blinks an LED on and off. In the Arduino environment, the user might write a program like this:

```
#define LED_PIN 13
void setup () {
pinMode (LED_PIN, OUTPUT); // Enable pin 13 for digital output
}
void loop () {
digitalWrite (LED_PIN, HIGH); // Turn on the LED
delay (1000); // Wait one second (1000 milliseconds)
```

digitalWrite (LED_PIN, LOW); // Turn off the LED

delay (1000); // Wait one second

}

The previous code would not be seen by a standard C++ compiler as a valid program, so when the user clicks the "Upload to I/O board" button in the IDE, a copy of the code is written to a temporary file with an extra include header at the top and a very simple *"main()"* function at the bottom, to make it a valid C++ program.

The Arduino IDE uses the GNU toolchain and AVR Libc to compile programs, and uploads programs to the board. Also, Arduino developers can use in programming their sketches the C/C++ AVR API provided bu the Arduino IDE.

2.2.2 RASPBERRY PI

The Raspberry Pi is a credit card-sized single-board computer developed in the UK by the Raspberry Pi Foundation with the intention of promoting the teaching of basic computer science in schools.

The Raspberry Pi is manufactured in four board configurations through licensed manufacturing agreements with Newark element14 (Premier Farnell), RS Components and Egoman. These companies sell the Raspberry Pi online. Egoman produces a version for distribution solely in China and Taiwan, which can be distinguished from other Pis by their red coloring and lack of FCC/CE marks. The hardware is the same across all manufacturers.

The Raspberry Pi is based on the Broadcom BCM2835 system on a chip (SoC), which includes an ARM1176JZF-S 700 MHz processor, VideoCore IV GPU, and was originally shipped with 256 megabytes of RAM, later upgraded (Model B & Model B+) to 512 MB. The system has Secure Digital (SD) or MicroSD (Model A+ and B+) sockets for boot media and persistent storage.

In 2014, the Raspberry Pi Foundation launched the Compute Module, which packages a BCM2835 with 512MB RAM and an eMMC FLASH chip into a module for use as a part of embedded systems.

The Foundation provides Debian and Arch Linux ARM distributions for download. Tools are available for Python as the main programming language, with support for BBC BASIC (via the RISC OS image or the Brandy Basic clone for Linux), C, C++, Java, Perl and Ruby.

As of October 2014, about 3.8 million boards had been sold.

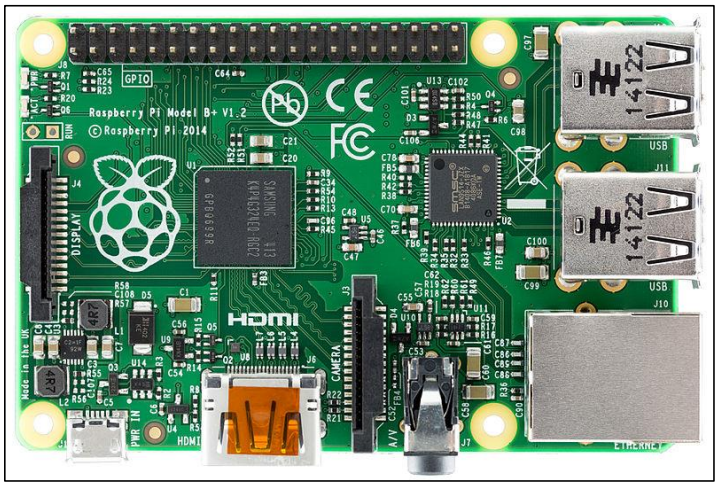

Figure 2-2. *The Raspberry Pi Model B+*

Hardware

Processor

The SoC used in the Raspberry Pi is somewhat equivalent to the chip used in older smartphones (Android or iPhone / 3G / 3GS). The Raspberry Pi is based on the Broadcom BCM2835 system on a chip (SoC), which includes an 700 MHz ARM1176JZF-S processor, VideoCore IV GPU, and RAM. It has a Level 2 cache of 128 KB, used primarily by the GPU, not the CPU. The SoC is stacked underneath the RAM chip, so only its edge is visible.

While operating at 700 MHz by default, the Raspberry Pi provides a real world performance roughly equivalent to 0.041 GFLOPS. On the CPU level the performance is similar to a 300 MHz Pentium II of 1997-1999. The GPU provides 1 Gpixel/s or 1.5 Gtexel/s of graphics processing or 24 GFLOPS of general purpose computing performance. The graphics capabilities of the Raspberry Pi are roughly equivalent to the level of performance of the Xbox of 2001.

Overclocking

The Raspberry Pi chip operates at 700 MHz by default and doesn't become hot enough to need a heatsink or special cooling, unless the chip is overclocked.

Most Raspberry Pi chips can be overclocked to 800 MHz and some even higher to 1000 MHz. In the Raspbian Linux distro the overclocking options on boot can be done by a software command running "sudo raspi-config" without voiding the warranty. In those cases the Pi automatically shuts the overclocking down in case the chip reaches 85°C (185°F), but it is possible to overrule automatic over voltage and overclocking settings (voiding the warranty). In that case, one can try putting an appropriately sized heatsink on it to keep the chip from heating up far above 85 °C.

Newer versions of the firmware contain the option to choose between five overclock ("turbo") presets that when turned on try to get the most performance out of the SoC without impairing the lifetime of the Pi. This is done by monitoring the core temperature of the chip, and the CPU load, and dynamically adjusting clock speeds and the core voltage. When the demand is low on the CPU, or it is running too hot, the performance is throttled, but if the CPU has much to do, and the chip's temperature is acceptable, performance is temporarily increased, with clock speeds of up to 1 GHz, depending on the individual board, and on which of the turbo settings is used. The five settings are:

- None; 700 MHz ARM, 250 MHz core, 400 MHz SDRAM, 0 overvolt,
- Modest; 800 MHz ARM, 250 MHz core, 400 MHz SDRAM, 0 overvolt,
- Medium; 900 MHz ARM, 250 MHz core, 450 MHz SDRAM, 2 overvolt,
- High; 950 MHz ARM, 250 MHz core, 450 MHz SDRAM, 6 overvolt,
- Turbo; 1000 MHz ARM, 500 MHz core, 600 MHz SDRAM, 6 overvolt.

In the highest (turbo) preset the SDRAM clock was originally 500 MHz, but this was later changed to 600 MHz because 500 MHz sometimes causes SD card corruption. Simultaneously in high mode the core clock speed was lowered from 450 to 250 MHz, and in medium mode from 333 to 250 MHz.

RAM

On the older beta model B boards, 128 MB was allocated by default to the GPU, leaving 128 MB for the CPU. On the first 256 MB release model B (and Model A), three different splits were possible. The default

split was 192 MB (CPU RAM), which should be sufficient for standalone 1080p video decoding, or for simple 3D, but probably not for both together. 224 MB was for Linux only, with just a 1080p framebuffer, and was likely to fail for any video or 3D. 128 MB was for heavy 3D, possibly also with video decoding (e.g. XBMC). Comparatively the Nokia 701 uses 128 MB for the Broadcom VideoCore IV. For the new model B with 512 MB RAM initially there were new standard memory split files released (arm256_start.elf, arm384_start.elf, arm496_start.elf) for 256 MB, 384 MB and 496 MB CPU RAM (and 256 MB, 128 MB and 16 MB video RAM). But a week or so later the RPF released a new version of start.elf that could read a new entry in config.txt (gpu_mem=xx) and could dynamically assign an amount of RAM (from 16 to 256 MB in 8 MB steps) to the GPU, so the older method of memory splits became obsolete, and a single start.elf worked the same for 256 and 512 MB Pis.

Networking

Though the Model A and A+ do not have an 8P8C ("RJ45") Ethernet port, it can be connected to a network using an external user-supplied USB Ethernet or Wi-Fi adapter. On the model B and B+ the Ethernet port is provided by a built-in USB Ethernet adapter.

Peripherals

Generic USB keyboards and mice are compatible with the Raspberry Pi.

Video

The video controller is capable of standard modern TV resolutions, such as HD and Full HD, and older standard CRT TV resolutions.

Low-level peripherals details

RPi A+,B+ have GPIO J8 40-pin pinout. Models A and B have only the first 26 pins.

Model B rev 2 also has a pad P6 of 8 pins offering access to an additional 4 GPIO connections.

Models A and B provide GPIO access to the ACT status LED using GPIO 16. Models A+ and B+ provide GPIO access to the ACT status LED using GPIO 47, and the Power status LED using GPIO 35.

Accessories

- Camera. On 14 May 2013, the foundation and the distributors RS Components & Premier Farnell/Element 14 launched the Raspberry Pi camera board with a firmware update to accommodate it. The camera board is shipped with a flexible flat cable that plugs into the CSI connector located between the Ethernet and HDMI ports. In Raspbian, one enables the system to use the camera board by the installing or upgrading to the latest version of the OS and then running Raspi-config and selecting the camera option. The cost of the camera module is 20 EUR in Europe (9 September 2013). It can produce 1080p, 720p, 640x480p video. The footprint dimensions are 25 mm x 20 mm x 9 mm.
- Gertboard. A Raspberry Pi Foundation sanctioned device designed for educational purposes, and expands the Raspberry Pi's GPIO pins to allow interface with and control of LEDs, switches, analog signals, sensors and other devices. It also includes an optional Arduino compatible controller to interface with the Pi.
- Infrared Camera. In October 2013, the foundation announced that they would begin producing a camera module without an infrared filter, called the Pi NoIR.

- HAT (Hardware Attached on Top) expansion boards. Together with the model B+, inspired by the Arduino shield boards, were devised by the Raspberry PI Foundation. Each HAT board carries a small EEPROM (typically a CAT24C32WI-GT3) containing the relevant details of the board, so that the Raspberry PI's OS is informed of the HAT, and the technical details of it, relevant to the OS using the HAT.

Software

Operating systems

The Raspberry Pi primarily uses Linux-kernel-based operating systems.

The ARM11 chip at the heart of the Pi is based on version 6 of the ARM. The current releases of several popular versions of Linux, including Ubuntu, will not run on the ARM11. It is not possible to run Windows on the Raspberry Pi.

The install manager for the Raspberry Pi is NOOBS. The operating systems included with NOOBS are:

- Archlinux ARM
- OpenELEC
- Pidora (Fedora Remix)
- Puppy Linux
- Raspbmc and the XBMC open source digital media center
- RISC OS – The operating system of the first ARM-based computer
- Raspbian (recommended) – Maintained independently of the Foundation; based on the ARM hard-float (armhf) Debian 7 'Wheezy' architecture port originally designed for ARMv7 and later processors (with Jazelle RCT/ThumbEE, VFPv3, and NEON SIMD extensions), compiled for the more limited ARMv6 instruction set of the Raspberry Pi. A minimum size of 4 GB SD card is required.

Other operating systems are:

- Xbian and the XBMC open source digital media center
- openSUSE
- Raspberry Pi Fedora Remix
- Slackware ARM – Version 13.37 and later runs on the Raspberry Pi without modification. The 128–496 MB of available memory on the Raspberry Pi is at least twice the minimum requirement of 64 MB needed to run Slackware Linux on an ARM or i386 system. (Whereas the majority of Linux systems boot into a graphical user interface, Slackware's default user environment is the textual shell / command line interface.) The Fluxbox window manager running under the X Window System requires an additional 48 MB of RAM.
- FreeBSD and NetBSD
- Plan 9 from Bell Labs and Inferno (in beta)
- Moebius – A light ARM HF distribution based on Debian. It uses Raspbian repository, but it fits in a 1 GB SD card. It has just minimal services and its memory usage is optimized to keep a small footprint.
- OpenWrt – Primarily used on embedded devices to route network traffic.
- Kali Linux – A Debian-derived distro designed for digital forensics and penetration testing.
- Instant WebKiosk – An operating system for digital signage purposes (web and media views)

- Ark OS – Website and email self-hosting
- Minepion – Dedicated operating system for mining cryptocurrency
- Kano OS http://kano.me/downloads
- Nard SDK For industrial embedded systems

Driver APIs

Raspberry Pi can use a VideoCore IV GPU via a binary blob, which is loaded into the GPU at boot time from the SD-card, and additional software, that initially was closed source. This part of the driver code was later released, however much of the actual driver work is done using the closed source GPU code. Application software uses calls to closed source run-time libraries (OpenMax, OpenGL ES or OpenVG) which in turn calls an open source driver inside the Linux kernel, which then calls the closed source Videocore IV GPU driver code. The API of the kernel driver is specific for these closed libraries. Video applications use OpenMAX, 3D applications use OpenGL ES and 2D applications use OpenVG which both in turn use EGL. OpenMAX and EGL use the open source kernel driver in turn.

Third party application software

- Mathematica – Since 21 November 2013, Raspbian includes a full installation of this proprietary software for free. As of 1 August 2014 the version is Mathematica 10.
- Minecraft – Released 11 February 2013; a version for the Raspberry Pi, in which you can modify the game world with code.

2.2.3 BEAGLEBOARD

The BeagleBoard is a low-power open-source hardware single-board computer produced by Texas Instruments in association with Digi-Key and Newark element14. The BeagleBoard was also designed with open source software development in mind, and as a way of demonstrating the Texas Instrument's OMAP3530 system-on-a-chip. The board was developed by a small team of engineers as an educational board that could be used in colleges around the world to teach open source hardware and software capabilities. It is also sold to the public under the Creative Commons share-alike license. The board was designed using Cadence OrCAD for schematics and Cadence Allegro for PCB manufacturing; no simulation software was used.

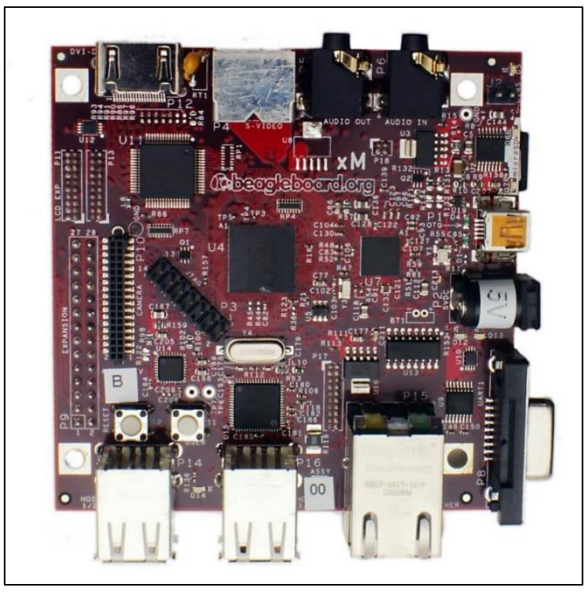

Figure 2-3. *The BeagleBoard-xM*

Hardware (-xM board)

The BeagleBoard measures approximately 75 by 75 mm and has all the functionality of a basic computer. The OMAP3530 includes an ARM Cortex-A8 CPU (which can run Linux, Minix, FreeBSD, OpenBSD, RISC OS, or Symbian; Android is also being ported), a TMS320C64x+ DSP for accelerated video and audio decoding, and an Imagination Technologies PowerVR SGX530 GPU to provide accelerated 2D and 3D rendering that supports OpenGL ES 2.0. Video out is provided through separate S-Video and HDMI connections. A single SD/MMC card slot supporting SDIO, a USB On-The-Go port, an RS-232 serial connection, a JTAG connection, and two stereo 3.5 mm jacks for audio in/out are provided.

Built-in storage and memory are provided through a PoP chip that includes 256 MB of NAND flash memory and 256 MB of RAM (128 MB on earlier models).

The board uses up to 2 W of power and can be powered from the USB connector, or a separate 5 V power supply. Because of the low power consumption, no additional cooling or heat sinks are required.

A modified version of the BeagleBoard called the BeagleBoard-xM started shipping on August 27, 2010. The BeagleBoard-xM measures in at 82.55 by 82.55 mm and has a faster CPU core (clocked at 1 GHz compared to the 720 MHz of the BeagleBoard), more RAM (512 MB compared to 256 MB), onboard Ethernet jack, and 4 port USB hub. The BeagleBoard-xM lacks the onboard NAND and therefore requires the OS and other data to be stored on a microSD card. The addition of the Camera port to the -xM provides a simple way of importing video via Leopard Board cameras.

The following list enumerates the Raspberry hardware features:

- Package on Package POP CPU/memory chip.
    - Processor TI DM3730 Processor - 1 GHz ARM Cortex-A8 core
    - 'HD capable' TMS320C64x+ core (800 MHz up to 720p @30 fps)
    - Imagination Technologies PowerVR SGX 2D/3D graphics processor supporting dual independent displays[8]
    - 512 MB LPDDR RAM
    - 4 GB microSD card supplied with the BeagleBoard-xM and loaded with The Angstrom Distribution
- Peripheral connections
    - DVI-D (HDMI connector chosen for size - maximum resolution is 1400x1050)
    - S-Video
    - USB OTG (mini AB)
    - 4 USB ports
    - Ethernet port
    - MicroSD/MMC card slot
    - Stereo in and out jacks
    - RS-232 port
    - JTAG connector
    - Power socket (5 V barrel connector type)
    - Camera port

    - Expansion port
- Optional expansion boards
    - BeagleBoard Zippy - Feature expander daughter card for BeagleBoard
    - BeagleBoard Zippy2 - Second-generation Zippy. (UART, EEPROM, 100BASE-T, SD-Slot, RTC, I²C (5 V))
    - BeagleTouch Display - Touchscreen 4.3" OLED panel with touchscreen, and drivers for Angstrom Linux built by Liquidware.
    - BeagleLCD2 Expansion Board - 4.3" wide aspect LCD panel + touchscreen with interface board. Developed by HY Research.
    - BeagleJuice - Lithium-ion battery pack for portability developed and built by Liquidware.
    - WLAN adapter - This additional expansion card enables wireless connectivity functionality for the BeagleBoard.
    - BeadaFrame - 7" TFT LCD display kit includes touch panel and a plastic frame, by NAXING Electronics.
    - 4DLCD CAPE - 4.3", 480x272 resolution LCD cape with resistive touch or non-touch and seven push buttons
    - Vifff-024 - a very sensitive camera allowing capture of video stream at quarter moon illumination. Developed by ViSensi.org.

Software

As a complete ARM computer, BeagleBoard can be equipped with all ARM compilable software.

The board has been demonstrated using Android, Angstrom Linux, Fedora, Ubuntu, VxWorks, Gentoo, Arch Linux ARM and Maemo Linux distributions, FreeBSD, the Windows CE operating system, Symbian, QNX and a version of RISC OS 5 made available by RISC OS Open.

2.2.4 UDOO

The UDOO is a single-board computer integrated with a microcontroller Arduino 2 compatible, designed for the education of computer science, the world of Makers and the Internet of Things.

The product was launched on Kickstarter at April 2013 reaching wide consensus. 3 models of UDOO available: UDOO Dual Basic, UDOO Dual, UDOO Quad, respectively at the price of $99, $115, $135.

UDOO is a development platform that merges a Dual or Quad Core ARM Freescale Cortex-A9 i.MX 6 CPU, with optimal performances both on Linux and on Android operating system, and an Arduino 2 compatible board, embedded with a dedicated ARM Atmel SAM3X8E CPU.

UDOO is a joint effort of Aidilab srl and SECO USA Inc, in collaboration with a multidisciplinary team of researchers with expertise in interaction design, embedded systems, sensor networks and cognitive science.

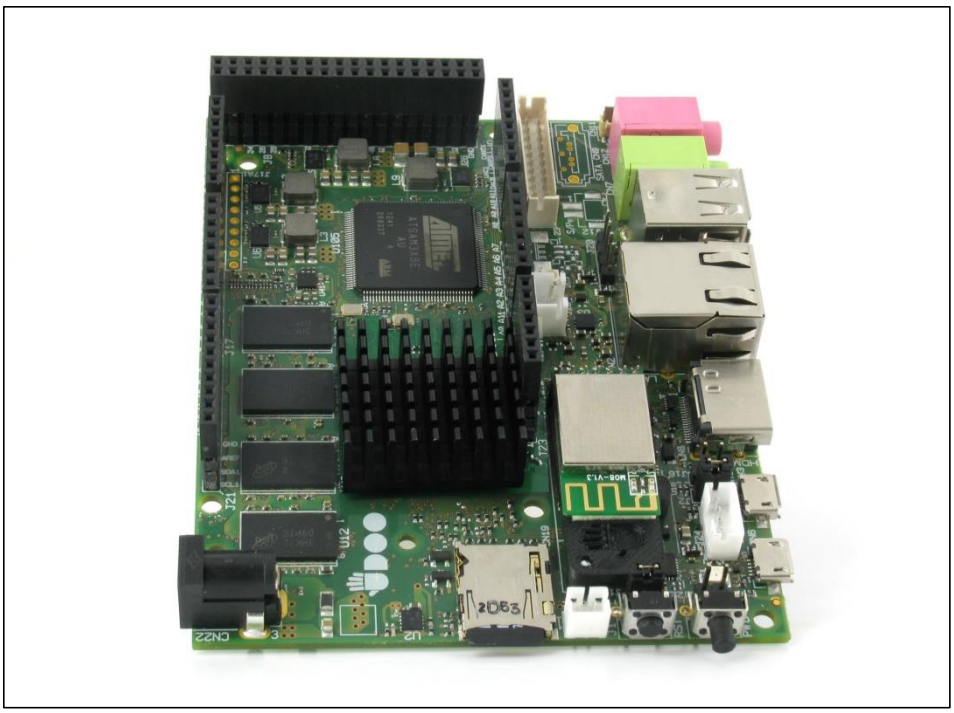

Figure 2-4. The UDOO board

Hardware

UDOO board is equipped with two CPU. The Freescale i.MX 6 is an ARM Cortex A9 processor based on ARM v7 instruction set. Next generation graphics and high-definition video are central in the i.MX 6 series. The i.MX 6 series supports up to 1080p60 video playback, enabling exceptional high-quality videos with low power consumption for devices playing high-definition content. The 3D graphics engine in the top of the line i.MX 6 Quad and i.MX 6 Dual processors are capable of providing up to 200 Mt/s, which enables ultra-vivid, realistic graphics critical for multimedia. These applications combine the power of the main cores with the until-now-untapped potential of the 3D engine to perform computational tasks.

Otherwise Atmel SAM3X implements real-time computing and allows full compatibility with Arduino Due prototyping board. With this solution UDOO can be used like an Arduino Due microcontroller and programmed in the same way.

They are both stand-alone with different clocks and no shared memory. They both process their tasks asynchronously without any kind of emulation.

They do however share some communication channels:

- a UART Serial (used with Linux)
- a USB OTG Bus (used with Android)
- all the digital pins of the external Arduino pinout.

UDOO is based on a Dual or Quad core ARM cortex-A9 CPU delivering great performance on both Android and Linux OS, and a dedicated ARM processor for the GPIO.

The hardware specifications are:

- Freescale i.MX 6 ARM Cortex-A9 CPU Dual/Quad core at 1 GHz
- Integrated GPU: each processor provides three separate accelerators for 2D, OpenGL ES2.0 3D and OpenVG
- Atmel SAM3X8E ARM Cortex-M3 CPU (same as Arduino Due)

- RAM DDR3 1GB
- 76 fully available GPIO
- Arduino-compatible R3 1.0 pinout
- HDMI and LVDS + Touch (7" and 15")
- Ethernet RJ45 (10/100/1000 MBit)
- WiFi module
- Micro USB and Micro USB OTG
- USB type A (x2) plus a supplementary USB key and a supplementary USB connector (requires a specific wire)
- I/O analog audio
- SATA (Quad-Core version only)
- CSI Camera connection
- Micro SD (boot device)
- 6V to 18v power supply and external battery connector

The following accessories are available:

- MIPI 5 MP Camera module
- MIPI 5 MP IR Camera module
- Kit LCD 7" Touch
- Kit LCD 15.6" Touch
- Kit LCD 15.6"

Software

As a complete ARM computer, UDOO can be equipped with all ARM compilable software.

The UDOO board comes with 2 micro SD card (optional) with inside Android and Ubuntu Linux.

2.2.5 Intel Galileo Gen 2

Intel Galileo is the first in a line of Arduino-certified development boards based on Intel x86 architecture and is designed for the maker and education communities.

Intel Galileo combines Intel technology with support for Arduino ready-made hardware expansion cards (called "shields") and the Arduino software development environment and libraries. The development board runs an open source Linux operating system with the Arduino software libraries, enabling re-use of existing software, called "sketches". Intel Galileo can be programmed through OS X, Microsoft Windows and Linux host operating software. The board is also designed to be hardware and software compatible with the Arduino shield ecosystem.

Intel Galileo features the Intel Quark SoC X1000, the first product from the Intel Quark technology family of low-power, small-core products. Intel Quark represents Intel's attempt to compete within markets such as the Internet of Things and wearable computing. Designed in Ireland, the Quark SoC X1000 is a 32-bit, single core, single-thread, Pentium (P54C/i586) instruction set architecture (ISA)-compatible CPU, operating at speeds up to 400 MHz.

In addition to supporting the Arduino shield ecosystem, the Intel development board comes with several computing industry standard I/O interfaces, including ACPI, PCI Express, 10/100 Mbit Ethernet, SD, USB 2.0 device and EHCI/OHCI USB host ports, high-speed UART, RS-232 serial port, programmable 8 MB NOR flash, and a JTAG port for easy debug. Intel Galileo supports the Arduino IDE running a top Linux software stack, supported by a common open source tool chain.

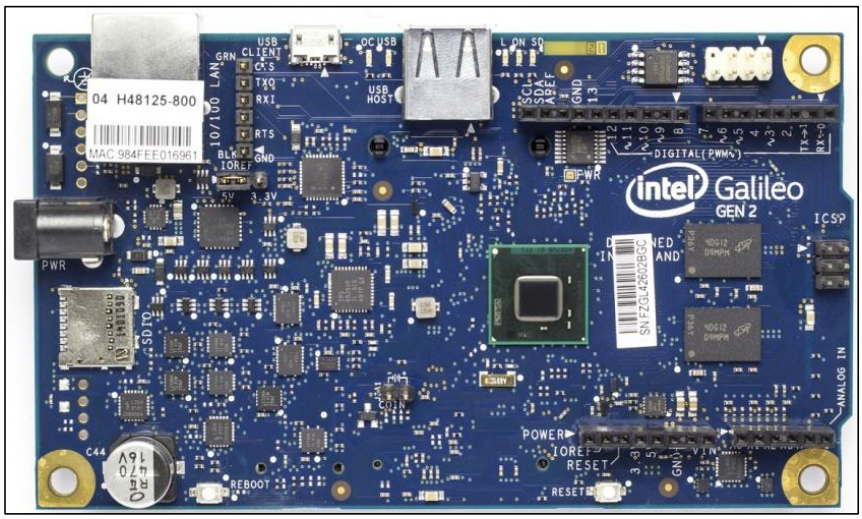

Figure 2-5. The Intel Galileo Gen 2

Hardware and Software

Hardware and software of the Intel Galileo board are the main topics of this document. They will be described in details in the next chapter.

2.3 Boards Comparison

After the above detailed technical description, we are going to compare the five boards to state our final view of the IoT platforms panorama. To make this comparison, our evaluation belongs the following criterias:

- Market data;
- CPU, GPU and Memory;
- I/O Interfaces and Ports;
- Audio/Video Interfaces;
- Operating System;
- Physical and Electrical features.

As representative of the Arduino platforms we choose the Arduino Uno board. The other Arduinos have similar technical specifications.

Market data

Name	Model	Release date	Price range [key]
Arduino Uno	R3	2010/09	1
BeagleBoard	D	2012/10	4
BeagleBoard-xM	C2	2010/09	4
Intel Galileo Gen 2	N/A	2013/10	2
Raspberry Pi	Model A, B, A+, B+	2012/02 (A)	1
		2012/10 (B)	
		2014/07 (B+)	
		2014/11 (A+)	
UDOO	Dual Basic	2013/10	4
	Dual, Quad	2013/10	5
Notes			
Price range : approximate price as of August 2014, grouped in these intervals: 1: Lowest cost, corresponding to 0-49 USD; 2: Low cost, corresponding to 50-79 USD; 3: Medium cost, corresponding to 80-119 USD; 4: High cost, corresponding to 120-169 USD; 5: Highest cost, corresponding to 170 USD and higher.			

As shown in the above comparison table, the Arduino Uno is the cheapest board. It is what we expected: the most simple hardware is also the cheapest. A different consideration must be done for the Raspberry Pi. As the Arduino, it is a low price board, even though it is a good performance mini PC. As we discuss in the next section, the Raspberry Pi is the best choice if we need a mini PC board and not a device microcontroller.

The Intel Galileo, with its entry level mini PC hardware, is the second in ascending price order. This suggests something we will discuss later: the Intel Galileo is more an Arduino microcontroller than a mini PC.

The BeagleBoard and the UDOO are the most expensive platforms. If the BeagleBoard is a middle level hardware, the UDOO is a real high performance mini PC.

CPU, GPU and Memory

Name	SoC	CPU		
		Architecture	Cores	Frequency
Arduino Uno	Atmel ATmega328P	AVR	1	16 MHz
BeagleBoard	TI OMAP3530	ARM Cortex-A8	1	720 MHz
BeagleBoard-xM	TI Sitara AM37x	ARM Cortex-A8	1	1 GHz
Intel Galileo Gen 2	Intel Quark SoC X1000	x86 Quark	1	400 MHz
Raspberry Pi Model A / B rev 1	Broadcom BCM2835	ARM11	1	700 MHz
Raspberry Pi Model B rev 2 / B+	Broadcom BCM2835	ARM11	1	700 MHz
UDOO Dual Basic	Freescale i.MX6 Dual Lite	ARM Cortex-A9	3 (2 + 1)	1 GHz
	Atmel SAM3X8E	ARM Cortex-M3		84 MHz
UDOO Dual	Freescale i.MX6 Dual Lite	ARM Cortex-A9	3 (2 + 1)	1 GHz
	Atmel SAM3X8E	ARM Cortex-M3		84 MHz
UDOO Quad	Freescale i.MX6 Quad	ARM Cortex-A9	5 (4 + 1)	1 GHz
	Atmel SAM3X8E	ARM Cortex-M3		84 MHz

Name	GPU
Arduino Uno	N/A
BeagleBoard	TMS320C64x @430 MHz, DSP
BeagleBoard-xM	C64x, DSP
Intel Galileo Gen 2	N/A
Raspberry Pi Model A / B rev 1	Broadcom VideoCore IV
Raspberry Pi Model B rev 2 / B+	Broadcom VideoCore IV
UDOO Dual Basic	Vivante GC880 + GC320
UDOO Dual	Vivante GC880 + GC320
UDOO Quad	Vivante GC2000 + GC355 + GC320

Name	RAM			
	Size	Data rate[MT/s]	Data path width [bits]	Type
Arduino Uno	2 KB	16	8	SRAM
BeagleBoard	256 MB	?	?	LPDDR
BeagleBoard-xM	512 MB	?	?	LPDDR
Intel Galileo Gen 2	256 MB	800	?	DDR3
Raspberry Pi Model A / B rev 1	256 MB	?	?	?
Raspberry Pi Model B rev 2 / B+	512 MB	?	?	?
UDOO Dual Basic	1 GB	800	32	DDR3
UDOO Dual	1 GB	800	32	DDR3
UDOO Quad	1 GB	1066	64	DDR3

The data above mark another time the difference between an Arduino microntroller (with its 16Mhz CPU frequency) and a mini PC with a quad core 1Ghz CPU and 1Gb of DDR3 Ram.

As shown by the above tables, a feature which differentiates the Intel Galileo board from the others is its x86 platform architecture, and as we see later this feature has a negative impact on the power consumption.

I/O Interfaces and Ports

Name	PCIe	USB			Storage		
		2.0	3.0	Device	On-board	Flash slots	SATA
Arduino Uno	No	No	No	No	32 KB Flash + 1 KB EEPROM	No	No
BeagleBoard	No	1	No	OTG	512 MB Flash	SD	No
BeagleBoard-xM	No	4	No	Yes	?	SD	No
Intel Galileo Gen 2	1 mini	1	No	Client	8 MB Flash + 8 KB EEPROM	SD	No
Raspberry Pi Model A	No	1	No	?	No	SD	No
Raspberry Pi Model B	No	2	No	?	No	SD	No
Raspberry Pi Model B+	No	4	No	?	No	microSD	No
UDOO Dual Basic	No	2+1	No	OTG	No	microSD	No
UDOO Dual	No	2+1	No	OTG	No	microSD	No
UDOO Quad	No	2+1	No	OTG	No	microSD	SATA

Name	Networking		Communication		
	Eth.	Wi-Fi	Bt.	I²C	SPI
Arduino Uno	No	No	No	Yes	Yes
BeagleBoard	No	No	No	?	?
BeagleBoard-xM	10/100	No	No	?	?
Intel Galileo Gen 2	10/100	No	No	Yes	Yes
Raspberry Pi Model A	No	No	No	Yes	Yes
Raspberry Pi Model B	10/100	No	No	Yes	Yes
Raspberry Pi Model B+	10/100	No	No	Yes	Yes
UDOO Dual Basic	No	No	No	Yes	Yes
UDOO Dual	GbE	n (RT5370)	No	Yes	Yes
UDOO Quad	GbE	n (RT5370)	No	Yes	Yes

Name	Generic I/O		Other interfaces
	GPIO	Analog	
Arduino Uno	22	10-bit ADC, PWM	Arduino 1.0 headers
BeagleBoard	Yes	No	
BeagleBoard-xM	?	?	?
Intel Galileo Gen 2	20	12-bit ADC, 6 PWM	Arduino 1.0 headers, JTAG, 6x UART
Raspberry Pi Model A	8	No	UART
Raspberry Pi Model B	8	No	UART
Raspberry Pi Model B+	17	No	UART
UDOO Dual Basic	76	10-bit ADC, PWM	Arduino 1.0 headers
UDOO Dual	76	10-bit ADC, PWM	Arduino 1.0 headers
UDOO Quad	76	10-bit ADC, PWM	Arduino 1.0 headers

The interesting data we retrieve from the tables above is the presence or not of General Purpose Input/Output pins (GPIOs). As we can see, the Intel Galileo and the UDOO platform are natively compatible by hardware (and software) with the Arduino headers, which can be linked to digital and analog devices. The Raspeberry Pi has GPIO interfaces only for digital devices. The BeagleBoard has some extension pins which can be configured as GPIO ports for digital devices.

The picture seems clear: if we need full support for digital and analog external devices (and the Arduino is the most widespread sensor pin interface) we have to choice Arduino as a pure microcontroller, Intel Galileo or UDOO if we need an equivalent of Arduino (i.e., an Arduino-compatible platform which we can expand with the big set of the Arduino external hardware) plus a low end or high end mini PC.

Audio/Video Interfaces

Name	Mic In	Audio Out	HDMI	LVDS	Other Video Out
Arduino Uno	N/A	N/A	N/A	N/A	N/A
BeagleBoard	Yes	Yes	YesDVI compatible	No	No
BeagleBoard-xM	Yes	Yes	Yes	No	DVI-D, S-Video
Intel Galileo Gen 2	N/A	N/A	N/A	N/A	N/A
Raspberry Pi	No	Yes	YesDVI compatible	?	Composite video
UDOO	Yes	3.5 mm, HDMI, S/PDIF	Yes	LCD header	No

The Audio/Video Interface features suggest another time that the Intel Galileo is more an Arduino than a mini PC due to the absence of the Mic In, Audio and Video Out interfaces (which are necessary for a PC configuration).

But, the Operating System specifications below highlight an important difference between Arduino and the Intel Galileo: the Intel platform is a Linux system.

Operating System

Name	Linux	Android	BSD	Windows	Other
Arduino Uno	No	No	No	No	
BeagleBoard	Yes	?	?	?	
BeagleBoard-xM	Yes	Yes	?	?	
Intel Galileo Gen 2	Yes	No	?	Yes	
Raspberry Pi	Yes	?	FreeBSD, NetBSD	?	RISC OS, Plan 9 OpenELEC Raspbian
UDOO	Yes	Yes	No	No	

Physical and Electrical features

Name	Size [mm]	Weight
Arduino Uno	75 × 53	?
BeagleBoard	78.74 × 76.2	?
BeagleBoard-xM	82.5 × 82.5	?
Intel Galileo Gen 2	123.8 × 72	?
Raspberry Pi Model A+	85.6 × 54.0 × 19.5	45 g
Raspberry Pi Model A	85.6 × 54.0 × 19.5	45 g
Raspberry Pi Model B	85.6 × 54.0 × 19.5	45 g
Raspberry Pi Model B+	85.6 × 54.0 × 19.5	45 g
UDOO	110 x 85	?

Name	Input voltage	Idle Power consumption	Max Power consumption	Power source
Arduino Uno	7–12 V	0.172 W	0.233 W	?
BeagleBoard	2.7–4.5 V	?	2 W	miniUSB or DC jack
BeagleBoard-xM	5 V	?	2 W	DC jack
Intel Galileo Gen 2	7-15 V	?	15 W	DC jack or PoE
Raspberry Pi Model A+	5 V	0.5 W	1.15 W (5 W supply)	Micro USB or GPIO header
Raspberry Pi Model A	5 V	0.7 W	5 W supply	Micro USB or GPIO header
Raspberry Pi Model B	5 V	1.8 W	2.4 W (5 W supply)	Micro USB or GPIO header
Raspberry Pi Model B+	5 V	1.0 W	5 W supply	Micro USB or GPIO header
UDOO	6-18 V	?	3-4 W	DC jack, or GPIO header or pin header

From the results above it is interesting to see the wide range in power usage. At the high end the Intel Galileo board consumes about 15W of energy when stressed! This high power consumption is likely a result of the peripherals and supporting chips on the board. The Galileo has a lot of peripheral chips like an I/O extender, analog to digital converter, Ethernet adapter, etc.. which all consume power, while the Arduino is an ATMega chip and something more. Also, we suppose that the difference between the Intel Galileo and the other mini PC platforms is all in the x86 Quark processor (responsible of more power consumption than an ARM processor board).

So, after our detailed boards comparison we could state our consideration about the current panoma of Internet-Of-Things devices.

First of all, our doubt about the IoT concept applicability to a classical desktop PC is now clear: a desktop PC is not a IoT device. All the Internet-Of-Things platforms are small, light and low power consumption electronic cards. While a classic PC is a general purpose machine, each IoT device has its particular feature which makes it different from the others. For example, in the case of Arduino we have the most cost effective platform when we deal with power consumption. Otherwise, if we need a Linux platform to setup an on board remote web control application, we have to choose at least a Raspberry Pi, or a UDOO if we have not the problem of minimizing the cost and if we want a best performance platform. But, for example if we need an analog pin interface for external hardware, our choice is restricted to the Arduino-compatible boards, i.e. the Intel Galileo or the UDOO board.

What we learn from our study is that the market offers to the makers everything they need. Each platform has its pros and cons and its peculiarities. The choice a maker is called to make depends on the project he wants to realize, on his budget and so on.

A second aspect we find studying these IoT devices is that their dimensions and power supply needs, although minimal, make applications like Home Automation and Industry more natural than clothes or "on the move" installations. Only recently the Arduino company made a "nano" Arduino platform porting called LilyPad (various model are shown in figure 2.6). The project is at its beginning and the device has no on-board network interfaces, but as the LilyPad has the same pin layout of the classical Arduino for analog and digital inputs and outputs, a standard Arduino Ethernet or WiFi adapter can be installed on it.

Figure 2-6. Various Arduino LilyPad models.

Described the state of art of the Internet-Of-Things devices, we are now ready to understand where the Intel Galileo places into the big set of the IoT platforms. The next section deals with it.

2.4 Where Does The Intel Galileo Fit?

As seen above in the detailed comparison, for some platforms (such as the Arduino Uno) the classification is simple and straightforward.

Arduino is basically a microcontroller which includes all the circuitry needed to manage the expansion hardware and software programming.

The Beagleboard is actually a mini PC. The term "mini" refers more to the size of the board than to the performance, because the platform characteristics are those of a midrange desktop PC of some years ago. Some of the expansion pins can be configured as GPIO ports, even if the expansion pin are designed primarily to use as socket for the optional expansion boards previously described.

Also the Raspberry Pi is mainly a mini PC, but it provides native GPIO ports for input and output to digital devices. In this sense the card approaches the logic of the microcontroller as a platform for external hardware management, although natively it has limitations with respect to the Arduino because of the lack of analog GPIO. "natively" is specified because recently the Cooking Hacks community (www.cooking-hacks.com) created and put on the market an expansion card which allows Raspberry to drive Arduino-compatible devices (the so called “shields”). Along with the card, Arduino APIs are also provided for development.

Raspberry is therefore a first example in which a classification between microcontroller or mini PC is not so straightforward.

The board UDOO is the full realization of this concept. The card wants to be natively both a mini PC (as seen above with high performance) and a hardware microcontroller. The will of the designers is so clear that the card uses two distinct processors, one for the PC side, the other to manage the hardware microcontroller.

The Intel Galielo board is the more difficult to classify. The card is certainly a microcontroller (it implements the hardware and software Arduino specification). It is an x86 processor platform similar to the old Pentium PC (400MHz CPU, 256Mb Ram), but it is not classifiable as a mini PC as it lacks some key features such as a video output (RCA, VGA or HDMI) and an audio output. These architectural choices suggest that the card has been designed as an evolution of the classic microcontroller (Arduino is the best known example) to approach the world of Internet of Things. The card is a Linux platform that emulates the Arduino hardware and software. We basically have a Arduino microcontroller which lives on a Linux system. But, unlike UDOO in which the Linux system can be used as a real PC, in the case of Galileo the Linux system is thought as the ground on which Arduino leans and from which Arduino can extend its functionalities. For example, an Arduino sketch can call the Linux subsystem and perform any operation, such as the launch of a C/C++ or Python application, Bash command, etc.. Through appropriate mechanisms, the Arduino sketch can read and manage the results of these calls to the Linux subsystem.

The Linux subsystem gives to the board, in addition to the just mentioned development tools, the possibility of a full connection to the Internet. For example, the maker can handle the remote control in a more effective way of an Arduino board. As we will see later, on the Linux distribution which runs on Galileo, tools such as Apache2 and PHP5 can be compiled and installed.

In this sense, the board wants to be the full realization of the Internet Of Things: a device designed for driving of hardware extensions (sensors and actuators of all kinds), but with the ability to connect to the Internet without limits. All this features on a single board.

Raspberry and UDOO could do the same things as Galileo, but in these cases we are faced with "general purpose" platforms that are able to perform both as an Arduino microcontroller and as a good performance PC. But, as seen, this obviously leads to higher costs.

While UDOO is an expensive board, Raspberry Pi borns as a cheap mini PC and microcontroller for digital devices. But if you want to use it as an Arduino microcontroller through the Cooking Hacks expansion module the cost rises significantly (this module costs 40 Euro). Else, if the GPIO analog ports and the Arduino shields are not necessary, then Raspberry Pi is a platform which offers more than Galileo at a lower cost; the developer loses only the ability to drive the Arduino-compatible devices. On the other side, if

the maker needs an Arduino-compatible board with some of the feature of a mini PC system (like a general purpose operating system), in this case the Intel Galileo board is the perfect platform.

Our analysis makes the picture more clear. As we detail in the following of this document, the Intel Galileo platform is the Internet-of-Things evolution of the Arduino system. The results of our analysis are shown in the next picture. The experimentation made on the board will confirm this results.

As we said in Chapter 1 at the beginning of this paper, in this document we focus on the Intel Galileo Gen 2 platform for the Internet-Of-Things. At this point, we have a more clear idea of what a IoT platform is and what kind of Iot platform is the Intel Galileo board with respect to other devices like Raspberry Pi or UDOO. We proceed in the next chapter with a detailed description and analysis of the Intel Galileo board, both hardware and software.

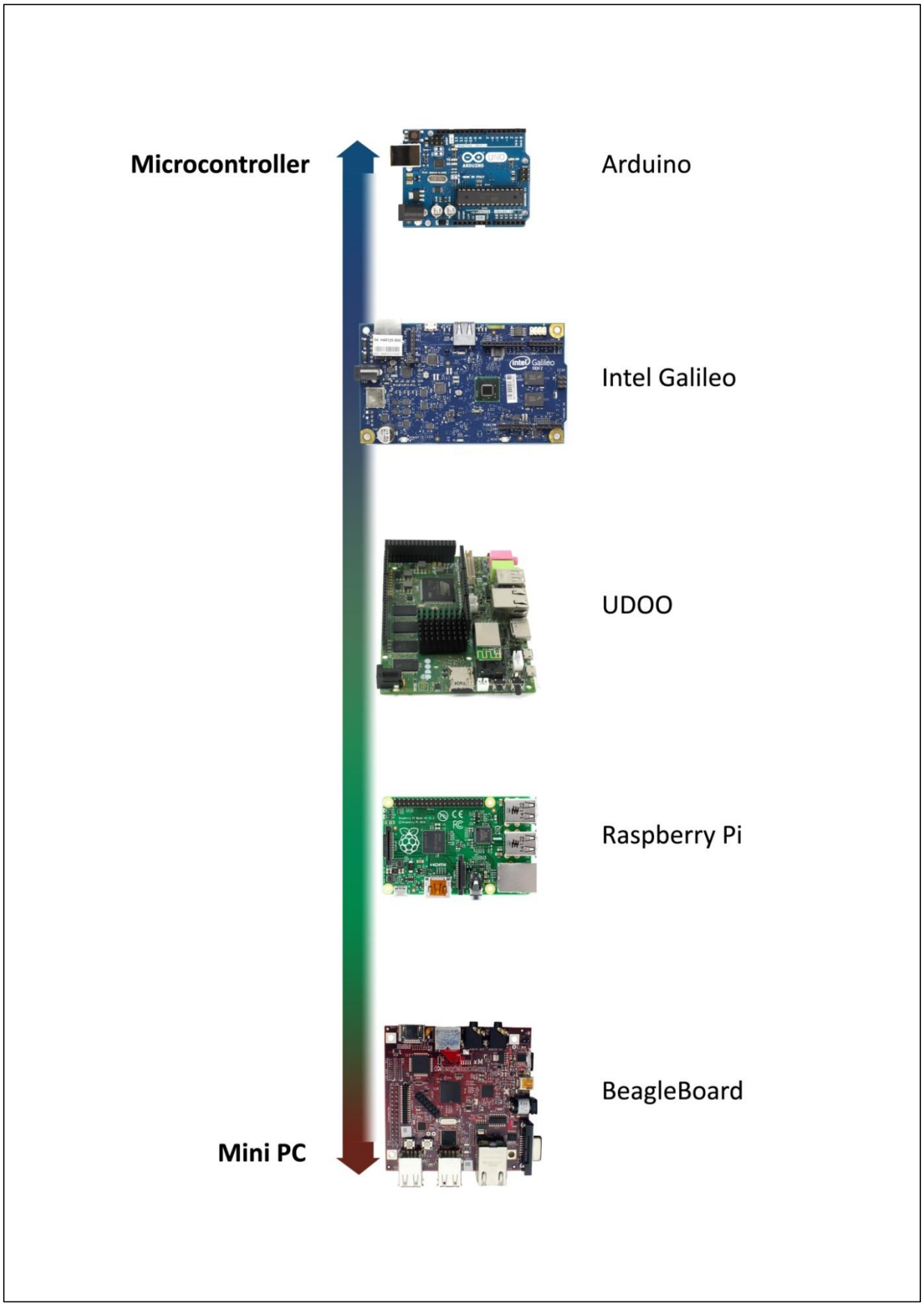

Figure 2-6. Board comparison

3 The Intel Galileo

3.1 What is Galileo?

Galileo is a hardware development board that helps maker to develop interactive objects by reading information from the physical world, processing it, and then taking action in the physical world. If it is connected to a network, it can also communicate to other devices on the Internet.

The Intel Galileo is the first Internet-Of-Things platform by Intel and the unique x86 embedded system of this type.

Galileo is hardware and software Arduino-compatible. Hardware compatible means that it is compatible with the Arduino 1.0 pinout, the design specification which defines the pin layout on the Arduino boards. Thanks to this, it is possible to attach the most part of the Arduino shields to Galileo. A shield sits on top of the board and expands its functionalities. Common circuits to drive motors, control many LEDs, or play sounds come in the form of shields. Arduino sensors and actuators can be used out of the box on Galileo. The pin layout compatibility also makes it easy to use Galileo when a maker directly deals with projects designed for other Arduinos. Software compatible means that the board can be programmed with a Intel version of the Arduino IDE using the Arduino programming language (born from the Processing project) and library (Wiring).

But Galileo is much more. Differently from other Arduino boards, Galileo is based on the Intel Quark SoC X1000 application processor, a 32-bit x86 400Mhz System on a Chip (SoC) which runs a complete Linux system.

Galileo comes out into two versions: the Gen 1 Intel Galileo and the Gen 2 Intel Galileo. In this document we focus on the Intel Galileo Gen2 board. The Gen 1 and Gen 2 boards are pretty similar. The Gen 2 board presents some hardware improvements which are highlighted in the following. Thus, the most part of this document is also Gen 1 compliant.

Let us find out how Galileo is built by the hardware and software points of view.

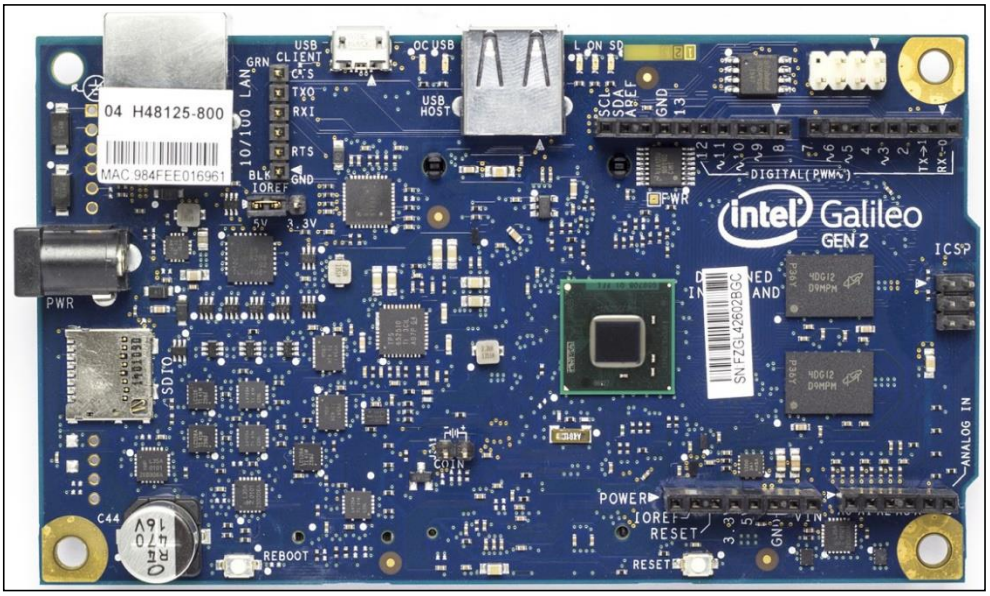

Figure 3-1. Intel Galileo Gen2 Board

3.2 INTEL GALILEO GEN2 HARDWARE

The Galileo hardware can be thought as composed by two set of hardware components which meet thanks to hardware and software links.

This two sets are the Arduino hardware and the “mini PC” hardware. The Arduino hardware is the part of the Arduino 1.0 pin layout, analog/digital converters for sensors input, digital/analog converters for actuators output and all the circuits necessary to manage the communication between the board and the external hardware. The “mini PC” hardware composed of an x86 CPU, RAM, Flash memory, mini SD card reader, Ethernet adapter and so on.

Intel lists the following specifications for Galileo Gen 2 in the product brief:

1. Processor — Intel Quark X1000 SoC @ 400MHz:
   - 32-bit Pentium-compatible ISA
   - 1.9 to 2.2W TDP (depending on operating voltage)
   - 32-bit Intel Pentium-compatible ISA
   - Supports ACPI sleep states

2. Memory:
   - 512KB embedded SRAM (in Quark SoC)
   - 256MB DDR3 DRAM

3. Storage:
   - 8MB legacy SPI NOR flash (for firmware/bootloader)
   - 8KB EEPROM (programmable via utilities)
   - Micro SD slot — supports up to 32GB
   - Supports USB 2.0 storage devices

4. 10/100 Ethernet (RJ45; supports Power-over-Ethernet)

5. USB:
   - USB 2.0 Host port (Type A)
   - USB 2.0 Client port (micro-USB, Type B)

6. 10-pin JTAG port

7. Other I/O:
   - 6-pin console UART (compatible with FTDI USB converters)
   - 6-pin ICSP

8. Arduino-compatible expansion headers, containing:
   - 20x GPIOs (12 fully native speed)
   - 6x analog inputs
   - 6x PWMs with 12-bit resolution
   - 1x SPI master
   - 2x UARTs (one shared with console UART)

- 1x I2C master

9. Mini-PCIe expansion — 1x slot (with USB 2.0 Host support)

10. RTC — onboard battery option

11. Buttons:
    - Reset button for resetting sketch and attached shields (resets Ethernet)
    - Reboot button for processor restart

12. Power:
    - 7-15VDC input jack (consumption not currently specified)
    - Supports Power-over-Ethernet (requires PoE module)
    - Optional 3V coin cell battery for standby power

13. Dimensions — 123.8 x 72.0mm (not including real-world port extensions beyond the board outline)

As the Intel Galileo emulates by hardware an Arduino microcontroller, the Arduino-compatible external devices (the "shields") are also Galileo-compatible. As demonstrated by a detailed experimentation described in the official Intel report on Arduino shields compatibility, almost all the Arduino add-on shields are fully compatible by hardware and fully supported by software on the Intel Galileo platform.

3.3 Hardware differences Between Gen1 and Gen2 boards

Intel implemented the following changes to the Galileo Gen2 with respect to the original Galileo Gen1 specifications:

1. 12 GPIOs are now "fully native," for faster speed and greater signal drive;
2. PWM now offers 12-bit resolution, for more precise servo control;
3. The USB Host port is now a Type A connector instead of a micro-USB connector;
4. A 6-pin TTL UART header (compatible with FTDI USB converters) replaces the earlier board's 3.5mm RS-232 console debug port; the new connector mates with standard adapters;
5. Console UART1 can be redirected to Arduino headers in sketches, which can eliminate the need for "soft-serial";
6. Now operates from either 7-15VDC input power or via optional 12V PoE, which has been added to the 10/100 Ethernet port (requires optional PoE module).

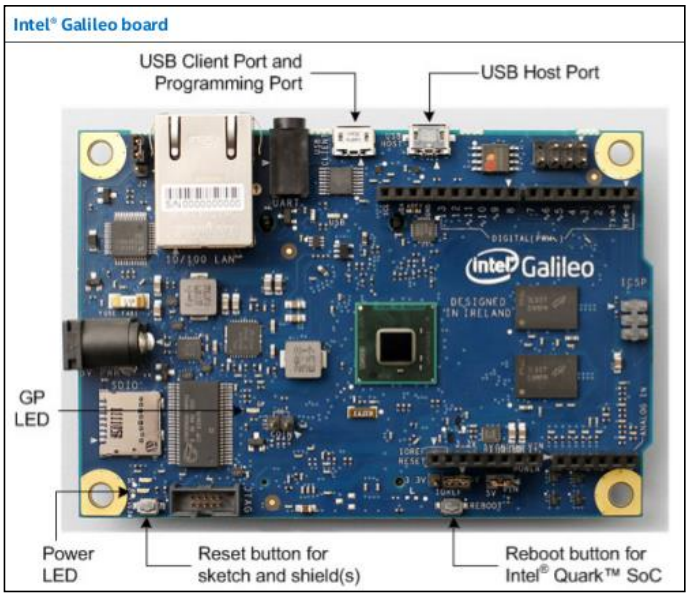

Figure 3-2. Intel Galileo Gen1 Board

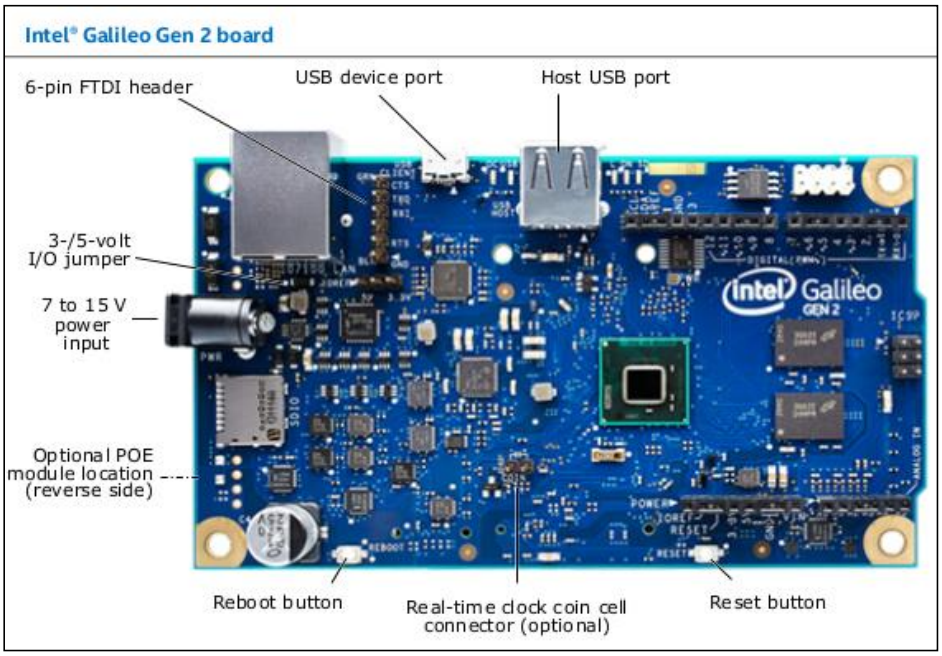

Figure 3-3. Intel Galileo Gen2 Board

3.4 INTEL GALILEO SOFTWARE

From the software point of view, the Intel Galileo is an embedded device running Linux.

Into his 8MB NOR flash the board hosts a Yocto 1.4 "Poky" Linux release. At startup, the firmware boots this tiny Linux distribution.

The environment lets the user to program the board through the standard Arduino IDE, Arduino programming language and API (Wiring). The developer can use all the Arduino software features, except the APIs which use directly the Arduino ATmega328 micro-controller assembly language.

The Arduino software (the so called "sketches") written with the Arduino PL is compiled by the Arduino IDE into a Linux runnable binary "sketch.elf" and transfered on the Linux FileSystem under the *"/sketch/"* folder. Then, the so called Arduino Sketch Interpreter (a Linux Service written in the C programming language) runs the sketch.

Thus, in the Galileo environment, the Arduino APIs are stubs (the so called "Arduino I/O Adapter") to input/output services implemented on the Linux OS as system libraries (the actual "I/O Driver").

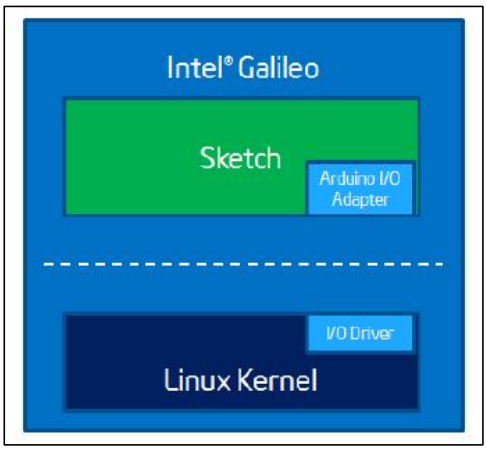

Figure 3-4. Intel Galileo software stack

This software architectural design of the Intel Galileo board allows the experienced Arduino makers to enjoy both the simplicity of the Arduino hardware and development environment, and the power of a PC-like hardware and software.

From the Arduino code the user could access the underlying Linux command-line through the *system()* function. The function gets in input a string which represents the Bash Linux command to be executed. Details about this will be shown in "Chapter 6 – Basic Experiments".

But, the Galileo board gives something more with respect to the Arduino when programmed as a Linux system. The 8MB flash embedded Linux provides only the core functions to run Galileo as a Arduino board. If a full Linux distribution is booted from an external miniSD card, the maker has a complete embedded Linux environment to work with.

This dual face of the Intel Galileo board (the Arduino side and the Linux side) gives to the developer the possibility for different approaches when programming the board.

The first consideration is that, as a Arduino sketch is translated by the Arduino IDE from a Arduino PL source file to a C/C++ source file and then to a Linux runnable binary, in theory software (the Arduino sketch) can be produced directly as Linux binary through the C/C++ programming language. As will be clear in the description of our experimentation, this is true only in theory, because when a maker programs the board through the Arduino IDE he has the full Arduino API available, but an official C/C++ Arduino API is was not released so far. This is why for a classical Arduino development the main tool to work with remains the Arduino IDE.

Another interesting aspect of the board is that, to emulate the Arduino software, the firmware links the hardware Arduino General Purpose Input Output pins (GPIOs) to the Linux FileSystem through the stream files under the *"/sys/class/gpio/gpio*"* folders (for digital input/output and analog output) and under the *"/sys/bus/iio/devices/iio\:device0/"* folder (for analog input). It means that, under the hood, the final sketch produced by the Arduino IDE is a Linux binary which realizes the I/O to the external devices through raw readings and writings on the Linux FileSystem.

This feature allows a maker to read sensor input and write actuator output through any programming language (i.e., Python, Bash, Java, C, C++, and so on) just reading and writing files on the Linux FileSystem. As we describe in the following, also this is true only in theory. If the external device to drive is a simple LED or any simple "one pin" controlled sensor we can actually manage the hardware though the Linux FileSystem. But, as the hardware complexity increases this is no longer easy to accomplish. This is the main reason why the absence of an official C/C++ Arduino API does not permit to the developer to program the Arduino side of the board directly as C/C++ code. The same is true for all programming language: we do not have any Python/Java/Bash/ecc.. Arduino API. As we see in the following of this document, we have only an example of a basic C/C++ Arduino API directly by Intel, but only for demonstrating purpose. In the same way, but this time by an independent developer, a basic Python library is available.

It is clear from the considerations above that it is not immediate what is the correct way to exploit the full power of the Intel Galileo embedded system. What we try in this document is to describe a methodology to work with it.

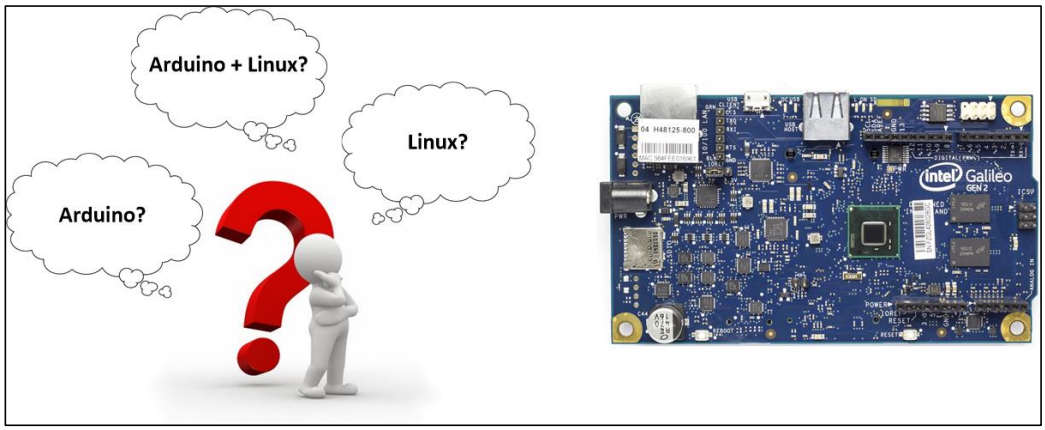

Figure 3-5. How is the right way to develop on the Intel Galileo board?

In "Chapter 4 – The Arduino side" we setup a working environment to develop on the Intel Galileo as Arduino system. In the following "Chapter 5 – The Linux side" is described the setup of the software (both on the Intel Galileo and on our working PC) necessary to exploit the Linux side of the board. "Chapter 6 – Basic Experiments" and "Chapter 7 – Advanced Experiments" are a collection of our tests on the board. Our experimentation is finalized to find the right way to deal with our Intel Galileo board.

4 The Arduino Side

4.1 What Is Arduino?

Arduino is a hardware and software platform for making devices that can sense and control the external physical world through sensor and actuators. It is an open-source physical computing platform based on a simple microcontroller board, and a development environment for writing, with simplicity, software which controls the board.

Arduino can be used to develop interactive objects, taking inputs from a variety of switches or sensors, and controlling a variety of lights, motors, and other physical outputs. As open source platform, Arduino boards can be purchased preassembled or assembled by hand (e.g., on a breadboard) according to the official specification; also the Arduino IDE is open source and can be downloaded for free.

The Arduino programming language is based on the Processing programming environment and the Arduino API is an implementation of the Wiring library.

4.2 Why Arduino?

There are many microcontroller platforms available for physical computing which hide the details of microcontroller programming and wrap the low level side of programming the platform in an easy-to-use package. Arduino is one of these, but it simplifies even more the process of working with microcontrollers and offers some advantage for teachers, students, and interested amateurs over other systems thanks to the following features:

1. Inexpensive: Arduino boards are relatively inexpensive compared to other microcontroller platforms. The least expensive version of the Arduino module can be assembled by hand, and even the pre-assembled Arduino modules cost less than $50;
2. Cross-platform: The Arduino software runs on Windows, Macintosh OSX, and Linux operating systems. Most microcontroller systems are limited to Windows;
3. Simple, clear programming environment: The Arduino programming environment is easy-to-use for beginners, yet flexible enough for advanced users to take advantage of as well. For teachers, it's conveniently based on the Processing programming environment, so students learning to program in that environment will be familiar with the look and feel of Arduino;
4. Open source and extensible software: The Arduino software is published as open source tools, available for extension by experienced programmers. The language can be expanded through C++ libraries, and people wanting to understand the technical details can make the leap from Arduino to the AVR C programming language on which it's based. Similarly, you can add AVR-C code directly into your Arduino programs if you want to;
5. Open source and extensible hardware: The Arduino is based on Atmel's ATMEGA8 and ATMEGA168 microcontrollers. The plans for the modules are published under a Creative Commons license, so experienced circuit designers can make their own version of the module, extending it and improving it. Even relatively inexperienced users can build the breadboard version of the module in order to understand how it works and save money.

We believe that the above are the reasons why Intel decided, to enter the world of Internet-Of-Things, to support the Arduino platform. Furthermore, as visible on the Arduino web site at http://arduino.cc/en/ArduinoCertified/Products , the Intel Galileos are Arduino certified products. As we can

imagine, this generates interest on the Intel boards into the Arduino community which is the widest of the maker communities in the world.

4.3 GALILEO GETTING STARTED

In the following we explain how to setup the Intel Galileo Arduino environment for the Windows, Linux and Mac platforms. As we saw in chapters 2 and 3, the Intel Galileo board is a full Arduino-compatible system. As such, the experienced Arduino maker could perform an easy migration of his projects from a standard Arduino board to the Intel one. So, the following setup is a prerequisite for the developer who wants to program the board in the same manner he programs the old Arduinos. Experimentations in Chapter 6 and Chapter 7 also require the following configuration.

4.4.1 GALILEO GETTING STARTED - WINDOWS

STEP 1 - CONNECT POWER FIRST, THEN USB

WARNING: Power supplies are NOT interchangeable between the Intel Galileo and Intel Galileo Gen 2 boards. The Gen 2 power supply is 12V and will permanently damage 1st generation Galileo boards.

1. With nothing else connected to your Galileo board, plug power cable into the wall and then connect it to the Galileo DC jack. You should see a couple of LEDs light up;
2. Connect Galileo to your computer using a USB A to micro B cable to USB (cable not included in package) client port on the board;

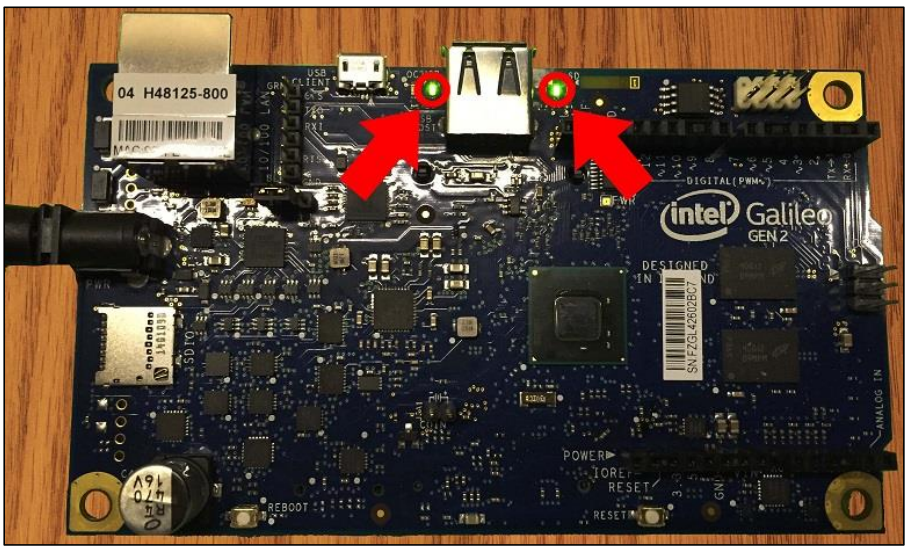

Figure 4-1. On board leds

STEP 2 - DOWNLOAD GALILEO ARDUINO IDE

1. Click the link (https://communities.intel.com/docs/DOC-22226) to download most up-to-date Intel Arduino IDE for Windows;
2. Unzip the file to your C:\ drive. You should rename the file if you have other versions of Arduino on your computer that you want to keep, The name must not contain any spaces. For example, if you have the original Arduino IDE you will want to name the new one something that indicates that it is for Galileo, e.g., "Galileo_Arduino." Be sure to unzip the file to your C:\ drive due to a known issue unzipping packages with long file paths. If you have the first generation Galileo Arduino IDE

installed, and you are upgrading to Gen 2, you must install the updated version to run your new board. You can overwrite the existing Intel Arduino folder without worrying about losing your existing sketches;

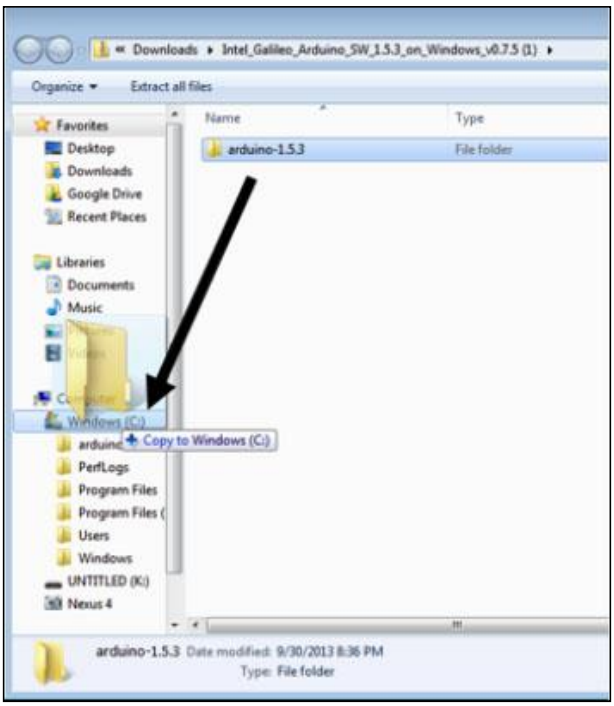

Figure 4-2. Extract the Arduino IDE somewhere on the disk

3. Launch application by double-clicking on the application inside your Arduino-1.5.3 folder;
4. For easier access to your IDE, right click on the application file and select "Pin to Start Menu."

STEP 3 - INSTALL DRIVERS

1. Select Start -> Control Panel->System and Security->Device Manager;
2. Locate the Gadget Serial v2.4 device, under the Other devices tree. Right-click that and select Update Driver Software;

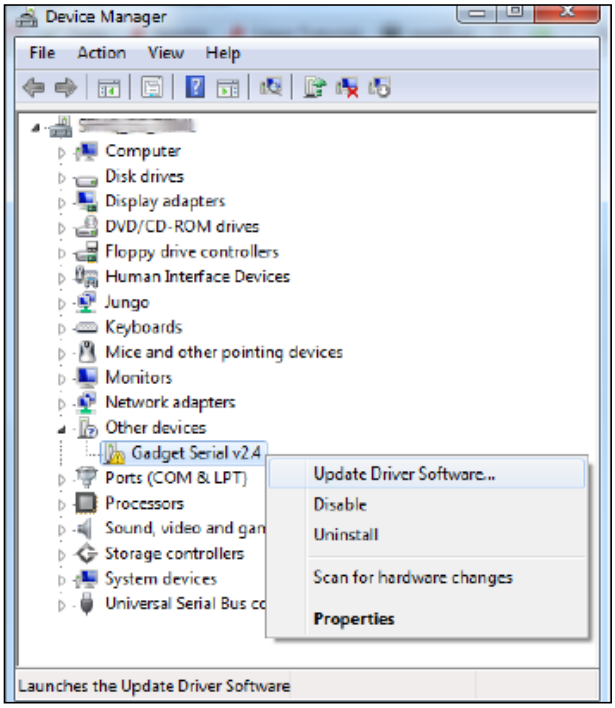

Figure 4-3. Update drivers into the Device Manager

NOTE: You might get an error regarding a problem installing drivers - you can ignore this error.

3. On the first window that pops up, click Browse my computer for driver software. And on the next page select Browse… and navigate to the Arduino-1.5.3\hardware\arduino\x86\tools folder within your Arduino Galileo software installation. Then click OK;

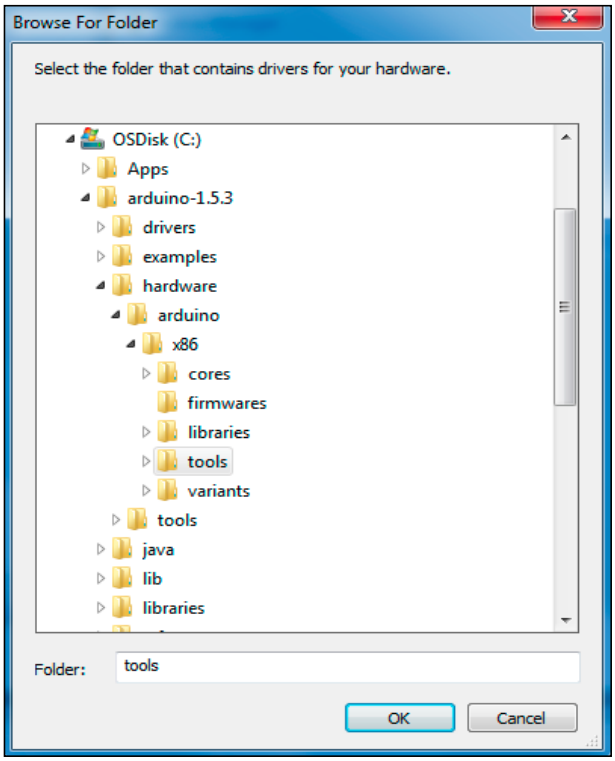

Figure 4-4. Select the drivers

4. Click Install on the next Windows Security window that pops up. And, after a number of loading-bar-scrolls, the installation should complete and you should be greeted with a Windows has successfully updated your driver software window;
5. Once the driver is successfully installed, Device Manager will show a Galileo (COMx) device under Ports (COM & LPT). Note the COMx port number as it will be needed in the IDE later.

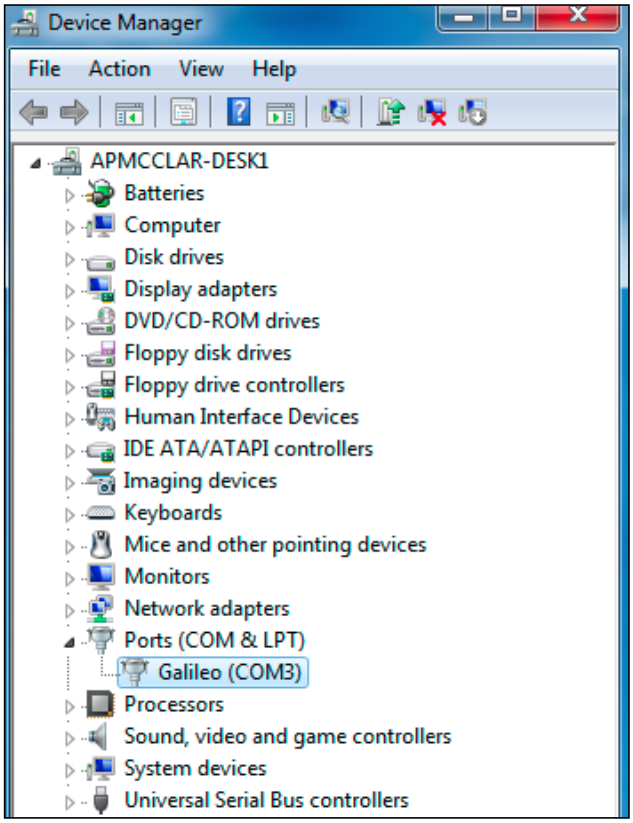

Figure 4-5. The driver is updated

STEP 4: UPDATE FIRMWARE

WARNING: During the firmware update it is extremely important that your board remain plugged in and powered on!

NOTE: If you have a microSD card inserted, you will need to power off your Galileo, remove the card, and reboot Galileo without the card in before updating your firmware.

1. In Galileo Arduino IDE, go to Tools pull-down menu at the top, select Board, and make sure “Intel Galileo” or "Intel Galileo Gen2" is selected, depending on which board you are using;

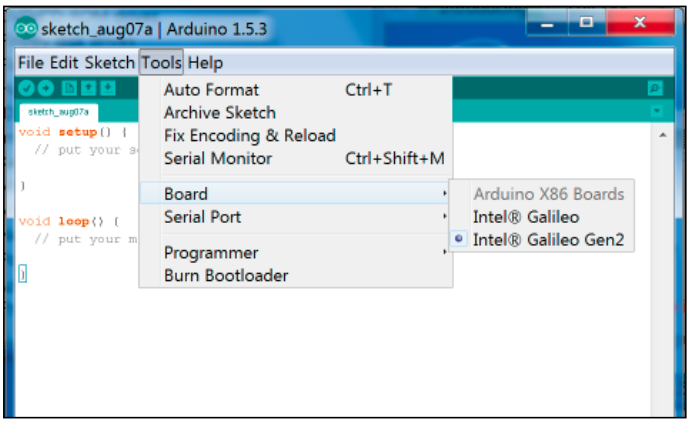

Figure 4-6. Select the Intel Galileo Gen 2 board

2. In Galileo Arduino IDE, go to Tools -> Serial Port in the menu. Select the serial port that looks like this: COMx, where "x" is the number referenced in the previous step;

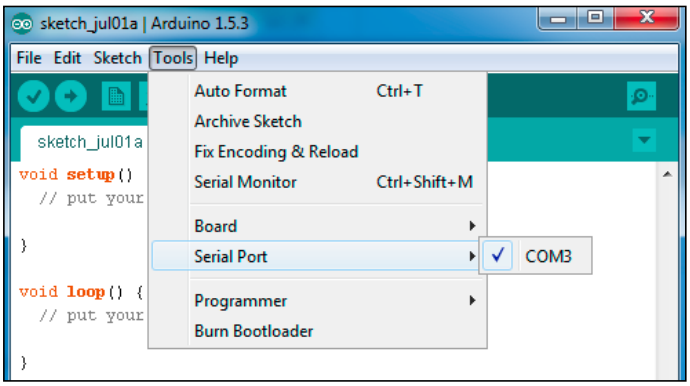

Figure 4-7. Select the right serial port

3. Go to Help -> Firmware Update check to see if your board has the latest version.If your firmware is up to date, you do not need to complete this step. If your firmware is not up to date, proceed with firmware update. When you have succeeded you will see a message indicating that your firmware has been updated. It takes about 5 minutes.

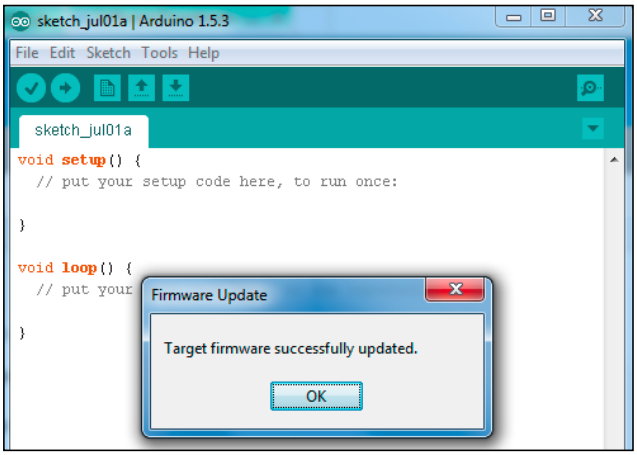

Figure 4-8. Successful message on firmware update

STEP 5: TEST THE BOARD WITH THE BLINK EXAMPLE

1. In IDE, Click: File -> Examples -> 01.Basics -> Blink;

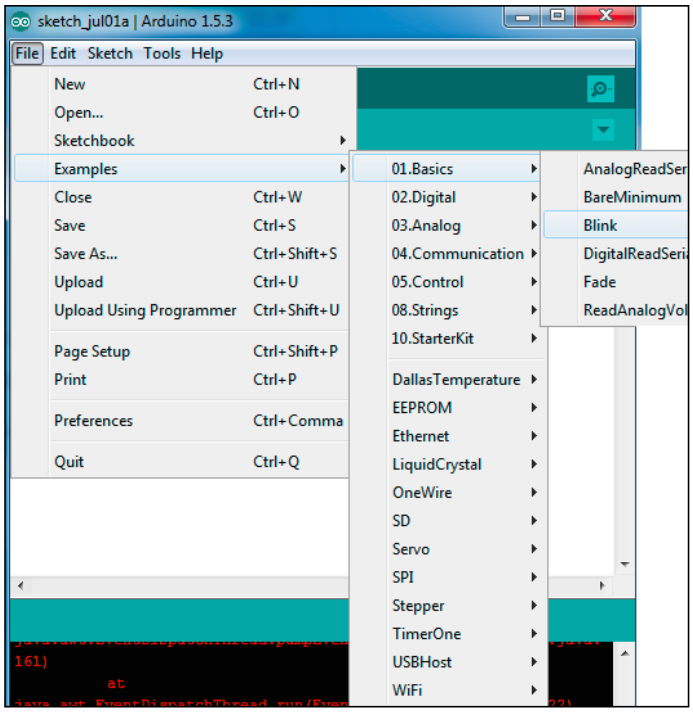

Figure 4-9. Select the Blink example

2. A new sketch window will open with some code in it;

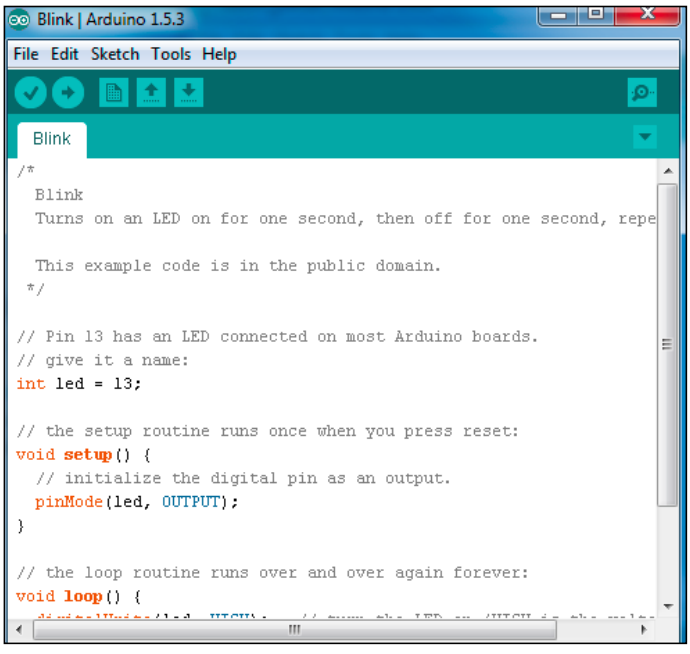

Figure 4-10. The Blink example

3. Click the Upload button in the toolbar;

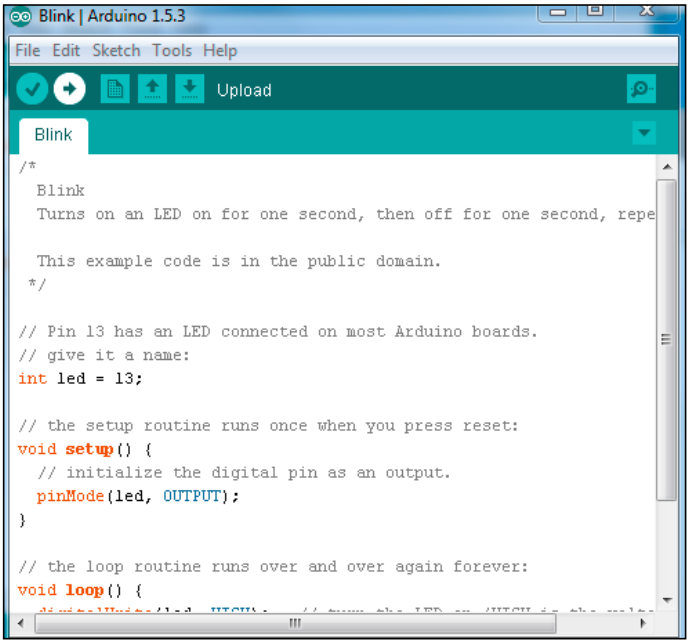

Figure 4-11. The Blink example

4. It may take a few moments for the code to compile and upload. You can see the progress at the bottom of the window;
5. When it is done, you will see the text "Transfer Complete" & “Done Uploading” at the bottom of the sketch window.

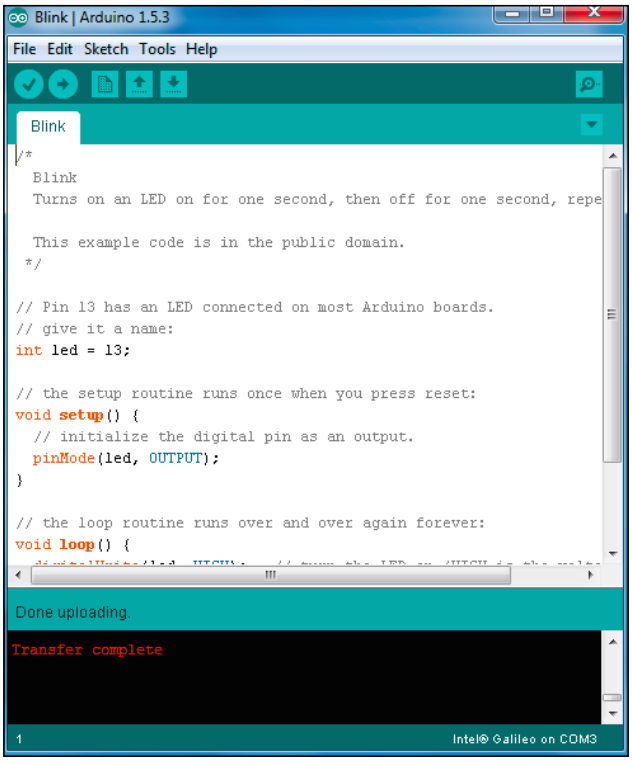

Figure 4-12. The Blink example uploaded on the board

You should see a blinking light!

4.2.2 GALILEO GETTING STARTED – LINUX

STEP 1 - CONNECT POWER FIRST, THEN USB

The steps are the same seen in section "3.2.1 Galileo Getting Started – Windows".

STEP 2: DOWNLOAD INTEL ARDUINO IDE

WARNING: If you already have a copy of the standard Arduino IDE installed, you should rename the one you downloaded from Intel and keep both in your Applications folder. The name you choose must NOT contain spaces. You may want to name it Galileo_Arduino, for example. If you have the first generation Galileo Arduino IDE installed, and you are upgrading to Gen 2, you must install the updated version to run your new board. You can overwrite the existing Intel Arduino folder without worrying about losing your existing sketches.

NOTE: These instructions have been tested on a fresh installation of Ubunto 12.04.3 LTS, but should work with most other distributions.

NOTE: A system service called modem manager can interfere with the Galileo. One option is to remove it with the following command in the terminal window:

sudo apt-get remove modemmanager

You will want to kill the modemmanager PID since it is likely running:

ps -ef|grep modem and kill [PID]*

Or, reboot after it's removed using:

sudo apt-get purge modemmanager

An alternative to removing modemmanger is to edit the ModemManager blacklist file. You can find this on Fedora at:

/usr/lib/udev/rules.d/77-mm-usb-device-blacklist.rules

and the line you need to add is:

ATTRS{idVendor}=="8086", ATTRS{idProduct}=="babe",

ENV{ID_MM_DEVICE_IGNORE}="1"

HINT: To see if your system is 32 bit (i686) or 64 bit (x86_64) use the below command. The response will include a "32" or "64" depending on what your system is.

uname –m

TERMINAL INSTALL

1. Go to the Galileo Software Downloads and click the 32-bit or 64-bit download link to download the most up-to-date Intel Arduino IDE for Linux.. You will need to close the new browser tab to return to these instructions;
2. If your browser asks you, save the .tgz file in your ~/Downloads directory;
3. Open a terminal window by clicking on its icon in your application launcher or by typing Ctrl+Alt+T;
4. Navigate to your downloaded file and extract if from there to your home directory with tar:

   cd ~/Downloads

   tar -xzf <tgz filename> -C ~/
5. Navigate to the extracted files.

cd ~/ arduino-1.5.3

6. Start the Arduino IDE with:

   ./arduino

GUI INSTALL

1. Go to the Galileo Software Downloads and click the 32-bit or 64-bit download link to download the most up-to-date Intel Arduino IDE for Linux.. You will need to close the new browser tab to return to these instructions;
2. Double click on the tgz package, select the folder where you would like to extract the IDE folder to;
3. After the package has extracted, select "Show the Files";

Figure 4-13.

4. Double click the arduino-1.5.3 folder;
5. Double click the executable "arduino" file and select "Run" from the options.

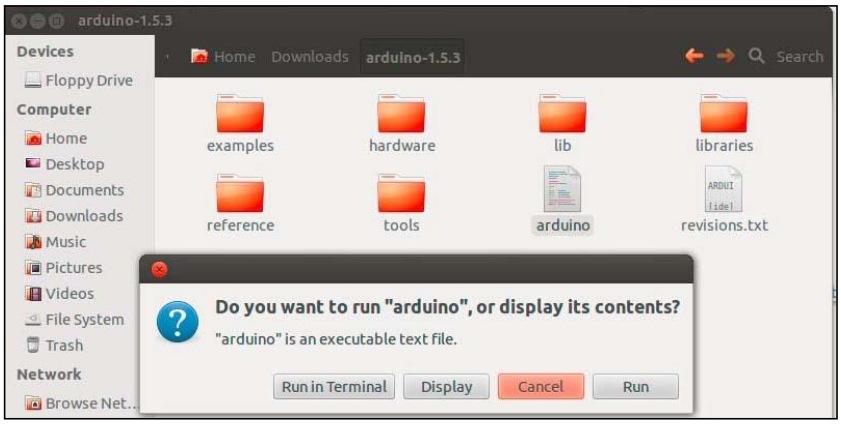

Figure 4-14.

NOTE: If you get an error about Java not being found, install it with:

sudo apt-get install default-jre

STEP 3: UPDATE FIRMWARE

WARNING: During the firmware update it is extremely important that your board remain plugged in and powered on!

NOTE: If you have a microSD card inserted, you will need to power off your Galileo, remove the card, and reboot Galileo without the card in before updating your firmware.

1. In Galileo Arduino IDE, go to Tools pull-down menu at the top, select Board, and make sure “Intel Galileo” or "Intel Galileo (Gen 2)" is selected;

Figure 4-15.

2. In Galileo Arduino IDE, go to Tools -> Serial Port in the menu. Select the serial port that looks like “*/dev/ttyACM0”*;

Figure 4-16.

3. Go to Help -> Firmware Update to update your board to the latest version. When you have succeeded you will see a message indicating that your firmware has been updated. It takes about 5 minutes.

Figure 4-17.

Step 4: Test the Board with the Blink Example

1. Close down your IDE and restart it after the firmware update has completed;
2. In IDE, Click: File -> Examples -> 01.Basics -> Blink. A new sketch window will open with some code in it;

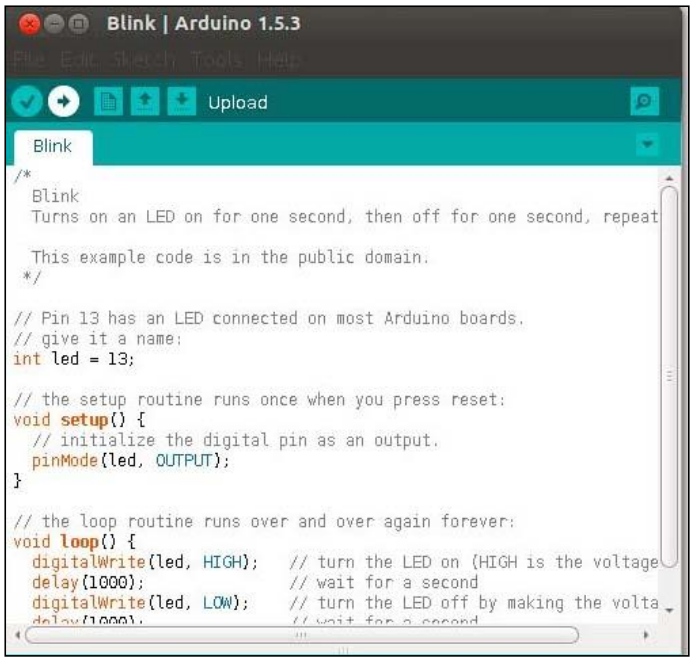

Figure 4-18.

3. Click the Upload button in the toolbar;
4. It may take a few moments for the code to compile and upload. You can see the progress at the bottom of the window;
5. When it is done, you will see the text "Transfer Complete" & “Done Uploading” at the bottom of the sketch window.

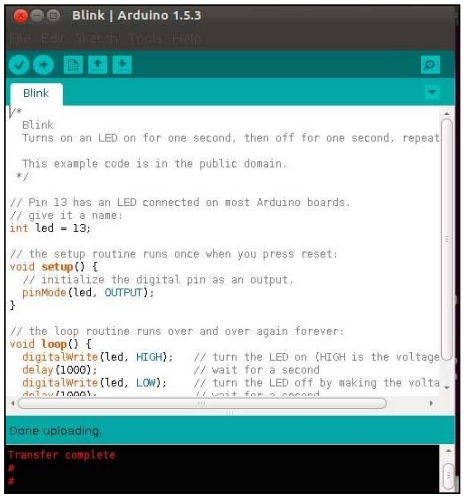

Figure 4-19.

You should see a blinking light!

4.2.3 GALILEO GETTING STARTED – MAC

STEP 1 - CONNECT POWER FIRST, THEN USB

The steps are the same seen in section “3.2.1 Galileo Getting Started – Windows”.

STEP 2: DOWNLOAD INTEL ARDUINO IDE

WARNING: If you already have a copy of the standard Arduino IDE installed, you should rename the one you downloaded from Intel and keep both in your Applications folder. The name you choose must NOT contain spaces. For example, if you have the original Arduino IDE you will want to name the new one something that indicates that it is for Galileo, e.g., "Galileo_Arduino." If you have the first generation Galileo Arduino IDE installed, and you are upgrading to Gen 2, you must install the updated version to run your new board. You can overwrite the existing Intel Arduino folder without worrying about losing your existing sketches.

1. Click the link Galileo Software Downloads to download most up-to-date Intel Arduino IDE for Mac. You will need to close the new browser tab to return to these instructions;
2. Accept the terms in the license agreement and your download will begin;
3. Install the software on your computer. Decompress file if necessary, drag Arduino icon into your /Applications folder to install it;
4. Launch application by double-clicking it form your /Applications folder.

Figure 4-20.

STEP 3: UPDATE FIRMWARE

WARNING: During the firmware update it is extremely important that your board remain plugged in and powered on!

NOTE: If you have a microSD card inserted, you will need to power off your Galileo, remove the card, and reboot Galileo without the card in before updating your firmware.

1. In Galileo Arduino IDE, go to Tools pull-down menu at the top, select Board, and make sure “Intel Galileo” or "Intel Galileo Gen 2" is selected;

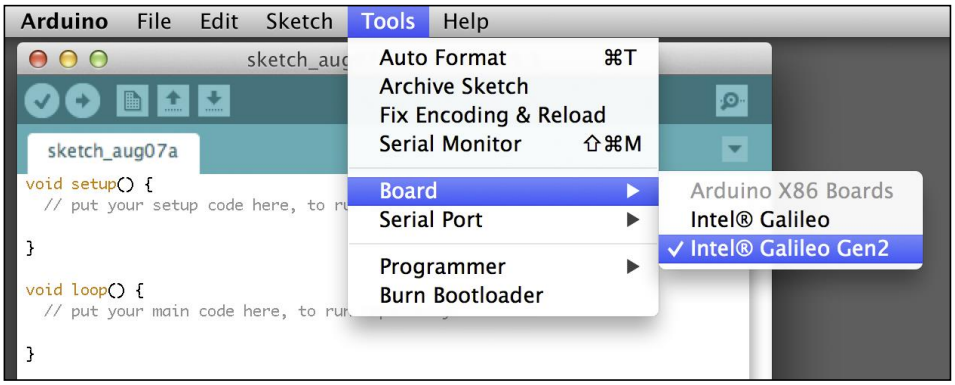

Figure 4-21.

2. In Galileo Arduino IDE, go to Tools -> Serial Port in the menu.
   Select the serial port that looks like *"/dev/cu.usbmodemXXXXX"*. XXXXX are values like fd121.

NOTE: Make sure you don't select the /dev/tty port.

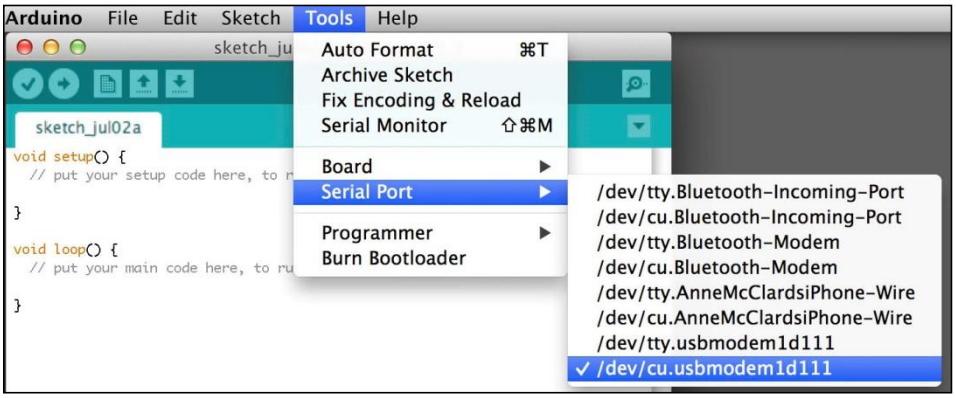

Figure 4-22.

3. Go to Help -> Firmware Update to update your board to the latest version. When you have succeeded you will see a message indicating that your firmware has been updated. It takes about 5 minutes.

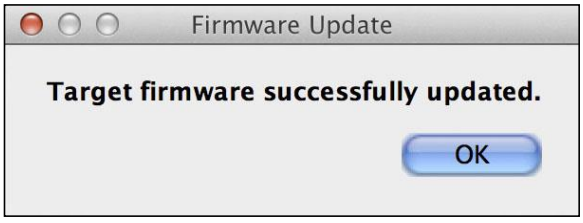

Figure 4-23.

Step 4: Test the Board with the Blink Example

1. Close down your IDE and restart it after the firmware update has completed;

2. In IDE, Click: Files -> Examples -> 01.Basics -> Blink;

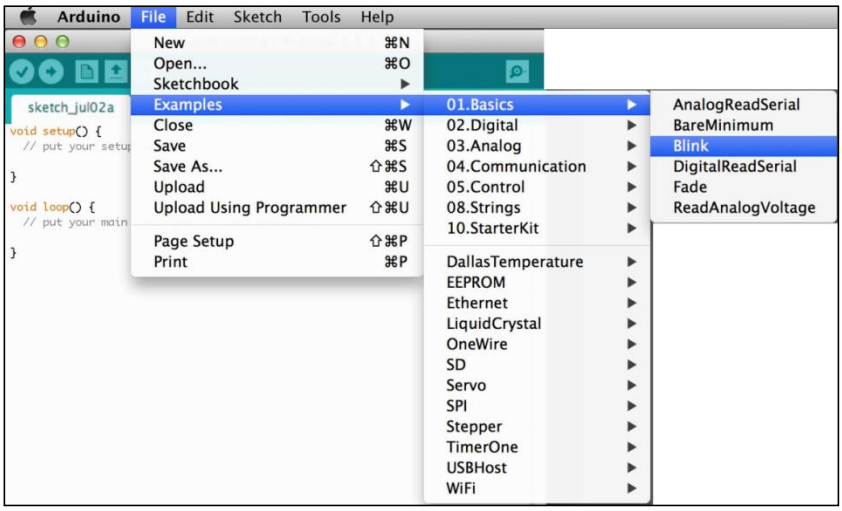

Figure 4-24.

3. A new sketch window will open with some code in it;

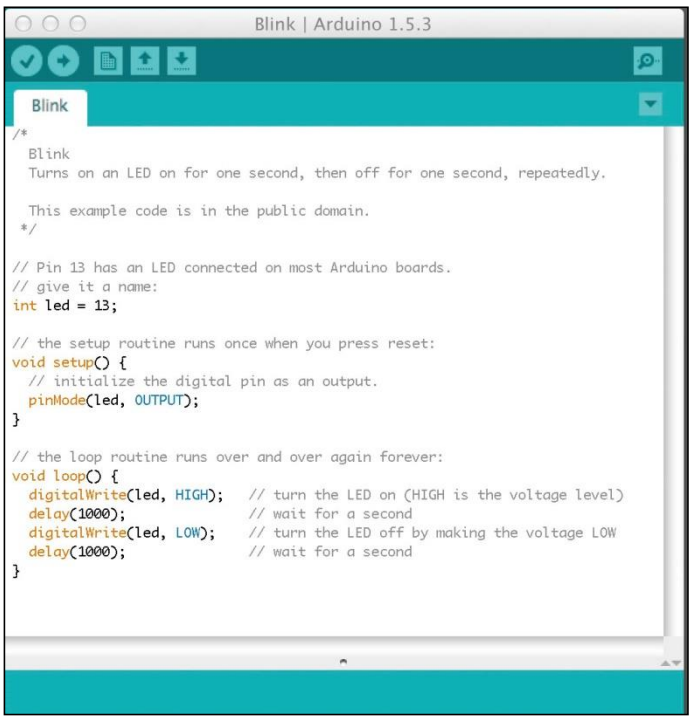

Figure 4-25.

4. Click the Upload button in the toolbar;
5. It may take a few moments for the code to compile and upload. You can see the progress at the bottom of the window;

6. When it is done, you will see the text "Transfer Complete" & “Done Uploading” at the bottom of the sketch window.

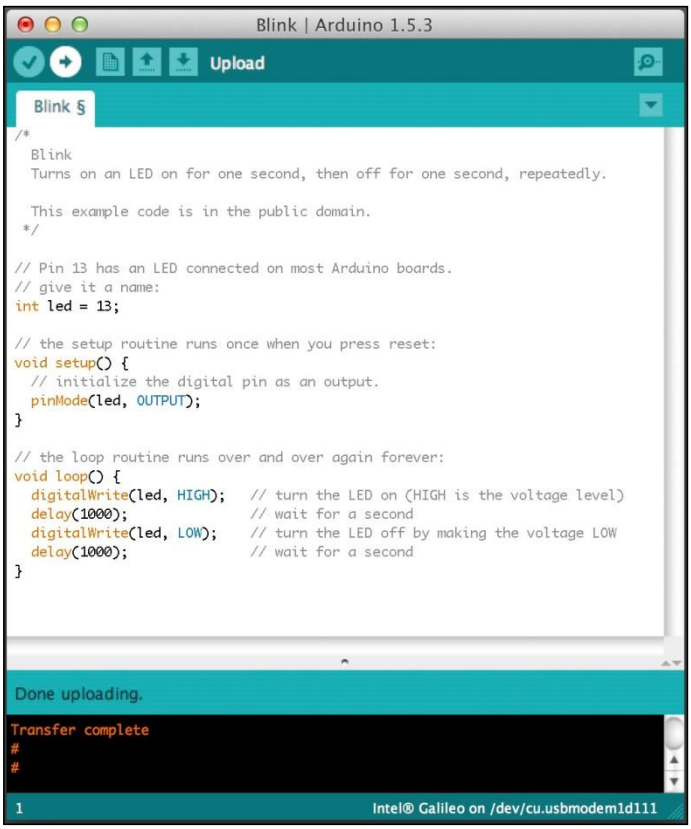

Figure 4-26.

You should see a blinking light!

At this point we have a working environment to program our Intel Galileo board as classical Arduino system. If a maker wants to employ the Intel Galileo board in place of an old Arduino (for example only for performance purpose, to get benefit of the 400Mhz processor) he could stop this lecture and go to develop on the board in the same manner he did before on the classical Arduino system. Thus, he has the same development IDE, the same programming language, the same APIs, the same shields to expand the board. Moreover, he has more CPU power, more RAM and a on-board Ethernet adapter to connect the Internet.

But, there is much more. The next chapter describes the setup of the Linux side development environment. The Linux environment setup is introductory to our experimentation described in Chapter 6 and Chapter 7. The Linux side of the board is what differences our Intel Galileo from the basic Arduino system from the software point of view.

5 The Linux Side

5.1 An introduction to Yocto Project for Intel Galileo

In the previous Chapter 4 we described how to setup the environment to exploit our Intel Galileo as full Arduino system. As we said, the experienced Arduino maker could stop the reading and start programming the board in the same manner he did with his classical Arduino board.

In the present chapter we begin to explore the other side of the coin, the Linux side. To access the Linux functionalities of our Intel Galileo we need more configurations, both on the board and on our working PC. The setup we describe here is introductory to our experimentation described in Chapter 6 and Chapter 7, where we shows projects that exploit both the Arduino and the Linux side of the Intel Galileo board.

As we learned in Chapter 3 when describing the Intel Galileo software stack, the Intel Galileo is a Linux embedded system, and the Arduino side in emulated by software. So, as we could access the Arduino side of the board forgetting that a Linux system is running under the hood, in the same way we could forget that we are dealing with an Arduino-compatible board and approach the board as a standard Linux system. And it is exactly what we do in the present chapter: we setup a complete Linux system on our Intel Galileo machine and configure our working PC to access to it. In fact, as we see in the following, we access the on-board Linux through standard instruments like a SSH connection for login, SFTP for file transfer and so on.

Probably the observant reader may have understood that our final development methodology on the Intel Galileo is an hybrid approach in which the Arduino side and the Linux side of the platform cooperate synchronously. This is the matter of Chapter 7, where we show how, step by step, our experimentation leads the developer to work on both side, confiding some part of the job to the Arduino way and the other to the Linux environment.

The Linux distribution running on Intel Galileo is an operating system born from the Yocto Project.

The Yocto Project is a Linux Foundation workgroup whose goal is to produce tools and processes that enable the creation of Linux distributions for embedded devices. The project was announced by the Linux Foundation in 2010. In March 2011, the project aligned itself with OpenEmbedded, an existing framework with similar goals, with the result being The OpenEmbedded-Core Project.

The Yocto Project is an open source project whose focus is on improving the software development process for embedded Linux distributions. The Yocto Project provides interoperable tools, metadata, and processes that enable the rapid, repeatable development of Linux-based embedded systems.

5.2 The Intel Galielo Yocto Linux

The Intel Galileo mounts by default on his small flash memory a tiny Yocto Linux distribution. This basic operating system offers only the trivial functionalities which make the board to work. These functionalities are a boot loader, a stable Linux running a minimal set of processes and the most important of these processes, i.e. the Arduino sketch subsystem.

The main core of this O.S. is the Arduino sketch subsystem, that makes the board to be programmed as an Arduino device. The system is also equipped with a special program, the Arduino sketch interpreter. This program runs the Arduino sketch uploaded on the system by the developer.

Thus, this tiny Yocto only provides the software implementation of the Arduino specifications. It is the ground on which the Arduino software lives. The Arduino API and user software is translated by the Arduino

IDE compiler into a Linux ELF executable. The tiny Yocto image has the minimum required functionalities to run this executable, no more.

So, when we switch on the board for the first time, we have a minimal Linux system which does one thing: emulating a Arduino system. At this point we could only program the board as described in "Chapter 4 – The Arduino Side", as a classical Arduino system from the standard Arduino IDE.

But, the Intel Galileo is able to boot a full Yocto Linux distribution from the embedded miniSD card reader on the board. The first important improvement of this second approach is that we have a persistent FileSystem to write on it persistent data. Also, this second approach permits to setup a generic sized Linux system, limited only by the SD card size.

The Intel support provides some version of the full Yocto Linux for Galileo. In this document we work with one of these.

This is a Yocto Linux image with EGLIBC (the embedded system version of the GLIBC library). It is equipped with GCC, MAKE; PYTHON, NODE JS, and other developing tools. The presence of a C/C++ compiler allows both the developing of C/C++ software and the making and installations of all the Linux environment software. Software can be installed also from the official Intel Galileo repository through the OPKG packets manager. We see the details in the following.

5.3 Environment Setup

In the following we explain how to setup the environment to get the maximum from the Linux capabilities of our Intel Galileo board.

We suppose to work on a Windows working PC. For Mac and Linux users the steps are the same.

The hardware we need is the following:

1) an Intel Galileo board and the provided power supply cable;
2) a 4GB miniSD card and a miniSD to SD card adapter;
3) an Ethernet cable;
4) a UART RS-232 serial cable .

5.3.1 Install Yocto Linux

The first step is to install a full Linux image on our board. As we said before, the default Linux booting on the board is a minimal Linux system which provides only the Arduino features of the system. To exploit the total Linux power of our Intel Galileo, the first thing we need is the image of this full Linux operating system.

To install the Yocto image on the 4GB miniSD card we need to perform the following steps:

1) download and install on our working PC the SDFormatter v.4 software found at https://www.sdcard.org/downloads/formatter_4/;
2) insert the 4GB miniSD card into your PC card reader through the miniSD to SD car adapter;
3) open SDFormatter and format the miniSD card. Setup SDFormatter as shown in Figure 4-1;

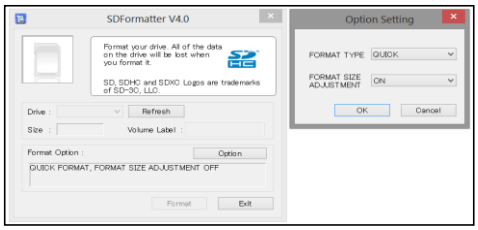

Figure 5-1. SDFormatter

4) download the full Yocto Linux image from http://intel-software-academic-program.com/download/galileo_yocto_tiny.zip;

5) extact the content of the zip into a folder. You have the following files:

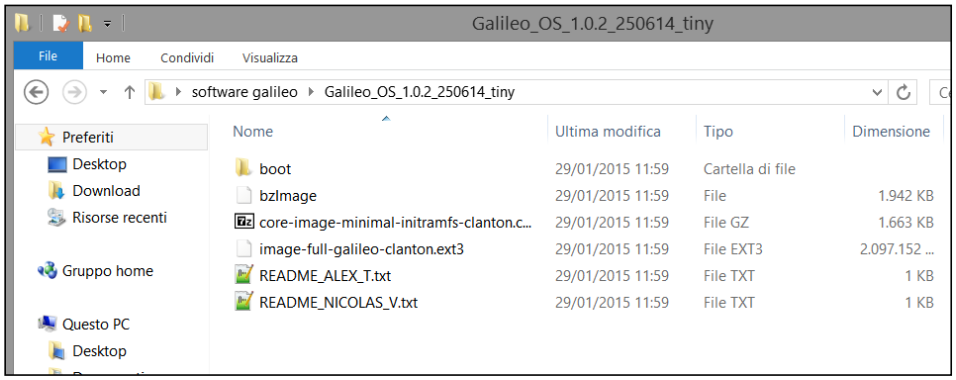

Figure 5-2. Extracted Yocto image files

6) copy the extracted files as they are into the previously formatted miniSD card;

7) put the miniSD card into the miniSD card reader on the Galielo board and plug in the power supply cable with nothing else connected to the board. The system will boot the full Yocto Linux on the miniSD card.

5.3.2 Login to the Yocto Linux

Booted the full Linux system on the board, we want to access to it. As described in the previous chapters, the Intel Galileo does not have any video output. So, to connect to the Galileo Linux command-line we can realize a connection link between the board and our working PC.

This can be accomplished in two main way. The first is through a serial connection. As shown in picture 5.4, the Intel Galileo is equipped with a UART RS-232 serial adapter (the six pins at top left). To realize the connection we need only the UART RS-232 USB cable depicted in figure 5.3. The installed Linux system has a serial server daemon running on it.

A second way to connect to the board is through a SSH connection via Ethernet. The installed Linux system has a SSH server daemon running on it . As we know, Galileo has a on-board Ethernet adapter and almost all PCs has one. But, as we just installed and booted for the first time the full Linux on the board, the Ethernet interface on Galileo is not configured by default for this (it should be configured in a static setup coherently with the working PC Ethernet interface), and we still have the problem of connect to the board to edit the network system properties.

So, the first option (the serial connection) is what we choose.

To connect to the board from our working PC via the UART RS-232 we need to plug the UART RS-232 serial cable (figure 5.3) to the Intel Galileo on the six pin UART adapter (figure 5.4) and to the PC to any USB port.

Figure 5-3. UART RS-232 to USB cable

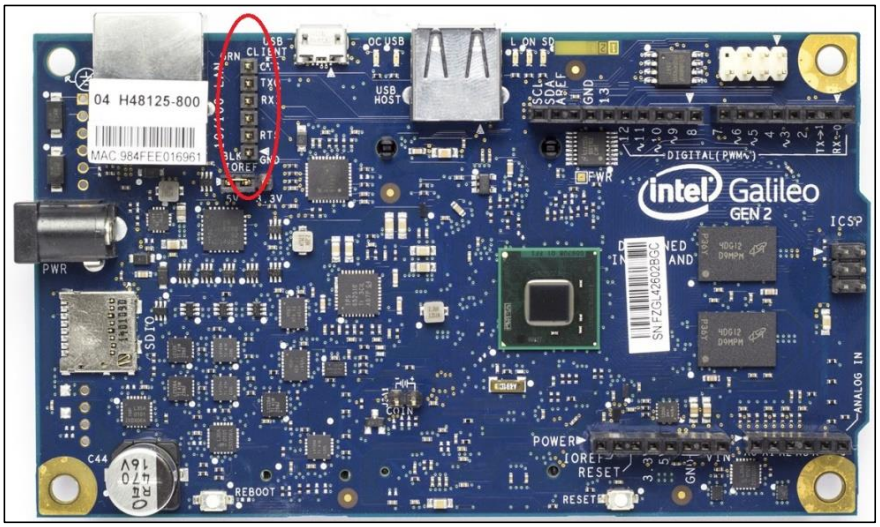

Figure 5-4. UART RS-232 adapter on Intel Galileo Gen 2

Now we need a client software installed on our working PC to connect to the board. We use Putty (it can be downloaded from http://www.putty.org/). After the installation, open Putty and setup as shown in the following picture.

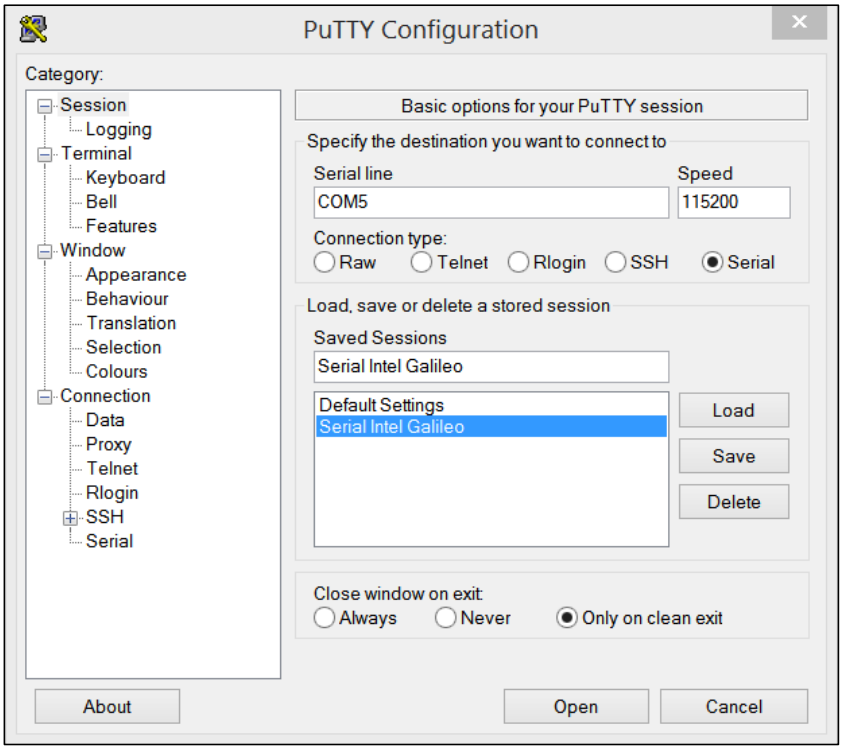

Figure 5-5. Putty configuration

As shown, we have to check "Serial" as connection type, "115200" as speed and the name of the serial line. In the example the link is on serial port "COM5". To check your port configuration, just look on the "Ports" item into Windows Device Manager.

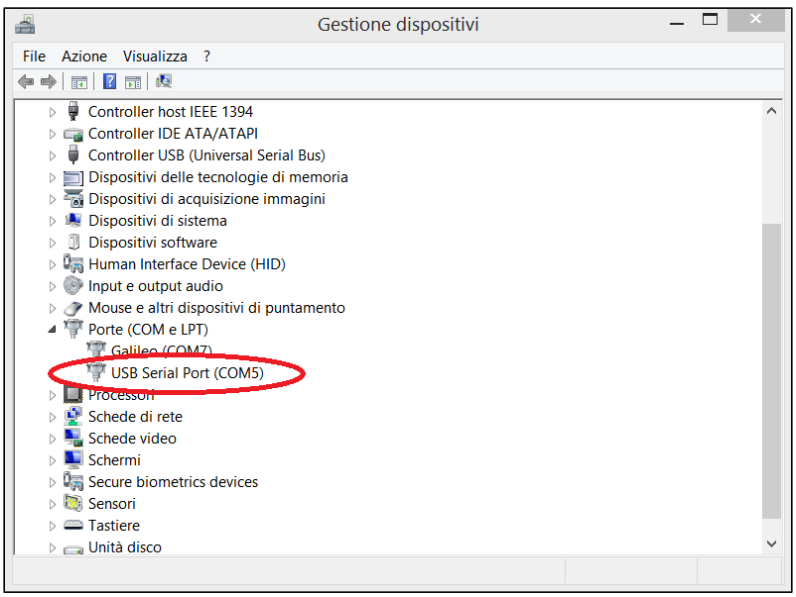

Figure 5-6. Serial port configuration on Windows Device Manager

Click “Open”. Now you should see the Intel Galileo Yocto Linux terminal on your Putty (if you see a black screen, press “Enter” once). Login as “root” and you are in. In the following, all Linux commands targeted to the Galileo board will be launched from this Putty remote terminal.

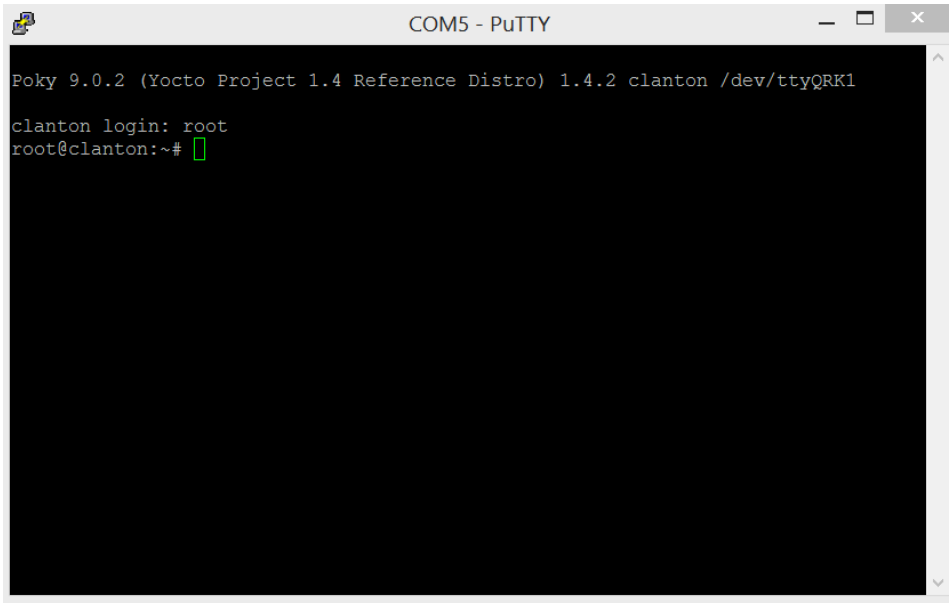

Figure 5-7. Yocto Linux command-line with Putty

5.3.3 Setup the Environment to connect Galileo to the Internet

At this point we have a full Linux distribution running on our Galileo. This Linux image embeds some developing tools as a C/C++ compiler, the Python interpreter and so on. But, in addition to the preinstalled software, the operating system runs the OPKG packets manager to install the add-on software available on the Intel Galileo official repository at http://repo.opkg.net/galileo/ . This is the main reason why an Internet connection on the board is very useful also for testing aims.

The situation is the following:

- we want a developing environment to program the board from our working PC;
- the board has an Ethernet adapter; an add-on Wireless adapter also exists, but we do not have it;
- we do not have an Internet wired access point nearby;
- our working PC is connected to the Internet with its Wireless adapter.

Our solution is to connect the Intel Galileo and our working PC through an Ethernet cable, and make our working PC to share its Internet connection with the Galileo board.

Also, this configuration creates a direct network connection via Ethernet between the PC and the Galileo board. As we see later, this feature will be exploited for a variety of purposes.

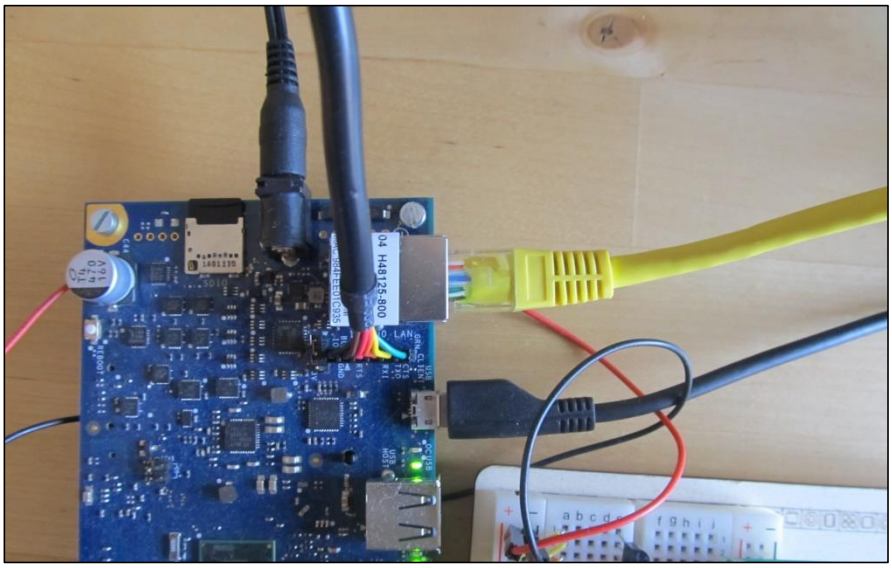

Figure 5-8. Cables connection on the Intel Galileo board

Figure 5-9. Cable connection on the side of our working PC

First, we have to configure our PC Wireless interface to share the Internet connection on the Ethernet interface. To do this, go to "Control Panel -> Network Connections -> Edit Adapter Configuration". The following window is shown:

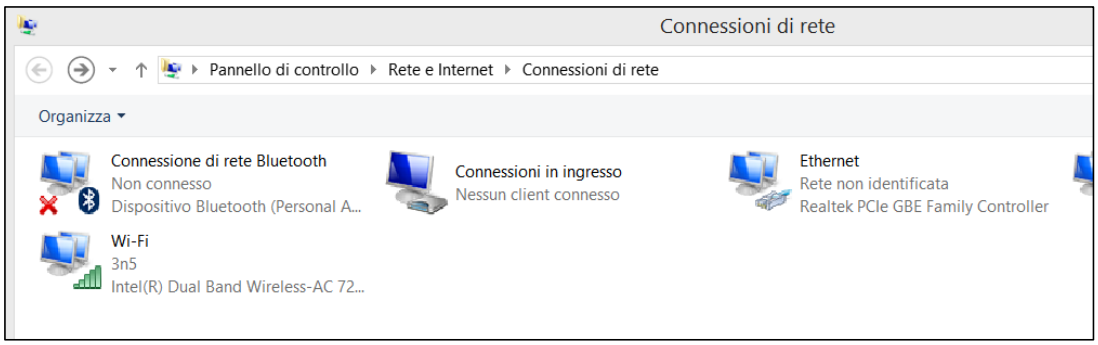

Figure 5-9. Windows Network Connections

Now, right click on “Wi-Fi” and go to “Properties”. Go to the “Sharing” tab as shown in the following picture.

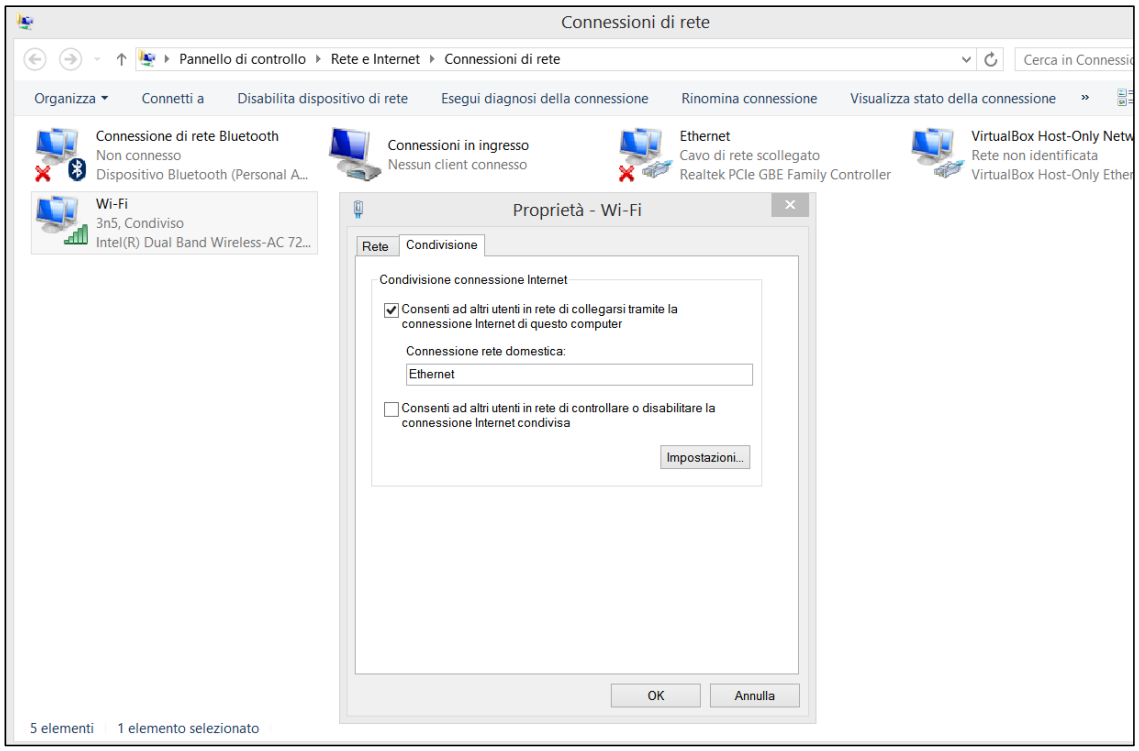

Figure 5-10. Wireless adapter “Sharing” properties

As picture 5.10 suggests, we have to check the “Allow sharing..” checkbox and select the “Ethernet” connection as sharing target. Once done, a confirmation dialog like in figure 5.11 appears, warning us that, as consequence of our new settings, the IP address 192.168.137.1 will be set on our PC Ethernet interface. Confirm by selecting “Yes”.

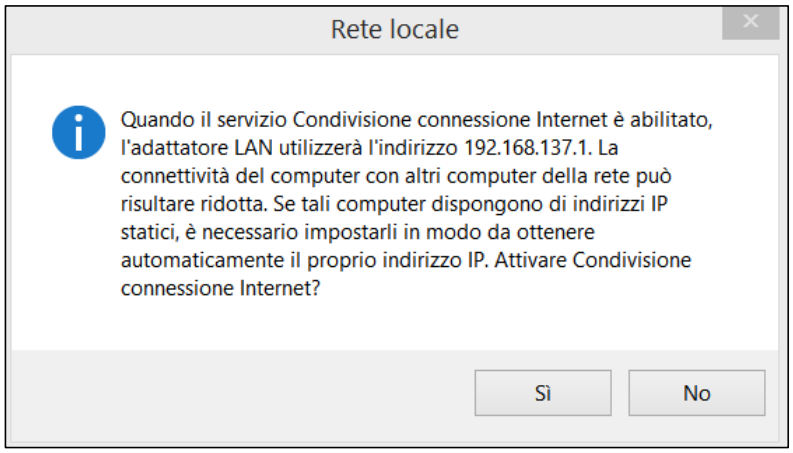

Figure 5-11. Confirmation dialog

To check the new configuration you can open a Windows terminal and type "ipconfig /all" command. Locating the Ethernet interface properties, the result should be like this:

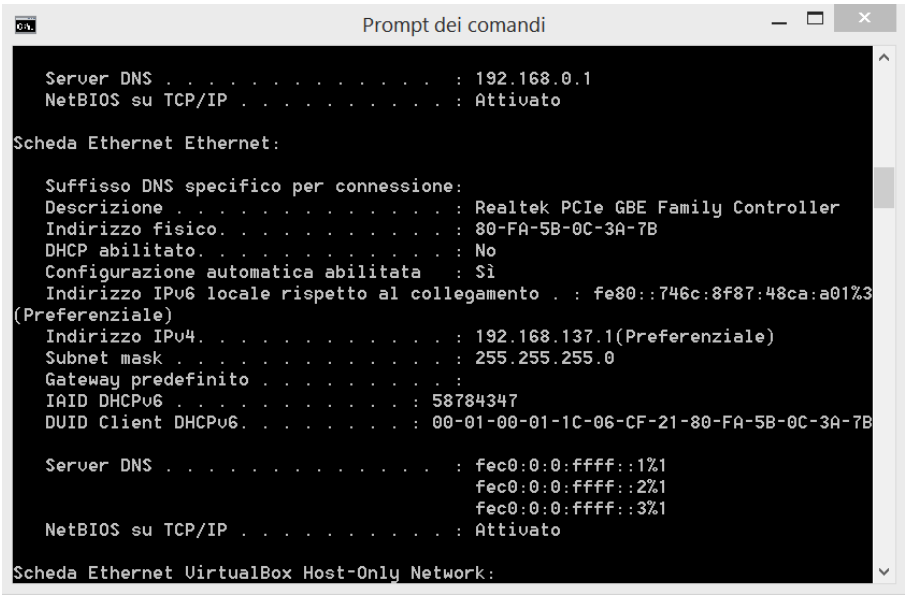

Figure 5-12. Check Ethernet IPv4 configuration on the Windows terminal

We are creating the 192.168.137.0/24 LAN and our working PC has address 192.168.137.1. To make the things to work, our Intel Galileo Ethernet interface has to expose a coherent IP address. So, we simply set it to 192.168.137.2. Configuration file for networking is at "/etc/network/interfaces". Save a copy of the original file with:

cp /etc/network/interfaces /etc/network/interfaces.backup

Edit the *"interfaces"* file as shown in the following picture (you can use the *"nano"* editor on Linux terminal).

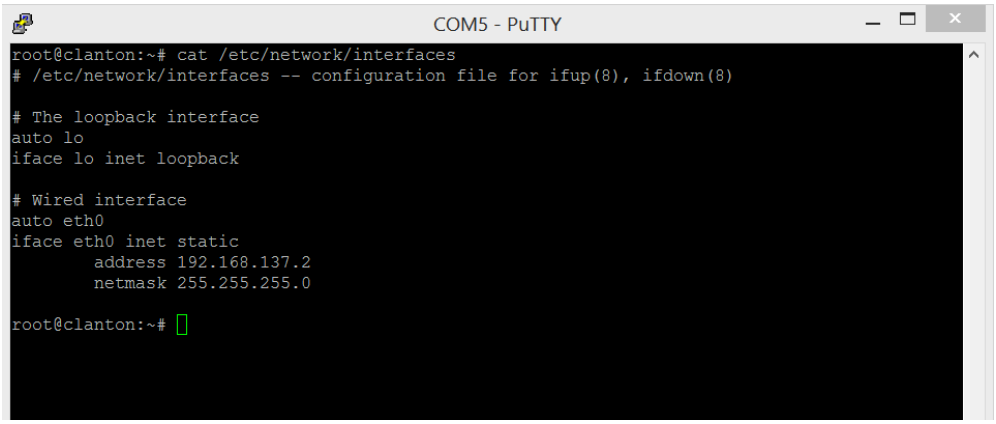

Figure 5-13. Yocto Linux Ethernet IPv4 settings

To enable the new Ethernet configuration you have to run the following command:

/etc/init.d/networking restart

At this point we have a duplex Ethernet connection through the Intel Galielo board (IP 192.168.137.2) and our working PC (IP 192.168.137.1). To test our connection we can "ping" from one side to the other. As we

see in the next section, it will be very useful for different purpose. Furthermore, our Intel Galileo is now connected to the Internet through our PC which works as connection bridge. Also this aspect is very useful, as we will see soon in this chapter.

5.3.4 Setup a File Transfer Environment

A very useful feature to ease the developing on the Intel Galileo is to have a file transfer environment to manage the Linux FileSystem on the board from our working PC. As we saw in the previous section, we have a duplex connection via Ethernet between the two machines, and this feature enables the setup this file transfer environment.

WinSCP is a free file transfer client for Windows. You can download and install it from http://winscp.net/eng/download.php . The Secure Copy Protocol (SCP) is a file transfer protocol over SSH. The Yocto Linux image we installed on Intel Galielo runs by default a SCP server. The following figure shows the parameters to set on the WinSCP login interface.

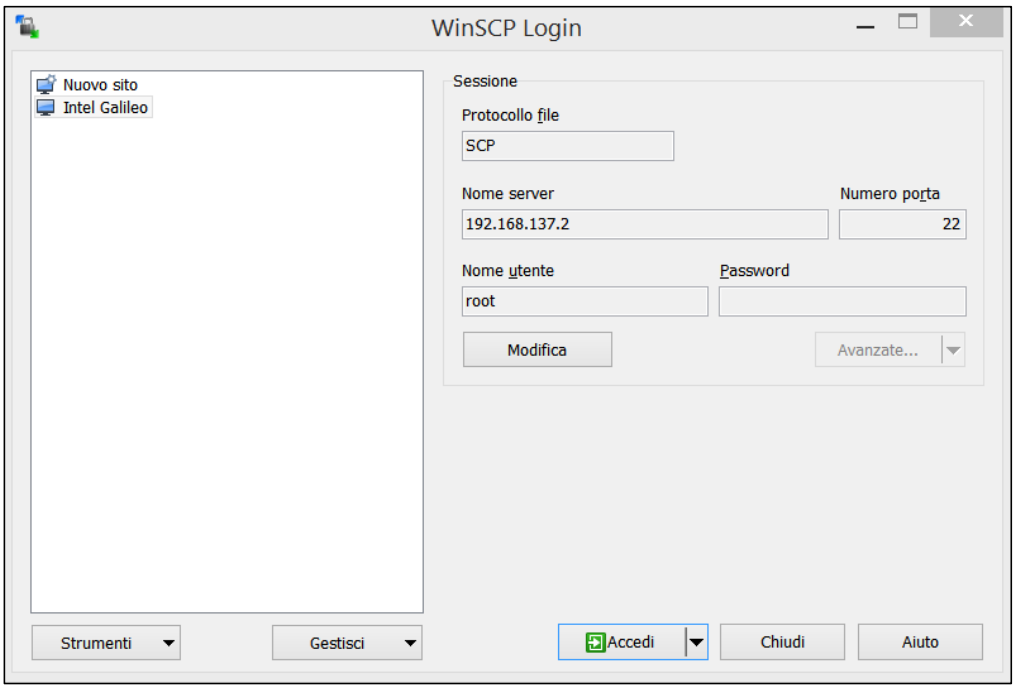

Figure 5-14. WinSCP connection settings

When logged in, you can see the Linux FileSystem on the WinSCP GUI.

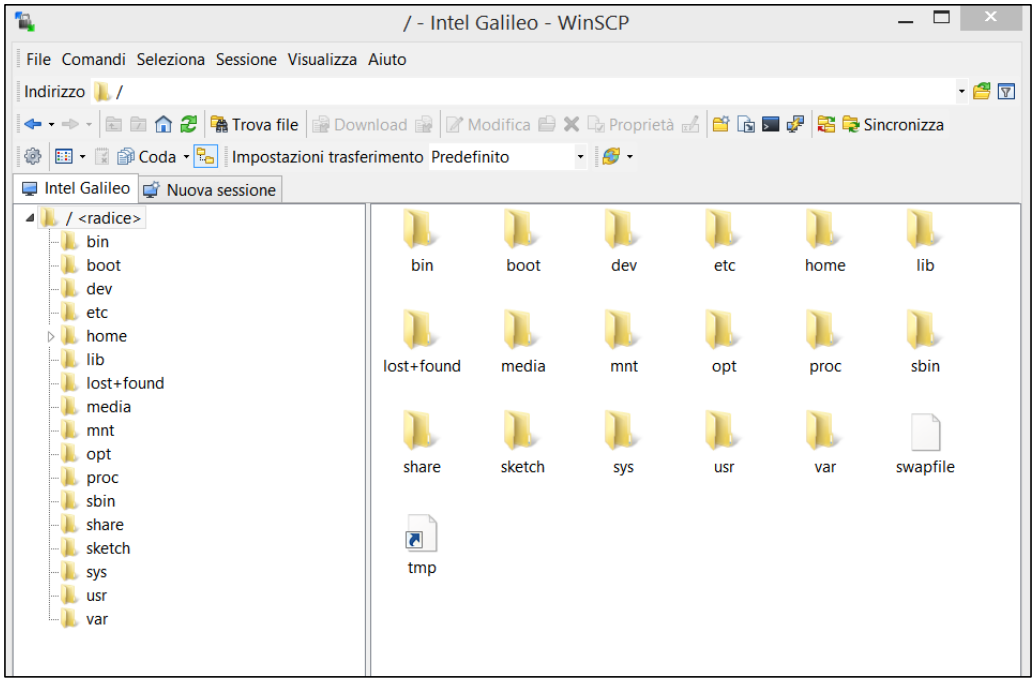

Figure 5-15. The Yocto FileSystem on WinSCP

At this point, the developer could manage the Linux FileSystem on the board directly from his PC. This very useful as the Intel Galileo has no video interfaces. So, for example, the maker could realize the software on his PC and easily transfer it on the board at last. As we see in Chapter 6, during our experimentation we make a wide use of this function.

5.3.5 Expand the Yocto Linux with more Software

At this point we have a fully operative developing environment. Our Intel Galileo is ready to be programmed on the Arduino side from the Arduino IDE. On the Linux side the maker can work from the Putty command-line and the WinSCP client. The two tools enable remote command launching and file transfer to and from the board.

Also, as previously said, an official Intel Galileo software repository exists. The board runs the OPKG packets manager. In this section we configure it and show how to install some new software.

First of all, we need to setup the OPKG with the official repository parameters. This is accomplished by editing the default *"/etc/opkg/base-feeds.conf"* file. Make a backup of the file with:

cp /etc/opkg/base-feeds.conf /etc/opkg/base-feeds.conf.backup

Edit the default file by inserting the three lines is shown in picture 5.16:

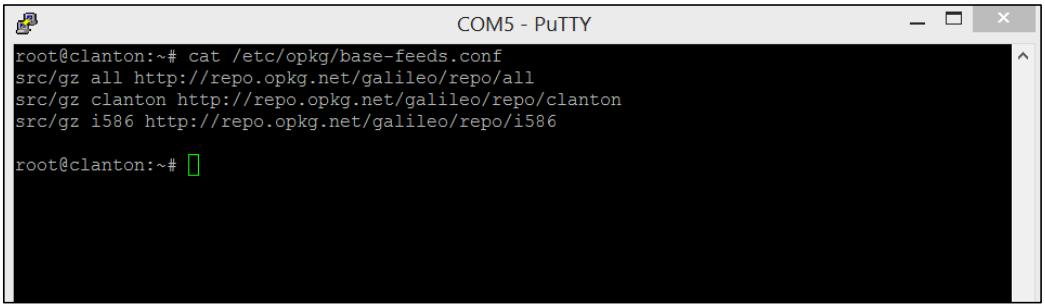

Figure 5-16. The OPKG configuration file

To make active the new configuration, type the following command which updates the local copy of packets list from the web:

opkg update

Now our Intel Galileo is ready to install the official Intel available software. The command to install the "X" packet is *"opkg install X"* The available packets can be listed on the web page http://repo.opkg.net/galileo/i586/ , as depicted in figure 5.17.

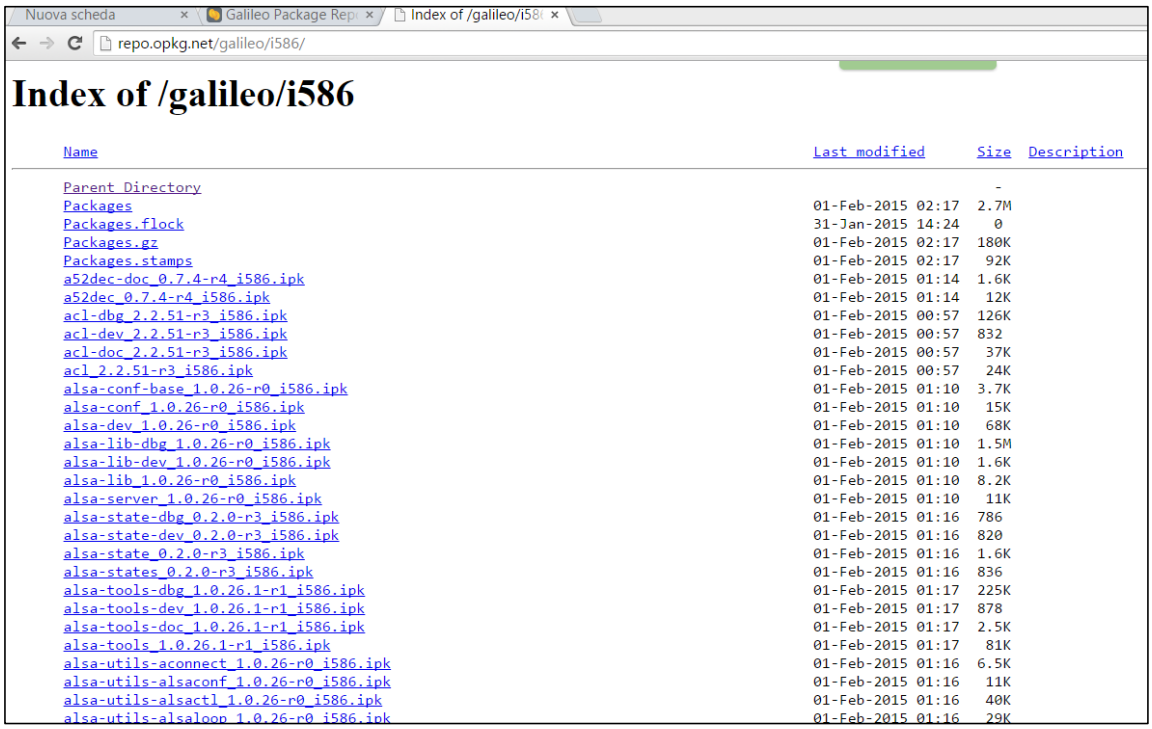

Figure 5-17. A sketch of the official Intel Galileo packets list

As we described previously, the Linux image we downloaded from the Intel Galileo official page provides some interesting developing tools like a C/C++ compiler, the Make utility, the Python interpreter and so on. But, as our aim is to experiment the Intel Galileo for the Internet-Of-Things, we need some extra software on our board. As we see later in Chapter 7, this extra software will allow us to remotely control our Intel Galileo embedded system from the web.

In this regard., we install on the board the Apache2 HTTPD to have a full web server. In the following of this document we see how to provide a web based control interface to manage the system from the Internet. Together with the Apache2 we install the PHP5 module to develop and provide to the remote administrator a board control dynamic web application.

On our Putty remote command-line we have to type the following commands:

opkg install apache2

opkg install modphp

The first command installs on the board the Apache2 Web Server, the second command install the Apache PHP module. By default, the Apache2 web server is installed into the "/usr/local/apache2" folder. Type the following command to run the Apache2 web server:

/usr/local/apache2/bin/apachectl start

To test if the installation was successful, open a browser on your working PC and go to the address 192.168.137.2; the following web page should appear:

Figure 5-18. Apache2 providing a test web page

Now, to test if the PHP module installation was successful, create somewhere on the Linux FileSystem the following *"info.php"* script file (we can do this directly on the Putty terminal or creating the file on our working PC and then copying it on the Linux FileSystem through the WinSCP tool):

```
<?php
// Show all information
phpinfo();
?>
```

Run the script with the command:

php info.php

The following output should appear:

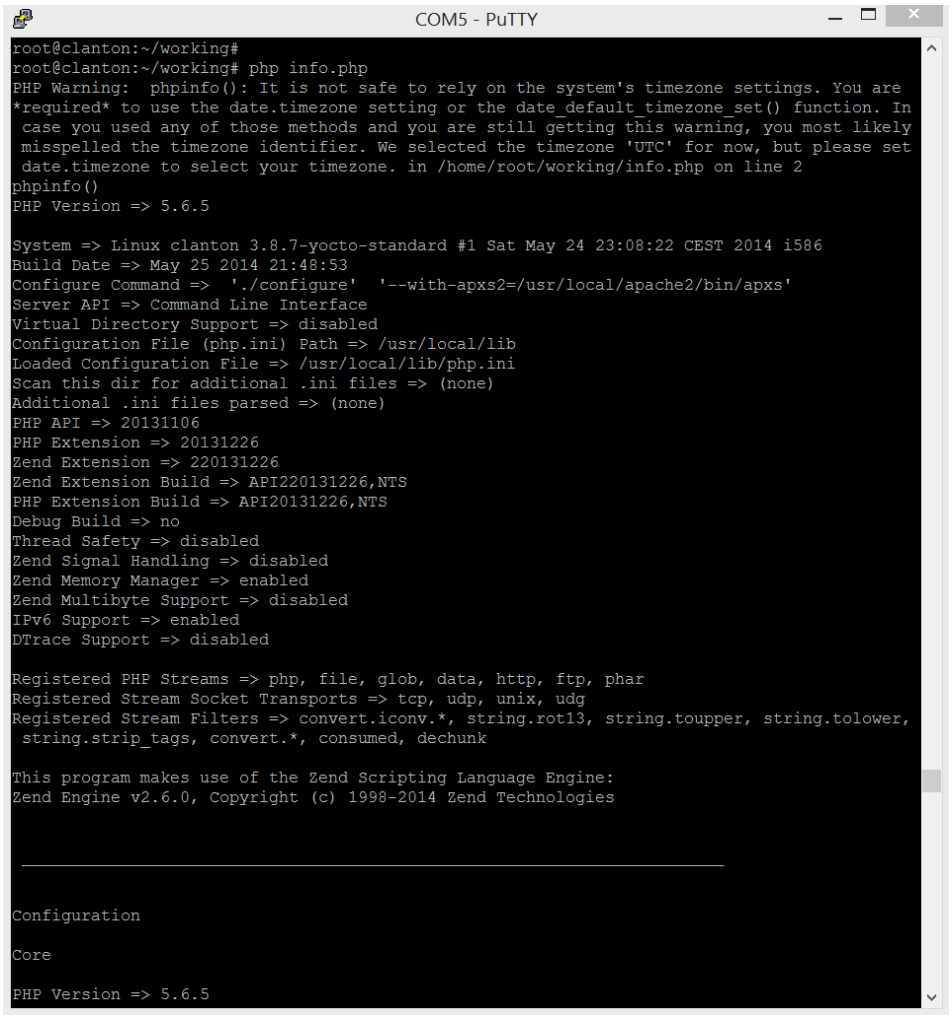

Figure 5-19. PHP5 info test

Now we check if the installation has correctly linked the Apache2 web server to the PHP module. To do this, check if the following lines are in the *"/usr/local/apache2/conf/httpd.conf"* configuration file. If for some reasons they are not, simply add them to the file:

```
LoadModule php5_module modules/libphp5.so
<FilesMatch \.php$>
    SetHandler application/x-httpd-php
</FilesMatch>
```

The first line is to load the PHP interpreter as Apache web server module. The next lines are to make Apache2 to forward the HTTP request of PHP resources to the PHP interpreter module.

Type the following command to restart the Apache2 web server to make sure the new configuration is validated:

```
/usr/local/apache2/bin/apachectl restart
```

At this point we have a web server for dynamic web applications running on our small Intel Galileo board. As the reader simply understand, it is a big step towards the Internet-Of-Things world.

Now we have all we need to begin our experimentation on the Intel Galileo platform. We can program the board as Arduino platform and we have a complete Linux embedded system to exploit to realize our testing projects. The next section 5.3.6 deals with an interesting matter: the installation on the Intel Galileo of generic software from source files. But, the environment setup ends here. Thus, the disinterested reader could skip section 5.3.6 to jump at Chapter 6 where we describe our first experiments on Galileo. As we will see with our most advanced projects in Chapter 7, the main goal of our experimentation is to exploit the full power of the board by working synchronously on both the Arduino and the Linux side of the Intel Galileo platform.

5.3.6 EXPAND THE YOCTO LINUX WITH UNOFFICIAL SOFTWARE

As we said, in the previous section we ended our environment setup. We installed some software from the official Intel Galileo repository through the OPKG packets manager. This is the main way to install software on the Intel Galileo: easy, reliable, complete. But, as the official repository offers only a small set of programs, we tested also the installation of software by compiling and installing from source files.

We did it trying to compile and install the software we installed previously, i.e. the Apache web server and the PHP5 interpreter. We removed the previously installed packets before. We created on the Galileo Linux FileSystem the working directory *"/home/root/working/temp"* to download temporary source packages. Let see our tests and results.

5.3.6.1 APACHE2 HTTPD

To install on the Yocto Linux the Apache2 web server we performed the following steps:

1) download the Apache2 source package from http://httpd.apache.org/download.cgi . We downloaded the 2.2.29, but any version works fine.
2) through WinSCP, copy the downaloaded package on the Yocto FS on the board into *"/home/root/working/temp"*.
3) go to the Putty console and change directory to the "temp" folder:

   cd /home/root/working/temp
4) run the following command to extract, configure, make and install the Apache2 web server:
   a. *gzip – d httpd-2.2.29.tar.gz*
   b. *tar xvf httpd-2.2.29.tar*
   c. *cd httpd-2.2.29*
   d. *./configure*
   e. *Make*
   f. *Make install*

By default, the Apache2 web server is installed into the *"/usr/local/apache2"* folder. Although the procedure takes some hours to complete (this is due to the 400Mhz processor), everything runs correctly and the procedure ends with success.

5.3.6.2 PHP5

To install the PHP5 interpreter we first need to install the LIBXML2 library:

1) download the "libxml2" library source package from http://xmlsoft.org/downloads.html ;

2) through WinSCP, copy the downloaded package on the Yocto FS on the board into *"/home/root/working/temp";*
3) go to the Putty console and change directory to the *"temp"* folder:

   cd /home/root/working/temp
4) run the following command to extract, configure, make and install the LIBXML2 library:
   a. *gzip – d libxml2-2.9.1.zip*
   b. *cd libxml2-2.9.1*
   c. *./autogen.sh*
   d. *./configure --without-python*
   e. *Make*
   f. *Make install*

The *"--without-python"* is to exclude from installation the LIBXML2 Python binding. It is necessary because the Galileo Linux system does not have all Python libraries needs to correctly compile. By default, the LIBXML2 library is installed into the *"/usr/local/lib"* folder. Also in this case, although the procedure takes a long time to complete, the procedure ends with success.

To install on the Yocto Linux the PHP5 interpreter we performed the following steps:

5) download the PHP5 source package from http://php.net/downloads.php . We downloaded the 5.6.5, but any version works fine;
6) through WinSCP, copy the downaloaded package on the Yocto FS on the board into *"/home/root/working/temp";*
7) go to the Putty console and change directory to the *"temp"* folder:

   cd /home/root/working/temp
8) run the following command to extract, configure, make and install the PHP5 interpreter:
   a. *tar xvf php-5.6.5.tar*
   b. *cd php-5.6.5*
   c. *./configure --with-apxs2=/usr/local/apache2/bin/apxs*
   d. *Make*
   e. *Make install*
   f. *Cp php.ini-development /usr/local/lib/php.ini*

By default, the PHP interpreter is installed into the "/usr/local/php" folder. The *"--with-apxs2=/usr/local/apache2/bin/apxs"* is to set the PHP interpreter as Apache module. Also for the PHP interpreter, although the procedure takes some hours to complete, the procedure ends with success.

Thus, everything seems to work fine on our Intel Galileo. The last test we did was trying to install an expanded version of the PHP interpreter, including the SOAP and the SOCKETS PHP extension to the interpreter. Why we chose these extension will become more clear in Chapter 7 during our experimentation on the board. To do this, the procedure is the same seen above, except for the *"./configure"* command which changes as follows:

./configure --with-apxs2=/usr/local/apache2/bin/apxs –enable-soap –enable-sockets

But, with our sorrow, during the *"make"* process we ends with the following error:

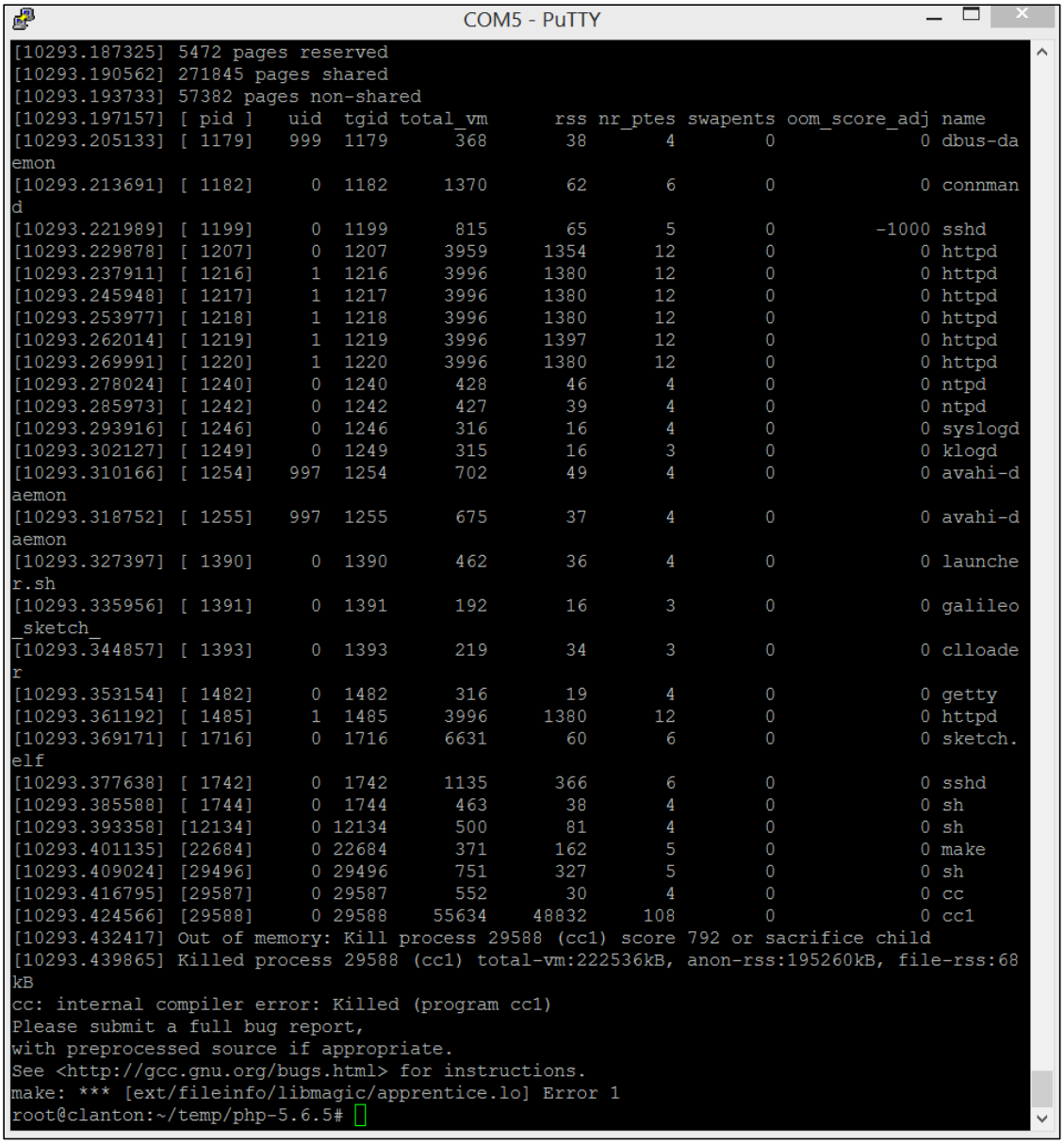

Figure 5-20. Out of memory error, the compiler process is killed

During the *"make"* procedure, after some hours of work the compiler depletes all the system primary memory, and the operating system kills the compiler process. We tried the same procedure more times, but we always ended with the same error.

As solution, we tried to set on the Galileo Linux system a SWAP partition to cope the lack of primary memory during a RAM intensive process, but we found out that the Galileo Linux operating system provided by Intel does not include the SWAP partition support. As the official Intel Galileo documentation suggests, the Galileo Yocto Linux could be rebuilt to include any tool in it, thus also the SWAP partition support. We suppose this could be a solution, but we did not try this way.

Our conclusion about this test is that almost everything could be installed on our Intel Galileo Linux platform, both official and generic software. Sometimes we could stumble on the hardware limitation of the board. But, as previously discussed, these limitations could be circumvented by software solutions.

6 Basic Experiments

6.1 Introduction

We saw in Chapter 1 how the Internet-Of-Things world spaces from the natural environment to the city one, from a sensor to catch and count our heartbeats to sensors and actuators of our home thermostat. The first natural consideration could be that this is something for a narrow range of people highly skilled in electronics and computer science. And is from the disproof of this idea that we can learn a very important concept about the Internet-Of-Things: it is a technology for everyone. Usually, when talking about electronic devices like PC, we make a natural differentiation between engineers who realize the product and the final user of this product. The generic PC user knows only the one percent of what is happening under the case when he is browsing the web or during a play of a video on his laptop. But, as the Internet-Of-Things is supposed to be for everyone and has the aim to reach all the aspects and situations of the everyday life of a person, the technology which realizes the IoT must not be a total black box for the final user. This final user will control, customize and edit for his personal needs the IoT device. The distance between producer and final user is shorter than before, and often the two figures overlap.

The considerations above are useful to understand the architectural philosophy behind the design of the IoT platform we are going to analyse, the Intel Galileo.

As we saw in Chapter 3 with the detailed description of the board and in Chapter 4 with the Arduino side setup, the Intel Galileo is an electronic device exposing an high level interface to the maker, the Arduino side of the board. As the Arduino concept states, the user must not be an engineer, he can be an hobbyist, a curious who want to enter for the first time the world of microcontroller programming, a maker who has a nice idea but any knowledge of electronics or programming and who needs a way to easily prototype his project.

But, at the same time the Intel Galileo is a complete Linux system which requires computer science or at least system and programming skills to be exploited. This is why, when we focus on the Intel Galileo platform in the following, we approach the experimentation as engineers. This is necessary to exploit the full power and features of the board to catch the philosophical idea behind his design and implementation. But, as engineers approaching the Internet-Of-Things world, our final aim is to use our knowledge of computer and programming to provide a "how to work" method to the generic user of the Intel Galileo.

This chapter describes various projects experimented with the Intel Galileo. The test cases are presented in increasing order of complexity. As clarified below, we describes both successful and failed attempts. Failed attempts are in fact of great interest in the understanding the philosophy of the board, what are its limitations, and therefore what types of projects can be realized and which cannot.

The source code files of the projects in this chapter and in "Chapter 7 – Advanced Experiments" can be found at the web page http://tinyurl.com/ndgwlsh .

At this point, the reader knows that the Intel Galileo board has two faces: the Arduino and the Linux one. From "Chapter 3 – The Intel Galileo", he also knows that the GPIO Arduino pins can be accessed from the Linux system by reading and writing files on the Linux FileSystem. Thus, the first thing we want to test is if the Arduino side and the Linux side of the board are absolutely equivalent or some differences exist. I.e., we want to show if from the Linux side the maker could rebuild as Linux programs the same projects he realized from the Arduino side of the board, getting rid of the Arduino features of the platform. This is an interesting topic because, as the Arduino development environment is very easy to use but also very essential, on the Linux side it is a bit more complex to work but we have the power of a complete general purpose operating system. So the question is: cloud we dispose the Arduino IDE and develop our projects totally and directly as Linux software?

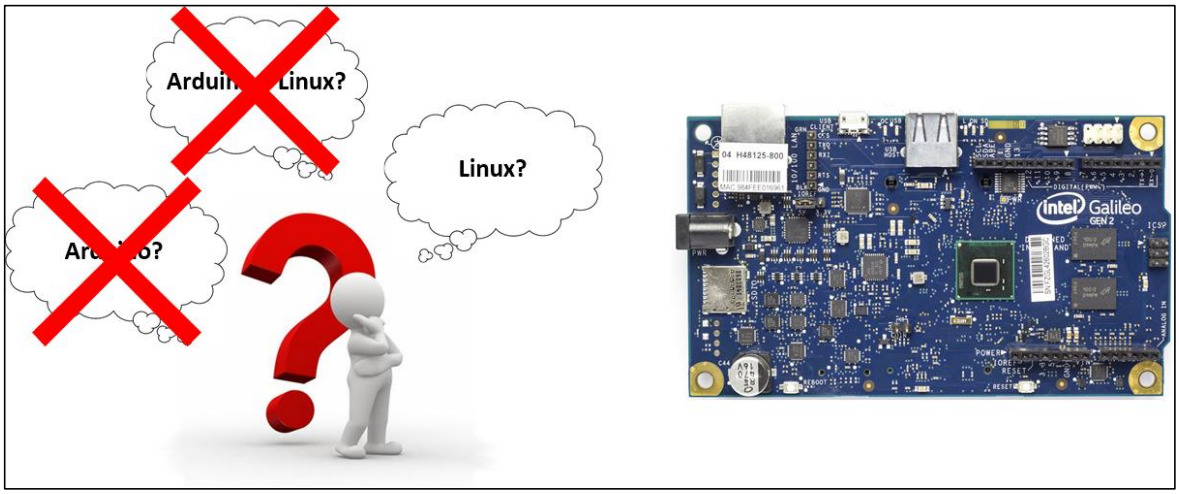

Figure 6-1. Could we dispose the Arduino software stack?

We begin our experimentation in section 6.2 starting with a simple example where everything seems to answer as positive to the previous question. But, as we switch to a bit more complex project in section 6.3, we see that the response to our question is "no". Let see why in the following of this chapter.

6.2 First Test Case: Simple Temperature

Our first test case is a simple temperature controller. To build the circuit we need the following components:

- a temperature sensor;
- a led;
- a 220Ohm resistor;
- some wires.

It is important to specify that all the electronic components we use in this paper are official Arduino components.

We use the wires to connect the temperature sensor to analog input pin A0 and the led (throught the 220Ohm resistor) to digital output pin 7. The following picture shows the schema.

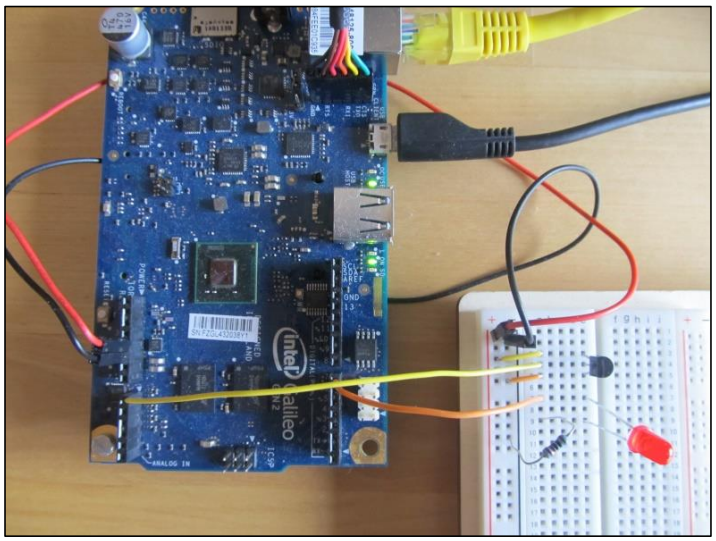

Figure 6-2. Circuit of the first test project

We want to realize a simple control program which reads the value returned by the temperature sensor. If the sensor returns a value corresponding to a temperature higher than a baseline temperature, then we switch on the led. We switch off the led otherwise.

As previously said, the first thing we want to understand is if the Arduino side of the board could be dispose to work totally on the Linux side. We are interested in this to answer the following question: if we have a complete Linux running on the board and if the final result of the Arduino IDE is an ELF executable file (as we described in "Chaper 3 – The Intel Galileo" in the software stack description section), why limiting our work to the Arduino IDE (which is an essential IDE) and to the Arduino programming language when we could develop a standard C/C++ (or other PLs) using a professional IDE (e.g., Eclipse)?

Thus, to answer our questions, we first implement the project as Arduino program, and then we try to rebuild the software as Linux executable forgetting for a moment that we are on a Arduino-compatible platform.

We begin in the following section with the Arduino implementation.

6.2.1 THE ARDUINO SKETCH IMPLEMENTATION

The first thing we do is to implement this test case as Arduino sketch. The following code (*temperature.ino*) can be written and compiled in the Arduino IDE, and then uploaded to the Galileo board.

```
const int sensorPin = A0;
const float baselineTemp = 10.0;
const int ledPin = 8;

void setup() {

Serial.begin(9600); // opena serial port

pinMode(ledPin, OUTPUT);
digitalWrite(ledPin, LOW);
}

void loop() {
// put your main code here, to run repeatedly:

int sensorVal = analogRead(sensorPin);

Serial.print("Sensor value: ");
Serial.print(sensorVal);

// convert the ADC reading to voltage
```

```
float voltage = (sensorVal / 1024.0) * 5.0;
Serial.print(", Volts: ");
Serial.print(voltage);

// convert the voltage to temperature in degrees
float temperature = (voltage - 0.5) * 100;
Serial.print(", degrees C: ");
Serial.print(temperature);


Serial.print("\n");

if(temperature > baselineTemp) {
digitalWrite(ledPin, HIGH);
}
else {
digitalWrite(ledPin, LOW);
}
```

The code is a standard utilization of the Arduino APIs. The program is easy to write and works out of the box on our Intel Galielo. While running the program on the board, we can see the red led switching on and off depending on the current sensed temperature with respect of the baseline temperature value.

This is the first proof that our Intel Galileo is a complete Arduino-compatible system. As we studied the board architecture, we know that the Arduino sketch is actually translated into a Linux binary executable and that this executable actually implements the dialog with the external hardware through the GPIO files linked by the firmware to the GPIO hardware pins. But, to run the above sketch we could not know that a Linux distribution is running under the hood.

In the next section we switch to the Linux side of the board. The first thing we try is to exploit directly the link between the GPIO pins and the Linux FileSystem. I.e., we try to read the temperature sensor value and to switch on/off the led communicating directly with the hardware via the GPIO files on the Linux FileSystem. The test follows.

6.2.2 The Bash script implementation

As described in "Chapter 3 – The Intel Galileo", the firmware links the hardware Arduino General Purpose Input Output pins (GPIOs) to the Linux FileSystem through the stream files under the *"/sys/class/gpio/gpio*"* folders (for digital input/output and analog output) and under the *"/sys/bus/iio/devices/iio\:device0/"* folder (for analog input). It means that, under the hood, the final sketch produced by the Arduino IDE is a Linux binary which realizes the I/O to the external devices through raw readings and writings on the Linux FileSystem.

In this section we try to exploit this link between the GPIO pins and the Linux FileSystem. We forget for a

moment that we are developing on a Arduino-compatible platform. The only thing we know here is that we are on a Linux system that is equipped to general purpose pins hardware and that the software can communicate with this hardware via files on the FileSystem.

The following is the Linux Bash version (*temperature.sh*) of the code seen above. The script runs directly by the Galileo board Linux command-line on Putty with the *"bash temperature.sh"* command.

As we can see, the process communicates with the hardware through the GPIO files on the Linux FileSystem. The program reads the analog temperature sensor value from the "*/sys/bus/iio/devices/iio\:device0/in_voltage0_raw*" file and writes the digital led output value to the "*/sys/class/gpio/gpio40/value*" file.

Note that the Bash script uses directly the sensor returned value without the arithmetic to compute the temperature in Celsius Degrees. This is because the Bash language does not support floating point arithmetic.

```
#! /bin/bash

# Initializing digital output pin number 8
echo -n "40" > /sys/class/gpio/export
echo -n "out" > /sys/class/gpio/gpio40/direction
echo -n "strong" > /sys/class/gpio/gpio40/drive
echo -n "0" > /sys/class/gpio/gpio40/value

# Initialize analog input A0
echo -n "48" > /sys/class/gpio/export
echo -n "out" > /sys/class/gpio/gpio48/direction
echo -n "0" > /sys/class/gpio/gpio48/value

# Read the value on A0
sensorvalue=$(cat /sys/bus/iio/devices/iio\:device0/in_voltage0_raw)

# Infinite loop
while [ "$sensorvalue" -ge 1 ]
do
# Print value
echo "sensorvalue: $sensorvalue"

if [ $sensorvalue -le 18 ]
then echo -n "0" > /sys/class/gpio/gpio40/value
else echo -n "1" > /sys/class/gpio/gpio40/value
fi
```

```
sensorvalue=$(cat /sys/bus/iio/devices/iio\:device0/in_voltage0_raw)

# Waits 1000 ms
usleep 1000000
done

echo -n "40" > /sys/class/gpio/unexport
echo -n "48" > /sys/class/gpio/unexport
```

The above script runs fine. So, our first test is successful. We are able to access and drive the external hardware from the Linux system. At this point a question comes out: dealing directly with the raw Linux FileSystem is the unique way we can drive the external hardware or we have some high level C/C++, Python or other PLs API to use programming our software? This topic is the matter of the next test case.

6.2.3 THE C PROGRAM IMPLEMENTATION

In the previous section 6.2.2 we could drive the external hardware dealing directly with the GPIO files on the Linux FileSystem. Now we would like to catch the same result with the C/C++ programming language. We could approach this problem in the same way we did previously with the Bash script: we could read and write to the Linux FileSystem from the C code through the standard I/O C APIs. But the question is: does Intel provide to the maker a C version of the Arduino API we used to write our sketch in section 6.2.1? If Intel does, we could get rid of the raw communication to the hardware via the Linux FileSystem and exploit this API to write a C/C++ program in the same manner we did for the Arduino sketch, thus using an API which hides the low level details of the communication to the hardware.

The official Intel Galileo documentation at http://intel-software-academic-program.com/courses/diy/ signals the presence of a C Arduino API we can use to write a C/C++ control program. Through this API, is not necessary for the maker to communicate to the hardware via the Linux FileSystem as we did with our Bash script. The C Arduino API seems to work as the official Arduino API we used on the Arduino IDE to write the sketch version of the project.

The C program (*temperature.c*) follows. It uses the Intel provided C Arduino API to access the board I/O pins. The complete code can be found at the web page http://tinyurl.com/ndgwlsh .

```
// Include C Arduino API
#include ...

#define sensorPin 0
#define ledPin 8
int sensorVal = 2;
float voltage = 0;
float temperature = 0;
float baselineTemp = 10.0;
```

```
void setup()
{
printf("Setting up...");
pinInit();
pinMode(ledPin, OUTPUT);
printf("Done\n");
}

void loop()
{
printf("Looping...\n");
while(sensorVal > 1)
{
sensorVal = analogRead(sensorPin);
printf("Sensor Value: %d ",sensorVal);

// convert the ADC reading to voltage

                float voltage = (sensorVal / 1024.0) * 5.0;
printf(", Volts: %f", voltage);



                // convert the voltage to temperature in degrees
float temperature = (voltage - 0.5) * 100;
printf(", degrees C: %f", temperature);

printf("\n");

if(temperature > baselineTemp) {
digitalWrite(ledPin, HIGH);
}
else {
digitalWrite(ledPin, LOW);
}


delay(1000);
```

```
}

}

int main( int argc, char ** argv )
{
printf("\n *** LED and sensor demo *** \n\n");

interrupt_init();
init(argc, argv);

// Setting up I/O
setup();

// Looping
loop();

}
```

To make this code to work, we need to include the set of headers and .c/.cpp files presented as the C Arduino API found at http://intel-software-academic-program.com/courses/diy/. These files are at the web page http://tinyurl.com/ndgwlsh .

We can write the code on our development PC, and then transfer it to the Intel Galielo Linux FileSystem through the WinSCP utility.

We organized the code into a "C_test" folder as follows:

- A "src" folder where we put the so colled C Arduino API;
- A "obj" folder in which we find the compiled C Arduino API files;
- A "bin" folder which contains the output executable program;
- The "_test.cpp" and "_test.h" source files that implement our procedure;

- A “Makefile” to compile all the source files in the project (APIs and our code).

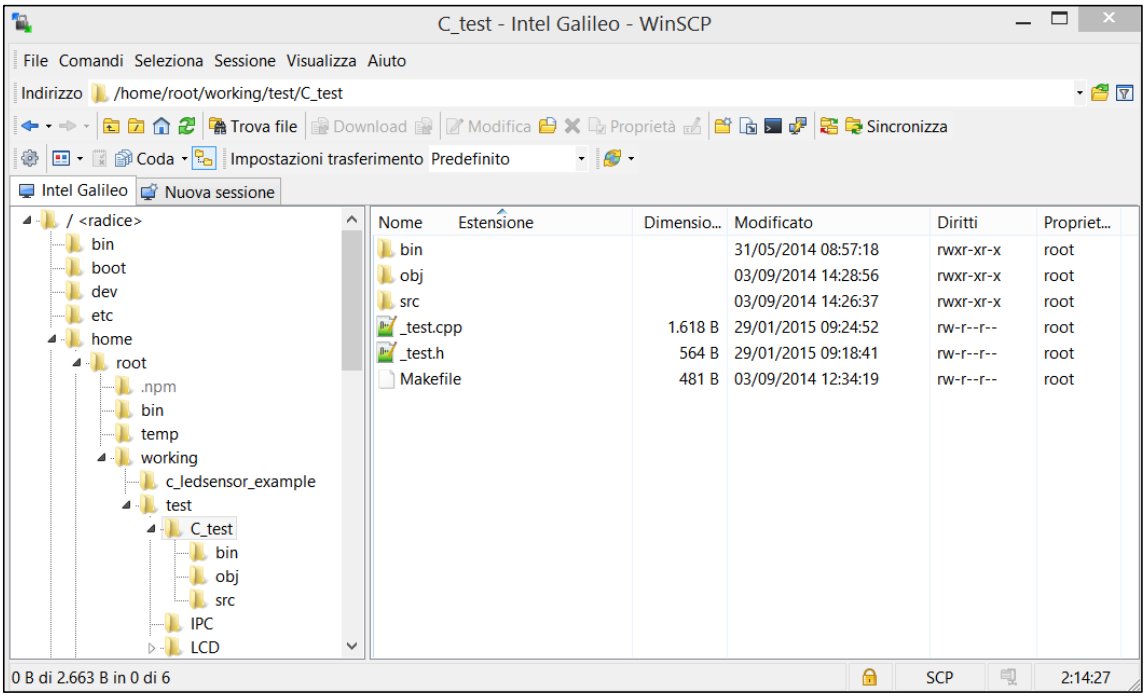

Figure 6-3. C_test folder structure

We can change directory to the “C_test” folder, launch the “make” command to compile the source files and run the “bin/_test” program. We will see our red led switching on if the temperature from the temperature sensor is greater than the baseline value.

So far, everything is working fine. In section 6.2 we presented a project to drive some simple hardware mounted on our Intel Galileo board. We developed the software as Arduino makers and as experienced Linux programmer in a complete interchangeable manner. At this point our consideration could be that we could dispose the Arduino side software stack of the board and program the Intel Galileo directly from the Linux side. But, as we see in the following section, this is a wrong thesis. In section 6.3 we describe our second test case. It is still a very simple example, but in place of the temperature sensor and the red led we mount on the board a unique two lines LCD. As we see later, dealing with a more complex external hardware is sufficient to disprove the previous thesis.

6.3 Second Test Case: LCD

In our first test case we drove two simple devices: a temperature sensor and a led. As we saw, if we have to manage a simple hardware we could directly deal with it roughly reading and writing from and to the Linux FileSystem. We did it in the Bash version of our implementation. Also, it seems we have an official C API which we can exploit to write our hardware control programs as pure C/C++ software.

In the following test case we try to replicate our previous approach to different circuit. Now we deal with a more complex component, a LCD panel. It is the LCD component found in the Arduino starter kit. It is a 16x2 LCD. As done with the previous test case, we try to implement our procedure first as Arduino sketch, then as Bash script and C/C++ program.

The hardware schema of the project follows. The necessary electronic components to build the circuit are:

- a Hitachi HD44780 LCD (or compatibles);

- a potentiometer;
- a 220Ohm resistor;
- some wires.

On the image is easy to see the various electronic links from the board to the LCD.

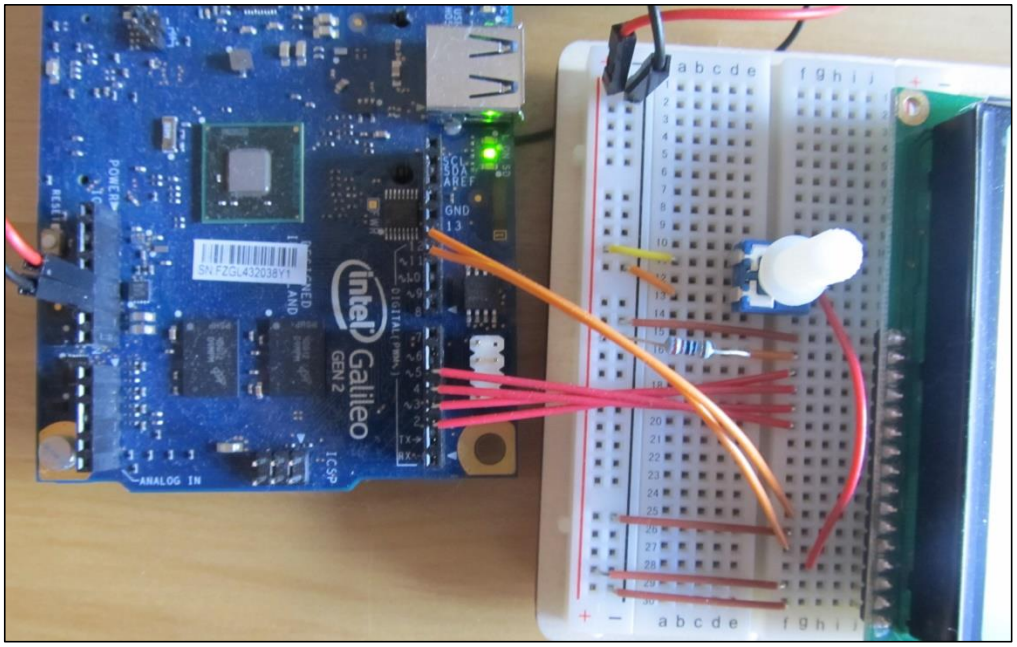

Figure 6-4. Circuit of the second test project

6.3.1 ARDUINO VERSION

The procedure presented in this section is the simple "HelloWorld" example of the Arduino IDE example library. It can be found under "File->Examples->LiquidCrystal->HelloWorld". The program prints on the LCD some simple strings. As a classic Arduino sketch, it can be compiled and upload to the board from the Arduino IDE.

```
/*
LiquidCrystal Library - Hello World
Demonstrates the use a 16x2 LCD display.  The LiquidCrystal
library works with all LCD displays that are compatible with the
Hitachi HD44780 driver. There are many of them out there, and you
can usually tell them by the 16-pin interface.
This sketch prints "Hello World!" to the LCD
and shows the time.
The circuit:
* LCD RS pin to digital pin 12
* LCD Enable pin to digital pin 11
* LCD D4 pin to digital pin 5
* LCD D5 pin to digital pin 4
```

```
* LCD D6 pin to digital pin 3
* LCD D7 pin to digital pin 2
* LCD R/W pin to ground
* 10K resistor:
* ends to +5V and ground
* wiper to LCD VO pin (pin 3)
Library originally added 18 Apr 2008 by David A. Mellis
library modified 5 Jul 2009 by Limor Fried (http://www.ladyada.net)
example added 9 Jul 2009 by Tom Igoe
modified 22 Nov 2010 by Tom Igoe
This example code is in the public domain.
http://www.arduino.cc/en/Tutorial/LiquidCrystal
*/
// include the library code:
#include <LiquidCrystal.h>
// initialize the library with the numbers of the interface pins
LiquidCrystal lcd(12, 11, 5, 4, 3, 2);

void setup() {
// set up the LCD's number of columns and rows:
lcd.begin(16, 2);
// Print a message to the LCD.
lcd.print("hello, world!");
}

void loop() {
// set the cursor to column 0, line 1
// (note: line 1 is the second row, since counting begins with 0):
lcd.setCursor(0, 1);
// print the number of seconds since reset:
lcd.print(millis()/1000);
}
```

At this point, as we did in section 6.2.2, we try to realize the Bash script equivalent to the above Arduino sketch.

6.3.2 BASH VERSION (FAILED)

Now, we try again to implement the Arduino sketch seen above as a Bash script which deals with the

hardware via reading and writing to the GPIO files linked to the GPIO board pins. But, differently from the previous test case, our LCD is linked to the Intel Galileo board through various pins. In theory, to deal with the LCD we could still roughly read and write values from these pins. But, for example to have a string printed to the LCD, we should know what is the device specification, how each pin works at electronic level and what value (0/1, LOW/HIGH) should have each interested pin to have a final "hello world" string on the display.

So, we found that when the hardware we use is more complex than a simple "one pin" device (a LED or a temperature sensor) we cannot actually translate the Arduino sketch code into the equivalent raw GPIO files stream, unless we decide to debug the Arduino API (the LCD Arduino class in our case) and find out what are the low level parts which realize the LCD class.

It is possible to make a reverse engineering attempt to find out these low level parts, but it is only for study or for the specific interest to know how the black box works under the Arduino API.

It is clear that this is not a work for a maker who wants to use the Intel Galileo board for prototyping his project.

6.3.3 C/C++ VERSION (FAILED)

As the direct I/O to the GPIO files is not a solution for our Linux environment implementation of the LCD project, we try to implement our procedure using the C Arduino API found in the official Intel Galileo guide at http://intel-software-academic-program.com/courses/diy/. So we expect to find a LiquidCrystal.h and LiquidCrystal.cpp into the "src" folder seen above. But, surprising enough, these file are not part of the C Arduino library. What is the matter?

Studying and exploiting a bit the Intel Galileo Arduino IDE C/C++ library (it is in the installation folder of the Arduino IDE on our working PC) we found out that the C Arduino API used in the simple temperature project is a raw test porting of this IDE C/C++ library. It is realized by Intel developers as a demonstrative example to illustrate how the Arduino sketch is actually a C/C++ program on the Intel Galileo, but Intel ported only a little part of the Arduino IDE API (only the part necessary to manage some "one pin" external hardware like the temperature sensor or the led). So it is clear that, if we want to implement our LCD project directly in C/C++, we have to port the Arduino API LCD class and use this ported code in our implementation. And this should must be done for every different external component we need to realize our projects.

Searching on the Internet we found out that an official complete and native C Arduino API does not exist so far for the Intel Galileo platform, so it is clear that a maker implementing his project cannot drive standard Arduino shields from native C/C++ application.

6.3.4 WHAT WE LEARNED?

Our LCD test case suggests some topics. First, we cannot drive complex hardware through raw I/O to GPIO files. Second, we do not have an official C Arduino API. These are both interesting matters which lead us to a strong conclusion: it is obvious that the main way to program the board as microcontroller remains the Arduino side of the board.

Initially, we supposed that the developer could totally replace the Arduino software layer thanks to the underlying Linux system, disposing the Arduino software stack and programming a native sketch directly in the C/C++/Java/Bash/Python or other programming languages which provide an Arduino API porting. But, after our tests we learned that this Arduino API porting does not exist.

At this point the natural question is: how could we exploit the Intel Galileo Linux system? This is the topic of the following Chapter 7. So far we learned that the main way to use the board as external hardware

microcontroller is the Arduino way. What we see in the following is that, as the Arduino features of the board provide the microcontroller skills to the Intel Galileo, the Linux system provides the Internet-Of-Things power of the platform. The Arduino sub-system gives to the programmer only basics network functionalities to the board, such as Ethernet and Wireless APIs which implement raw TCP and UDP communication channels. But, as we all know, a pure Linux system supplies an high level and complex environment which could run any kind of networking software.

As we see in Chapter 7, this is what we realize on the Linux side of Galileo: a network software stack for the remote controlling of the board.

7 Advanced Experiments

7.1 Introduction

At the beginning of Chapter 6, we supposed that the developer could totally replace the Arduino software layer thanks to the underlying Linux system, disposing the Arduino software stack and programming a native sketch directly in the C/C++/Java/Bash/Python or other programming languages which provide an Arduino API porting. But, after our tests we learned that this Arduino API porting does not exist. This is the main reason why the Intel Galileo Arduino software stack is necessary to the maker if he wants to program the board as hardware microcontroller.

In this chapter we finally find out what we can actually do with the Linux side of the board. We commit the microcontroller functions of our next project to the Arduino sub-system of our Intel Galileo board, and we implement the network features of the project on the Linux system. As we describe soon, at the end this results in a software project composed by two running processes: the Arduino sketch process (which, as we described previously, at running time is a standard Linux process) and the a Linux process which implements a control application to manage the Intel Galileo platform from a remote system. The interesting part of the next project is that, in some way, this two processes have to communicate one to the other. In the following sections we describe how.

As for "Chapter 6 – Basic Experiments", the source code files of the projects in this chapter can be found at the web page http://tinyurl.com/ndgwlsh .

7.2 Third Test Case: Advanced Temperature

Our third test case is an advanced version of the temperature project. As the official Intel Galileo guide suggests, we have a complete Linux system on our board. Also, the device is advertised as a new level device for the Internet-Of-Things. Thus, what we do in the following project is to test the Internet capabilities of our platform. But, as the Arduino side is the master when dealing with sensors and actuators, we suppose that the Linux sub-system is the master when exploiting the Internet features of the board. Let us prove our thesis.

The original Arduino boards support the Internet connectivity through on-board components or network extension shields as the Ethernet or the WIFI shields. This components are also supported by the original Arduino APIs.

The same is true for our Intel Galileo. We have an on-board Ethernet adapter and the possibility to expand the board with a miniPCI WIFI adapter. In both cases, the hardware is supported out of the box by the Intel Galileo Arduino APIs. But these APIs provide raw TCP/UDP transport layer connections.

What if we want to remotely control our board? One possibility is to implement an application layer protocol on top of the Ethernet or Wireless Arduino API to communicate over the Internet.

A second and best possibility is to exploit the Linux system to take advantage of his great power. In "Chapter 5 – The Linux Side" we described how to setup the Intel Galileo Linux system to have the Apache2 web server and the PHP5 interpreter running on the board. The time has come to take advantage of this high level networking tools.

As we said, the following project is an advanced version of the temperature test case. We could say that it is the "Internet-Of-Things" version of that project. We realize a remote controlled temperature platform. We put on our breadboard a temperature sensor, a red led, a potentiometer and an LCD. The electronic links are shown in the following picture. They are the same circuits seen in the two projects described in "Chapter 6 – Basic Experiments".

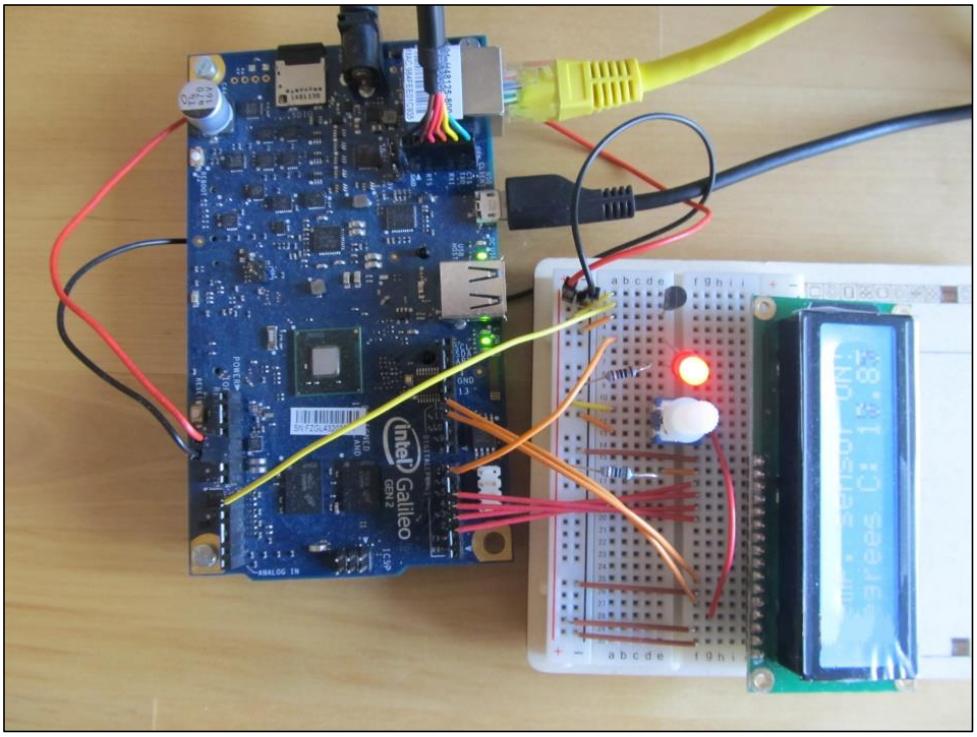

Figure 7-1. Circuit of the third test project

Our device senses the environment temperature and check if it is greater than a baseline temperature or not. If it is true, we switch on the red led. Each time the device check the temperature, it is printed on the LCD as Celsius degrees. The potentiometer controls the brightness of the panel. This part of the project is developed as Arduino sketch through the Arduino IDE and APIs in the classical Arduino way we showed more times in the previous sections.

The board is connected to the Internet and it is in the same LAN with our working PC (as explained in Chapter 5) thanks to an Ethernet connection. We develop the remote control layer as a dynamic web application provided by the Apache2 web server. The PHP language is used to implements the dynamic part of the application.

So, our project splits into two processes running on the Intel Galileo Linux system. The first is the process of the Arduino sketch which implements the I/O to the external devices (led, sensor, LCD). The second process is the web application managed by the Apache2 web server. This application provides to the remote user a simple web interface to read the current state of the system (current temperature and baseline value) and a form to input the a new value for the baseline temperature value, as shown in the following figure.

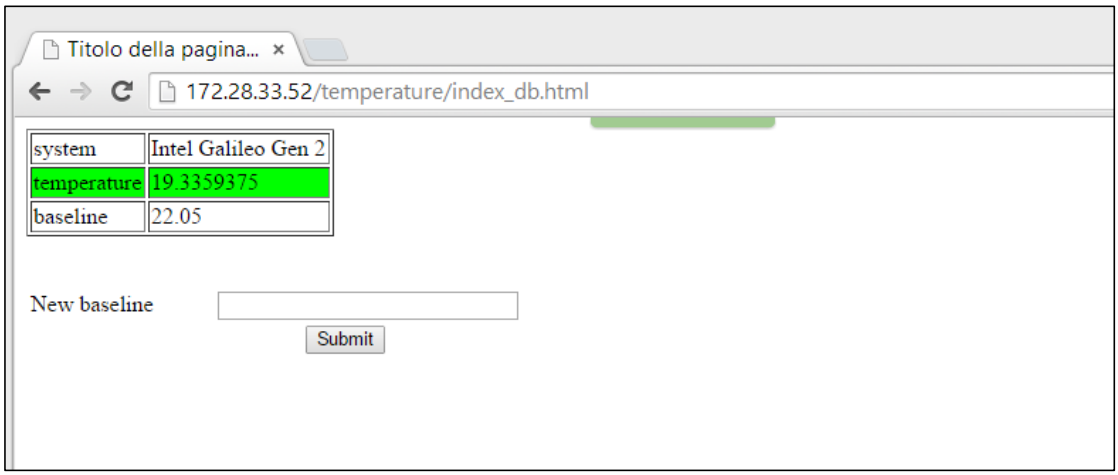

Figure 7-2. Web application UI

The web application we realized is implemented as a RESTful web application though the AJAX technology. The front end of the application is written with the HTML and JavaScript languages. The JavaScript language implements the client side network logic via the XMLHTTPResquest API. At the backend, the other endpoint of network communication is managed by two PHP scripts: the *"info.php"* and the *"baseline.php"* scripts. When called through an HTTP GET request by the frontend, the *"info.php"* script "queries" to the Arduino sketch process for the current temperature value sensed by the temperature sensor. Finally, the PHP script returns a JSON value representing this information to the caller. In the same way, the *"baseline.php"* script waits for HTTP requests from the remote web page. When the frontend sends a new baseline value, the script updates the Arduino sketch process with the new baseline value. After that, the Arduino sketch process will consider this new baseline value and will behave accordingly. The schema in picture 7.3 shows the project structure we just described.

At this point it is clear why we said that the two processes have to communicate in some way one to the other. To read the status of the system, the web application process (actually the PHP process) needs to communicate with the sketch process to know the current temperature sensed by the temperature sensor. In the same way, the sketch process needs to communicate with the web application process to know the value of the new baseline value sent in input by the remote user via the web interface described above.

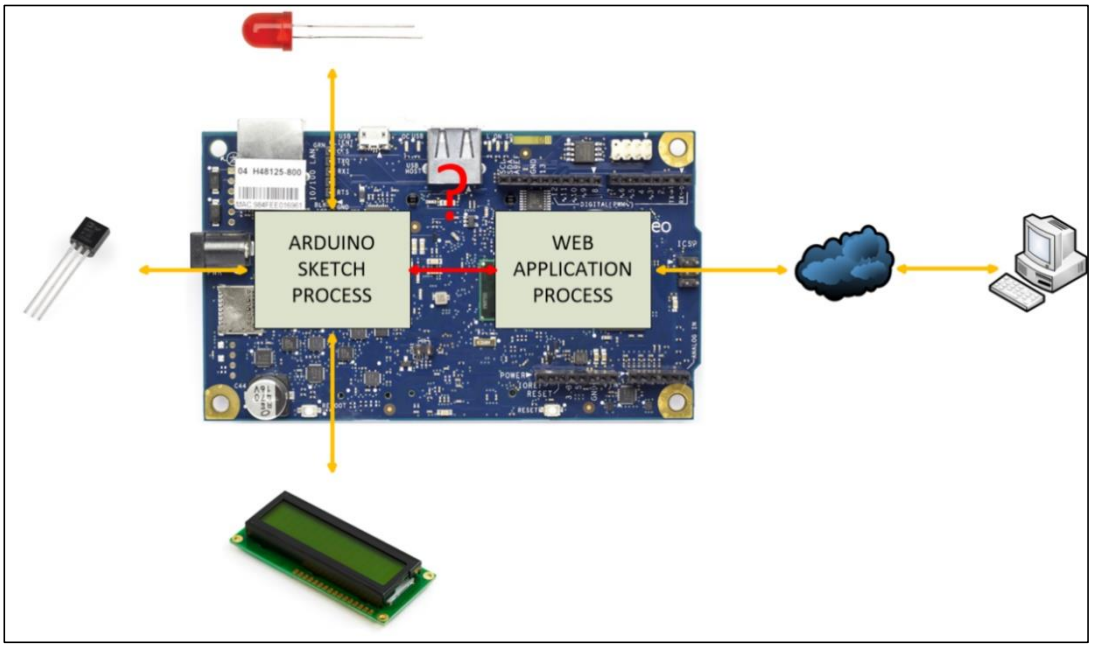

Figure 7-3. Schema of the Advanced Temperature project

There exists different ways to implement this Inter-Process Communication (IPC) mechanism. In the following section 7.2.1, we first provide an example of IPC through some files on the Linux FileSystem. Thus, this implementation consists of the Arduino and the Linux processes communicating by roughly reading and writing on some shared files.

Then, in section 7.2.2 we make the two processes to communicate through a relational DBMS on the Linux FileSystem. In a similar way to the shared files method, in this case the two processes share a relational schema and the inter-process communication is accomplished accessing some table into this schema. With respect to the shared files version, this is a big enhancement of our project, because with it we could easily manage complex structured data.

The previous are both example of shared persistent memory. Finally, in section 7.2.3 we describe a

methodology of how to make the Arduino and the Linux processes to communicate on RAM (volatile memory) realizing this communication link through the socket technology.

7.2.1 FILE IPC VERSION

As previously described, our first IPC implementation uses two files on the Linux FileSystem as communication links between the Arduino sketch process and the Linux process of our dynamic web application. The following code is our Arduino sketch implementation. As usually, it can be written, compiled and uploaded to the board from the Arduino IDE.

```
#include <LiquidCrystal.h>
// initialize the library with the numbers of the interface pins
LiquidCrystal lcd(12, 11, 5, 4, 3, 2);

const int sensorPin = A0;
float baseline = -1.0;
const int ledPin = 7;

void setup() {
Serial.begin(9600); // open serial port
pinMode(ledPin, OUTPUT);
digitalWrite(ledPin, LOW);

// set up the LCD's number of columns and rows:
lcd.begin(16, 2);
// Print a message to the LCD.
lcd.print("Temp. sensor ON!");
}

void loop() {
// read baseline from file
baseline = getBaseline();

Serial.print("baseline: ");
Serial.print(baseline);
Serial.print("\n");

// read temperature from sensor
float temperature = readTemperature();
```

```
if(temperature > baseline) {
digitalWrite(ledPin, HIGH);
}
else {
digitalWrite(ledPin, LOW);
}

// set the cursor to column 0, line 1
// (note: line 1 is the second row, since counting begins with 0):
lcd.setCursor(0, 1);
// print the number of seconds since reset:
lcd.print("degrees C: ");
lcd.print(temperature);

// write temperature to file
writeTemperature(temperature);

delay(1000);
}

float getBaseline() {
char output[5];
FILE *fp;
fp = popen("cat /usr/local/apache2/htdocs/temperature/system/baseline",  "r"); //

if (fp == NULL) { //
Serial.println("Couldn't run the curl command.");
return -1.0;
}
else {
fgets(output, sizeof(output), fp);
}
if (pclose(fp) != 0) { //
Serial.println("The cat command returned an error.");
return -1.0;
}

return atof(output);
```

```
}

void writeTemperature(float temperature) {

char _float_temp[5];
sprintf(_float_temp, "%2.2f", temperature);

char _prefix[] = "echo -n \"";
char _suffix[] = "\" > /usr/local/apache2/htdocs/temperature/system/temperature";

char _command[sizeof(_prefix) + sizeof(_float_temp) + sizeof(_suffix)];
int i = 0;
int j;
// the -1 is to cut the '/0' character at the end of _prefix
for(j=0; j<sizeof(_prefix) - 1; j++) {
_command[i] = _prefix[j];

i++;
}
for(j=0; j<sizeof(_float_temp); j++) {
_command[i] = _float_temp[j];
i++;
}
for(j=0; j<sizeof(_suffix); j++) {
_command[i] = _suffix[j];
i++;
}

Serial.print(_command);
Serial.print("\n");
//system("echo -n \"45\" > /usr/local/apache2/htdocs/temperature/system/temperature");
system(_command);

}

float readTemperature() {
int sensorVal = analogRead(sensorPin);
// convert the ADC reading to voltage
```

```
float voltage = (sensorVal / 1024.0) * 5.0;
// convert the voltage to temperature in degrees
float temperature = (voltage - 0.5) * 100;

Serial.print("Sensor value: ");
Serial.print(sensorVal);
Serial.print(", Volts: ");
Serial.print(voltage);

Serial.print(", degrees C: ");
Serial.print(temperature);
Serial.print("\n");

return temperature;
}
```

At each cycle of the loop the Arduino sketch process reads the baseline value from the "*/usr/local/apache2/htdocs/temperature/system/baseline*" file on the Linux FileSystem through the *"system()"* function. Then, the current temperature is read from the external sensor and some actions are taken in function of the baseline value (switch on/off the LED, print something on the LCD). Finally, the current temperature is stored into the "*/usr/local/apache2/htdocs/temperature/system/temperature*" file, once again thanks to the *"system()"* function. The *"system()"* function is a Intel Galileo extension to the Arduino API. It can be seen as an entry point to the Linux system from the Arduino side of the board. The function takes in input a string representing the command-line Linux command to be executed. Our Arduino sketch implementation uses this function twice to read and write the two files described above. As the function takes in input a single string, in the case of the writing command we have to build by hand a dynamic command string at each cycle. See the above code for details.

As previously said, in our project the Linux process which implements the IPC logic is the PHP process. Its code follows. This part is implemented into the *"info.php"* and the *"baseline.php"* files. As described, the web application is implemented as AJAX project, and this two PHP scripts implements the backend of the application. When called by the web client through an HTTP GET request, the *"info.php"* endpoint reads the temperature and the baseline values stored into the respective files seen above and finally returns a JSON value representing this informations to the caller. In the same way, the *"baseline.php"* script waits for HTTP requests. When the web client sends a new baseline value, the script updates the value stored into the baseline file on the Linux FileSystem. After that, the Arduino sketch process will read the new baseline value and will behave accordingly.

Info.php

```
<?php
$temperatureFile = fopen("../system/temperature", "r") or die("Unable to open file!");
```

```
$temperature = fread($temperatureFile, filesize("../system/temperature"));

fclose($temperatureFile);
$baselineFile = fopen("../system/baseline", "r") or die("Unable to open file!");
$baseline = fread($baselineFile, filesize("../system/baseline"));
fclose($baselineFile);

$response = array();
$response["system"] = "Intel Galileo Gen 2";
$response["temperature"] = $temperature;
$response["baseline"] = $baseline;
echo json_encode($response);
?>
```

Baseline.php

```
<?php
$baseline = $_GET['baseline'];
$file = fopen("../system/baseline", "w") or die("Unable to open file!");
fwrite($file, $baseline);
fclose($file);
?>
```

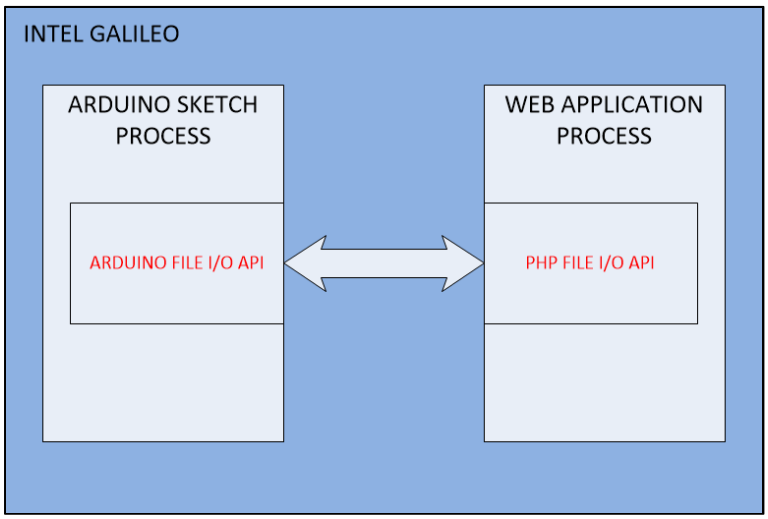

Figure 7-4. Schema of the File IPC version

In the next section we describe the evolution of the above approach: we make the Arduino and the PHP processes to communicate through a shared DB schema.

7.2.2 DBMS IPC VERSION

As seen above in the File IPC implementation, the Arduino sketch process can communicate with the underlying Linux system through the *"system()"* function. This function get in input a unique string. As in the previous example, if we need to run a static Linux command the function works very well, but if we need to pass to the function a dynamic command we have to construct the string representing the command by hand at each cycle. But, is this the unique way to expand the Arduino sketch functionalities? Could we expand the features of our sketch in some other manner? The response is positive.

As we learned that the Intel Galileo Arduino IDE generates a sketch which is actually a C/C++ executable Linux binary, we found out that the Arduino IDE we installed on our working PC embeds a C/C++ cross compiler and so we have available the standard C API to program our sketch. Thus, we can expand our sketch with standard C/C++ code (also divided in multiple files) to realize a final big Arduino sketch which could contains every kind of standard C/C++ source code.

This is the key feature in implementing our next project version.

In the DBMS IPC version, both the sketch process and the PHP process read and write temperature and baseline values into a SQLite DB file. While the PHP language includes the API to communicate with a SQLite DB, the sketch process does not have this capability as native. So, what we do is expand our sketch with the SQLite_v3 API source code. After that, our sketch can call this APIs to communicate with any SQLite DB file.

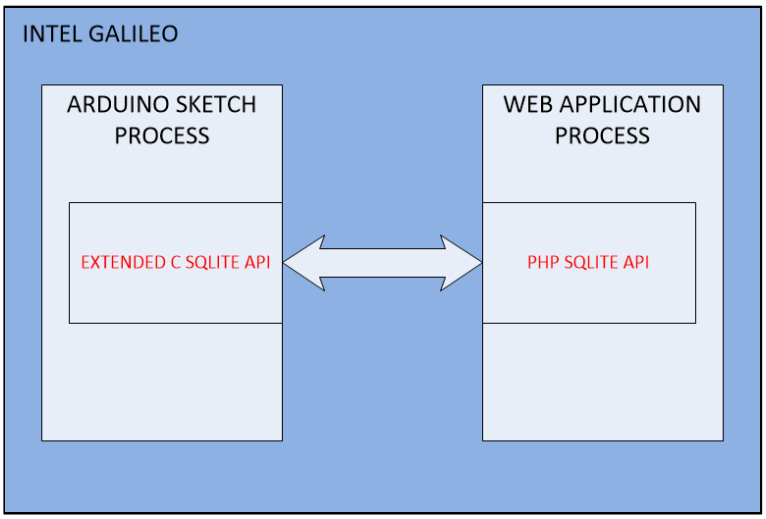

Figure 7-5. Schema of the DBMS IPC version

As the following picture shows, the "sqlite3.c" and the "sqlite.h" files are included in the project as external files. This is accomplished by the "Sketch->Add File.." from the Arduino IDE menu.

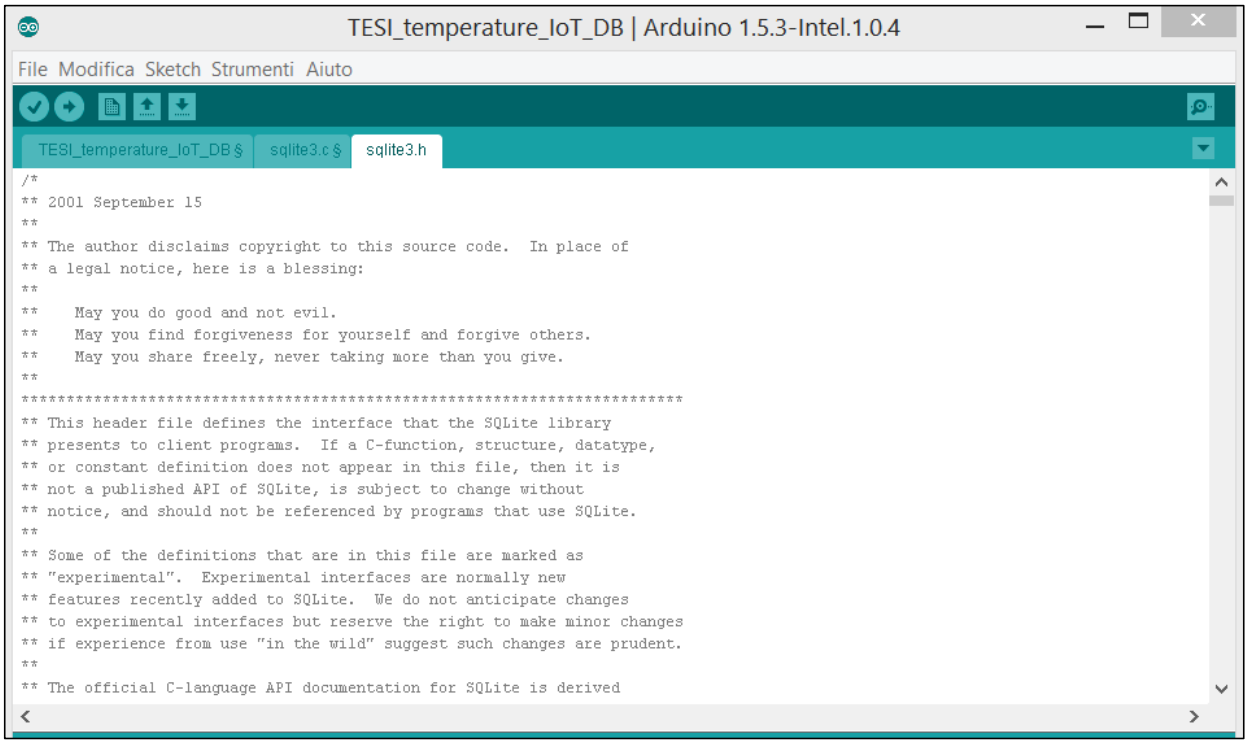

Figure 7-6. The sketch expanded with the SQLite source code

When the SQLite code is available, our sketch is composed of Wiring library language lines and of standard C SQLite API function calls. The new Arduino sketch code follows (we present here only the SQLite coded function calls, the *"setup"* and *"loop"* functions are similar to the previous ones. As usually, the entire code is attached to this document).

```
int setupDB() {

sqlite3 *db;
char *err_msg = 0;

int rc = sqlite3_open(dbFile, &db);

if (rc != SQLITE_OK) {

Serial.print("Cannot open database: ");
Serial.print(sqlite3_errmsg(db));
Serial.print("\n");

sqlite3_close(db);

return 1;
```

```
}

char *sql = "CREATE TABLE Baseline(Id INT, Value REAL);"
"INSERT INTO Baseline VALUES(0, 20);"
"CREATE TABLE Temperature(Id INT, Value REAL);"
"INSERT INTO Temperature VALUES(0, -1.0);";

rc = sqlite3_exec(db, sql, 0, 0, &err_msg);

if (rc != SQLITE_OK ) {

Serial.print("SQL error: ");
Serial.print(err_msg);
Serial.print("\n");

sqlite3_free(err_msg);
sqlite3_close(db);

return 1;
}

sqlite3_close(db);

}

int readBaselineFromDB() {

sqlite3 *db;
char *err_msg = 0;

int rc = sqlite3_open(dbFile, &db);

if (rc != SQLITE_OK) {

Serial.print("Cannot open database: ");
Serial.print(sqlite3_errmsg(db));
Serial.print("\n");
sqlite3_close(db);
```

```
return 1;
}

char *sql = "SELECT Value FROM Baseline WHERE Id=0;";

rc = sqlite3_exec(db, sql, callback, 0, &err_msg);

if (rc != SQLITE_OK ) {

Serial.print("Failed to select data. SQL error: ");
Serial.print(err_msg);
Serial.print("\n");

sqlite3_free(err_msg);
sqlite3_close(db);

return 1;
}

sqlite3_close(db);

return 0;
}

int callback(void *NotUsed, int argc, char **argv, char **azColName) {

NotUsed = 0;

/*for (int i = 0; i < argc; i++) {

//printf("%s = %s\n", azColName[i], argv[i] ? argv[i] : "NULL");
baseline = atof(argv[i]);
}*/

baseline = atof(argv[0]);
Serial.print("New baseline: ");
Serial.print(atof(argv[0]));
```

```
Serial.print("\n");

return 0;
}


int writeTemperatureToDB(float temperature) {

sqlite3 *db;
char *err_msg = 0;
sqlite3_stmt *res;

int rc = sqlite3_open(dbFile, &db);

if (rc != SQLITE_OK) {

/*Serial.print("Cannot open database: ");
Serial.print(sqlite3_errmsg(db));
Serial.print("\n");*/
sqlite3_close(db);

return 1;
}

char *sql = "UPDATE Temperature SET Value = ? WHERE Id=0;";

rc = sqlite3_prepare_v2(db, sql, -1, &res, 0);

if (rc == SQLITE_OK) {

sqlite3_bind_double(res, 1, temperature);
}
else {

Serial.print("Failed to execute statement: ");
Serial.print(sqlite3_errmsg(db));
Serial.print("\n");
}
```

```
int step = sqlite3_step(res);

if (step == SQLITE_ROW) {

/*Serial.print(sqlite3_column_text(res, 0));
Serial.print(": ");
Serial.print(sqlite3_column_text(res, 1));
Serial.print("\n");*/

}

sqlite3_finalize(res);
sqlite3_close(db);

}
```

In the *"setupDB()"* function a schema composed of two tables is created. These tables are the *"baseline"* and the *"temperature"* tables. Each table contains a single raw with stored the baseline or the current temperature values. The *"readBaselineFromDB()"* and the *"writeTemperatureToDB()"* functions read and update these values at each cycle.

In the same way, the *"info.php"* and *"baseline.php"* scripts read and write to the SQLite DB. The code follows.

Info.php

```
<?php
$temperature = -1;
$baseline = -1;

try {
// write baseline to DB
$dbPath = "/tmp/sketch_db";
$db = new SQLite3($dbPath);

$result = $db->querySingle("SELECT Value FROM Temperature WHERE Id=0");
$temperature = $result;
$result = $db->querySingle("SELECT Value FROM Baseline WHERE Id=0");
$baseline = $result;
```

```
$db->close();
} catch (Exception $e) {
echo 'Caught exception: ',  $e->getMessage(), "\n";
}
$response = array();
$response["system"]  = "Intel Galileo Gen 2";
$response["temperature"]  = $temperature;
$response["baseline"]  = $baseline;
echo json_encode($response);
?>
```

Baseline.php

```
<?php
$baseline = $_GET['baseline'];
$file = fopen("../system/baseline", "w") or die("Unable to open file!");
fwrite($file, $baseline);
fclose($file);
?>
```

How we saw, our final sketch includes Arduino API code and the standard C/C++ code of the SQLite3 source files. As said, the Arduino IDE installed on our workink PC embeds a x86 cross-compiler which builds the software for our Intel Galielo. So, compiling our sketch, also the external SQLite C/C++ code is compiled by the same cross-compiler and thus it is compatible to run on the Intel Galileo platform.

At the end of our development everything works fine, but our first attempts to compile the SQLite C/C++ code on the Arduino IDE threw some errors. Details about this follows.

If we try to compile the "sqlite3.c" source file as it is, we get the following error:

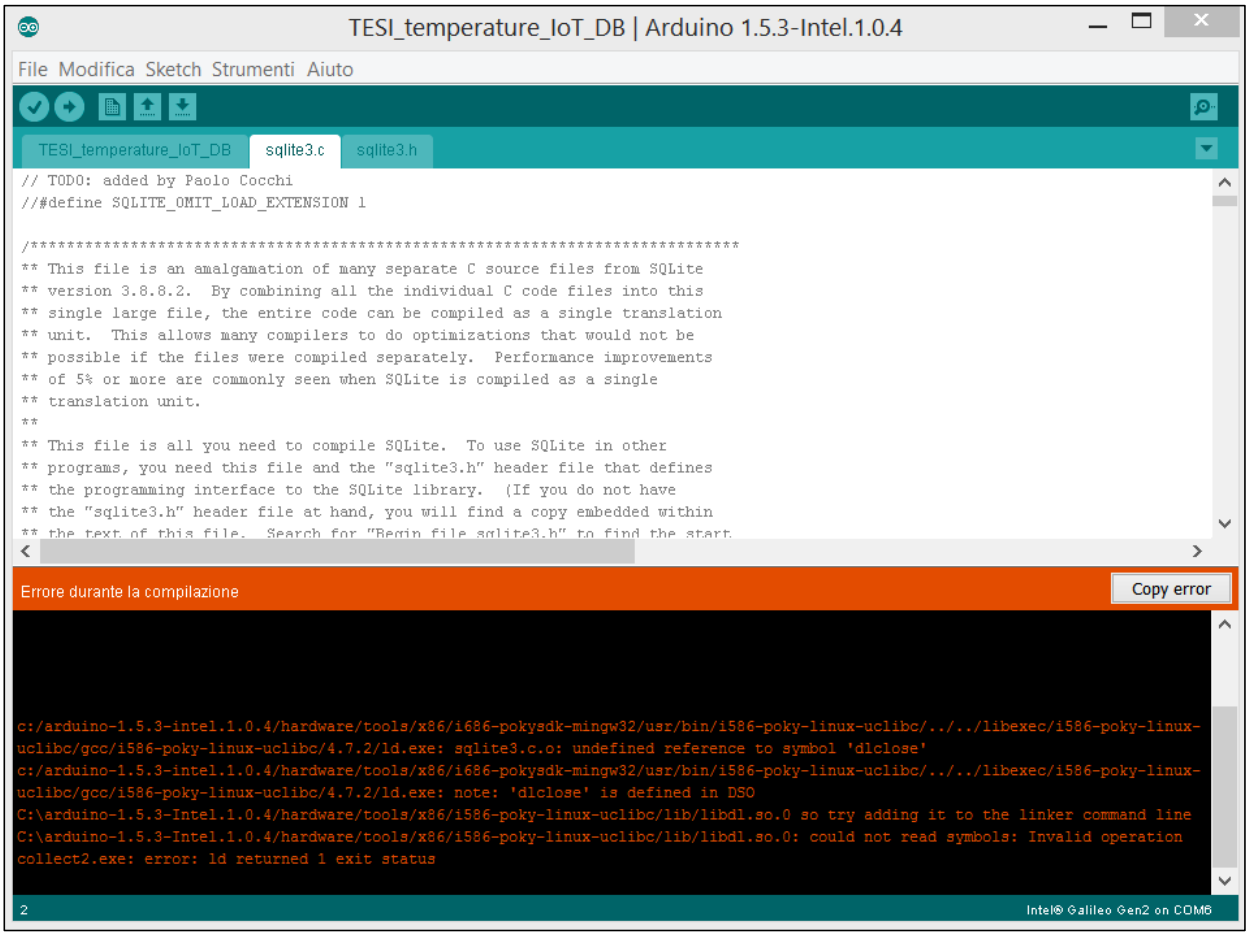

Figure 7-7. The error compiling the raw SQLite source code

The *"dlclose"* function is part of the LIBDL library, an utility for dynamic linking on the Linux systems. When compiling, the compiler searches for this library but it is unable to *"adding it to the linker command line"*, as shown by the error message in figure 6.7. As visible on the left top end of the previous image, our final solution is to disable dynamic linking by adding the code line *"#define SQLITE_OMIT_LOAD_EXTENSION 1"*.

To solve this issue, the first thing we tried was to manually set the linker option in the Arduino IDE to add something like "*-ldl*" (i.e, we tried to manually add to the linker the LIBDL library). But, this was not possible because the Arduino IDE does not provide a way to set options on the linker launch command.

Finally, we made an analysis of the SQLite source code to find out what could be a method to disable dynamic linking at compilation time. At the end of our analysis, we found out that all the dynamic linking feature are disabled if the "*SQLITE_OMIT_LOAD_EXTENSION*" macro is set to 1.

For the second time after the LCD project, we had to do some reverse engineer work. Differently from the LCD case, this time the solution came out easily enough. But it is clear that it is not a job for a maker who has experience in the Arduino world and who wants to prototype a project with the Intel Galileo board. Going forward with our experimentation, we find that working on the Intel Galileo board is straightforward when we use the board as standard Arduino device but less immediate when we try to enhance the level of difficulty on the side of the Linux system.

In section 7.2.1 and 7.2.2 we described two methodologies to realize a communication between the Arduino sketch process and the web application Linux process. As we described, both the FILE IPC and the DBMS IPC versions share something which resides on persistent memory. It is clear that it could be a big enhancement for the general system performance if we could implement the inter-process communication on primary memory. This is what we do in the following section 7.2.3.

7.2.3 SOCKET VERSION

The two previous project implementations used a persistent way of inter-process communication. In the next example we try to realize a volatile (i.e., in RAM) IPC with the SOCKET technology. As we know, sockets are the endpoints of a network connection. In the set of IPC types, sockets are objects that provide a communication between processes running on different machines. In our case, processes are on the same machine (our Intel Galileo). This is the special case when both processes bind the *"localhost"* address.

As seen above, our project splits into the Arduino sketch and the Apache/PHP processes. While for the Arduino process we have all the instruments to implement our procedure, this is not the case for the PHP process. In fact, the default installation of the PHP interpreter does not include the PHP SOCKETS extension and, as we experimented in Chapter 5 during the Linux environment setup, we were unable to make and install an extended version of PHP because of problems linked to the Intel Galileo hardware specification (see Chapter 5 for details).

So, to test our sockets IPC, we get rid for a while of the Apache/PHP process and we make a test in which the two actors are the Arduino sketch process and a C program process. This C program process reads the system status and set a new baseline value in the same manner as our HTML/JS/PHP web application. As the reader knows, on the Apache web server the Common Gateway Interface for dynamic web content can be implemented in different ways. One is to implement the backend as PHP script, or Perl script language. Another option is to write C programs which are called by the Apache web server. So, dealing with the C programming language in place of the PHP PL is not a restriction.

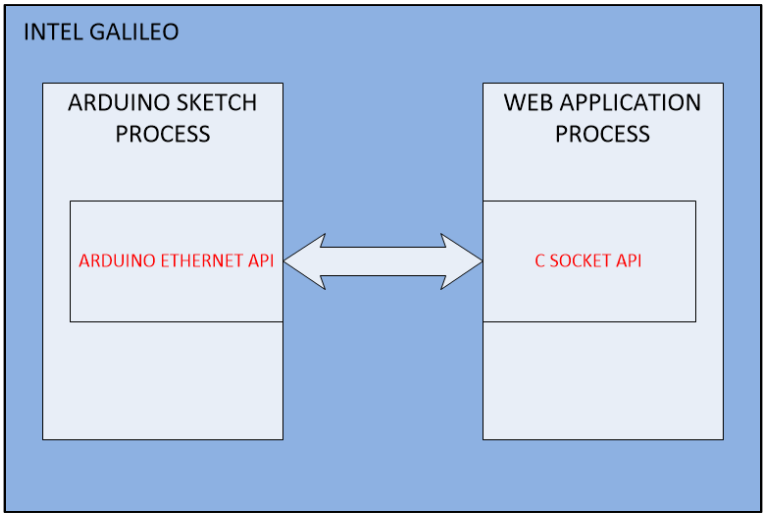

Figure 7-8. Schema of the Socket IPC version

For first, we see the Arduino sketch code. As we have all the Arduino APIs to use, we find that the simplest way to make our implementation is to use the Ethernet API of the Wiring library. Thanks to the Ethernet API, all sockets details are hidden, and the developer deals with high level functions like *"read()"* and *"print()"* to receive and send data to other endpoint of connection. Our sketch implementation follows.

```
#include <LiquidCrystal.h>
#include <SPI.h>
#include <Ethernet.h>

// initialize the library with the numbers of the interface pins
LiquidCrystal lcd(12, 11, 5, 4, 3, 2);


const int sensorPin = A0;
float baseline = 20.0;
float temperature = -1.0;
const int ledPin = 7;

// Enter a MAC address and IP address for your controller below.
// The IP address will be dependent on your local network:
byte mac[] = {
0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED };;
IPAddress ip(127, 0, 0, 1);

// Initialize the Ethernet server library
// with the IP address and port you want to use
// (port 80 is default for HTTP):
EthernetServer server(10000);

void setup() {

// Open serial communications and wait for port to open:
Serial.begin(9600);
while (!Serial) {
; // wait for serial port to connect. Needed for Leonardo only
}

pinMode(ledPin, OUTPUT);
digitalWrite(ledPin, LOW);

// set up the LCD's number of columns and rows:
lcd.begin(16, 2);
```

```
// Print a message to the LCD.
lcd.print("Temp. sensor ON!");

// start the Ethernet connection and the server:
Ethernet.begin(mac, ip);
server.begin();
Serial.print("server is at ");
Serial.println(Ethernet.localIP());

}

void loop() {

// manage IPC for first
// it update baseline and/or send to a calling process the status info
ipc();

// print current baselune
Serial.print("baseline: ");
Serial.print(baseline);
Serial.print("\n");

temperature = readTemperature();

if(temperature > baseline) {
digitalWrite(ledPin, HIGH);
}
else {
digitalWrite(ledPin, LOW);

}

// set the cursor to column 0, line 1
// (note: line 1 is the second row, since counting begins with 0):
lcd.setCursor(0, 1);
// print the number of seconds since reset:
lcd.print("degrees C: ");
lcd.print(temperature);
```

```
delay(1000);

}

int ipc() {

Serial.println(".");

// listen for incoming clients
EthernetClient client = server.available();
if (client) {
Serial.println("new client");

if (client.connected()) {

Serial.println("client connected");

if (client.available()) {

Serial.println("client available");

char request[256];
int i = 0;
char c;
while(((c = client.read()) != -1) && (c != '%') && (i < 256)) {
request[i] = c;
i++;
}
request[i] = '\0';

Serial.print("request: ");
Serial.println(request);

if(strncmp(request, "GET_INFO", 8) == 0) {
client.print("received GET_INFO: [baseline=");
client.print(baseline);
client.print(";temperature=");
```

```
client.print(temperature);
client.print("]");
client.print('\0');
}
else if(strncmp(request, "SET_BASELINE", 12) == 0) {
client.print("received SET_BASELINE");
client.print('\0');

if (client.available()) {
char request[256];
int i = 0;
char c;
while(((c = client.read()) != -1) && (c != '%') && (i < 256)) {

request[i] = c;
i++;
}
request[i] = '\0';

Serial.print("value: ");
Serial.println(request);


baseline = atof(request);
}

}
else {
client.println("command not found");
}

}

}
// give the web browser time to receive the data
delay(1);
// close the connection:
client.stop();
```

```
Serial.println("client disonnected");
}

}


float readTemperature() {
int sensorVal = analogRead(sensorPin);
// convert the ADC reading to voltage
float voltage = (sensorVal / 1024.0) * 5.0;
// convert the voltage to temperature in degrees
float temperature = (voltage - 0.5) * 100;

Serial.print("Sensor value: ");
Serial.print(sensorVal);
Serial.print(", Volts: ");
Serial.print(voltage);
Serial.print(", degrees C: ");
Serial.print(temperature);
Serial.print("\n");

return temperature;
}
```

The *"ipc()"* function above implements the network behaviour of the process. The code is very easy to read and understand, as all the Arduino code in the most cases. The Arduino sketch implements the server side of the connection, and as server it listens for external clients to connect.

An interesting aspect of the code above is the second line of the *"ipc()"* function:

EthernetClient client = server.available();

This line is to listen for incoming connection, so we could expect that it is an equivalent of the *"accept()"* function of the standard C socket API. But there is a big difference in the *"available()"* function: it is NON-BLOCKING. As we saw in the DBMS IPC program, we can use all the standard C API in our sketch. So, we could also implement the socket part of the program with the standard C socket API. But, as the *"accept()"* function is blocking, we should implement a multithreading program to correctly manage the socket IPC. In fact, once we enter the *"loop()"* function of a sketch, a blocking call blocks the entire sketch.

This consideration reminds to us to keep in mind that we are still programming an Arduino sketch when we use the C programming language and API into it. An Arduino sketch usually contains code to implement reading from sensor and writing to actuators. If we code a blocking line in the *"loop()"* function of our sketch, also all the I/O from the external sensors will block.

The next code is the C program process. It is a classical C socket client implementation.

```
#include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <arpa/inet.h>

int main(int argc , char *argv[]) {

if(argc < 2) {

printf("usage: main <command> [<input_value>] \n");
return -1;
}

int sock;
struct sockaddr_in server;
char* message;
char server_reply[1024];

//Create socket
sock = socket(AF_INET , SOCK_STREAM , 0);
if (sock == -1) {
printf("Could not create socket");
}
puts("Socket created");


server.sin_addr.s_addr = inet_addr("127.0.0.1");
server.sin_family = AF_INET;
server.sin_port = htons(10000);

//Connect to remote server
if (connect(sock , (struct sockaddr *)&server , sizeof(server)) < 0) {
perror("connect failed. Error");
return -1;
}
```

```
puts("Connected\n");

message = (char*)malloc(strlen(argv[1]));
strcpy(message, argv[1]);

if(strcmp(message, "GET_INFO") == 0) {

strcat(message, "%");

printf("sending \'%s\' \n", message);

if( send(sock , message , strlen(message) , 0) < 0) {
puts("Send failed");
goto exit;
}

//Receive a reply from the server
if( recv(sock , server_reply , 1024 , 0) < 0) {
puts("recv failed");

goto exit;
}

puts("Server reply :");
puts(server_reply);
}
else if(strcmp(message, "SET_BASELINE") == 0) {

if(argc < 3) {
printf("usage: main <command> [<input_value>] \n");
goto exit;
}


strcat(message, "%");
printf("sending \'%s\' \n", message);
if( send(sock , message , strlen(message) , 0) < 0) {
```

```
puts("Send failed");
goto exit;
}

char* value = (char*)malloc(strlen(argv[2]));
strcpy(value, argv[2]);
strcpy(message, value);
strcat(message, "%");
printf("sending \'%s\' \n", message);
if( send(sock , message , strlen(message) , 0) < 0) {
puts("Send failed");
goto exit;

}

//Receive a reply from the server
if( recv(sock , server_reply , 1024 , 0) < 0) {
puts("recv failed");
goto exit;
}

puts("Server reply :");
puts(server_reply);
}
else {
puts("command not found");
}

exit:
close(sock);
return 0;

}
```

The program can be launched from the command-line on Putty as a standard Linux executable. It takes as input the request string and a numeric parameter if necessary. We created a simple communication protocol which consists of only two possible requests: the GET_INFO request and the SET_BASELINE request. When the C program process sends the GET_INFO request to the Arduino sketch process, the Arduino sketch answers with system status info. The SET_BASELINE request expects the new baseline value as input parameter. When the C program process sends the SET_BASELINE request, theArduino sketch process saves the new baseline value to work with. An example of execution is shown in the following picture:

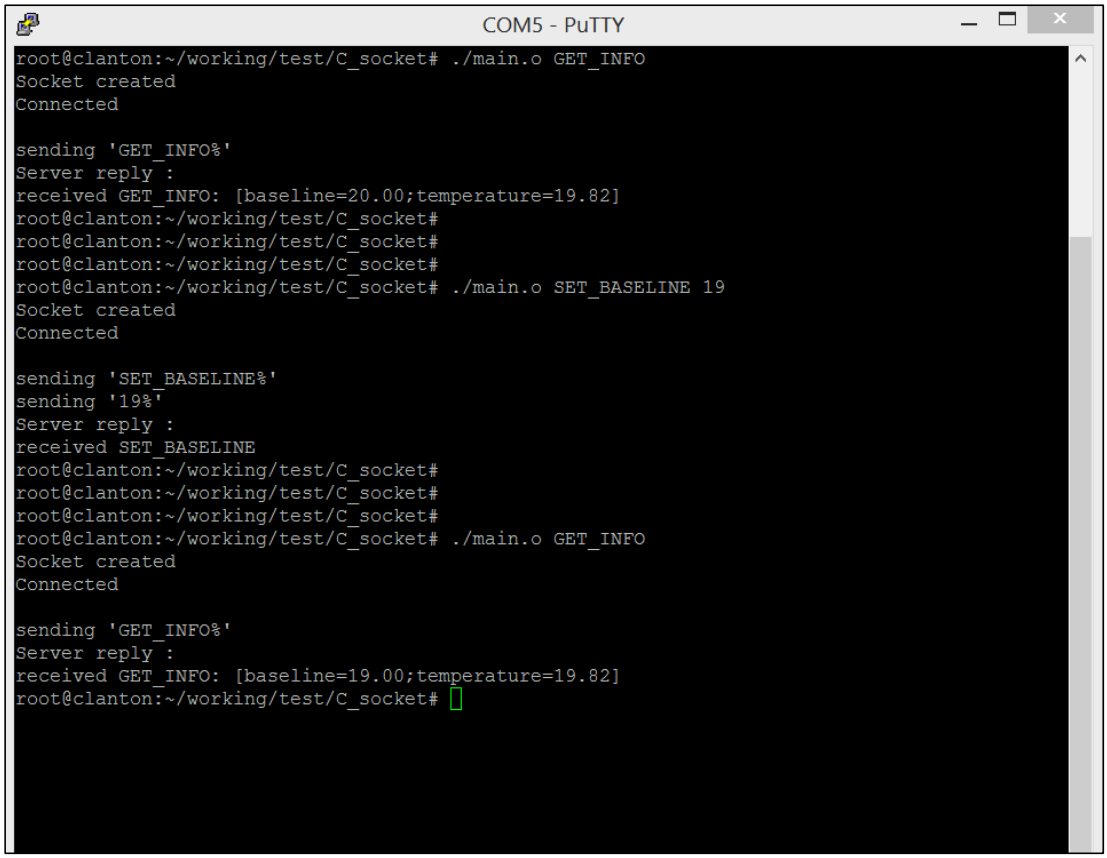

Figure 7-9. A run example of the Linux process implementing SOCKET IPC

As we described, for our Arduino sketch implementation we used the Arduino Wiring Ethernet API. As shown, this is the main and simplest way to follow because a pure standard C implementation is less obvious to realize. But, while with the standard C socket API we could implement a socket as LOCAL, using the Arduino Ethernet library this is not possible. A LOCAL socket is a connection endpoint visible only on the local machine. A local connection between sockets is an optimized version of the general sockets communication for local IPC. This would be the best architectural way to implement a local IPC with sockets. But, as the Arduino Ethernet library is design for network communication, we can only implement a communication between two standard sockets both bindings on the *"localhost"* address (i.e., 127.0.0.1).

In our last IPC implementation we used sockets as technology to realize a "in RAM" IPC between the Arduino sketch and another process running on the Intel Galileo Linux system. In the same manner we could use the standard C socket API to implement the Arduino side of the IPC, all the other standard Linux IPC mechanisms are available to use. So, shared memory, mapped memory, pipes and FIFOs are all technologies

we could employ to build a communication system between the Arduino sketch and some other C/C++ process running on the Intel Galileo platform.

8 Conclusion

8.1 Final Remarks

We began Chapter 1 of this paper with a brief description of what Internet-Of-Things means, dealing with the wide application area of the IoT concept.

In Chapter 2 and Chapter 3 we analysed the physical devices which realize the Internet-Of-Things in the real world and introduced the Intel Galileo platform on which we focus in the present document. Our review on the IoT devices panorama made us to have a more detail understanding about the real state of art of the Internet-Of-Things world at the beginning of 2015. As we saw, even if the IoT theory includes applications from nano heartbeat sensor to factory machine control, at the time we are writing this paper the IoT devices come out in the form microcontroller and mini PC. Thus, the most advanced use of the current technology is realized as implementation of Home Automation and Smart Home projects. As we described in "Chapter 2 – The Board World", some micro device like the Arduino LilyPad exists, but the big part of the IoT platform set is composed of "smartphone size" boards like those we described in Chapter 2. The Intel Galileo platform on which we focus is part of this consideration.

In the following chapters from 4 to 7, we exploit the Intel Galileo board and provide a methodology to work with it, with the aim to catch the total power of both the Arduino features and the Linux ones of the Galileo platform.

Our experimentations helped us to actually understand what is the architectural design of the Intel Galileo board and how it should be programmed. We learned that the Arduino side of the board is designed to manage the microcontroller functions of the platform, while the Linux side system running on it is the perfect technological solution to implement the remote control layer of our projects.

We believe that Intel chose to realize a Arduino-compatible board to take advantage of the wide diffusion and success of the Arduino platform. The Arduino maker community is the most widespread maker community in the microcontroller panorama. As we described, almost all the Arduino shields are supported out of the box by the Intel Galileo board.

At the end of our analysis, we realize that actually the Intel Galileo is first of all an Arduino microcontroller. This is clear by the first projects we developed in "Chapter 6 – Basic Experimets", where we tried to dispose the Arduino software stack in favour of a complete Linux development environment. As we saw, our experiments proved that the Arduino part of the system is primary to realize our projects, and we cannot get rid of it.

Intel provides a test Eclipse environment set up with a Galileo cross-compiler which should replace the Arduino IDE and all the software Arduino stack of the board. This Eclipse environment should provide a C/C++ IDE on which write, compile and upload programs on the Intel Galileo Linux FIleSystem, and a basic version of Arduino API porting to write C/C++ code with the aim to drive the Arduino-compatible extension hardware. The following figure 8.1 shows a screenshot of the Eclipse IDE for Intel Galileo.

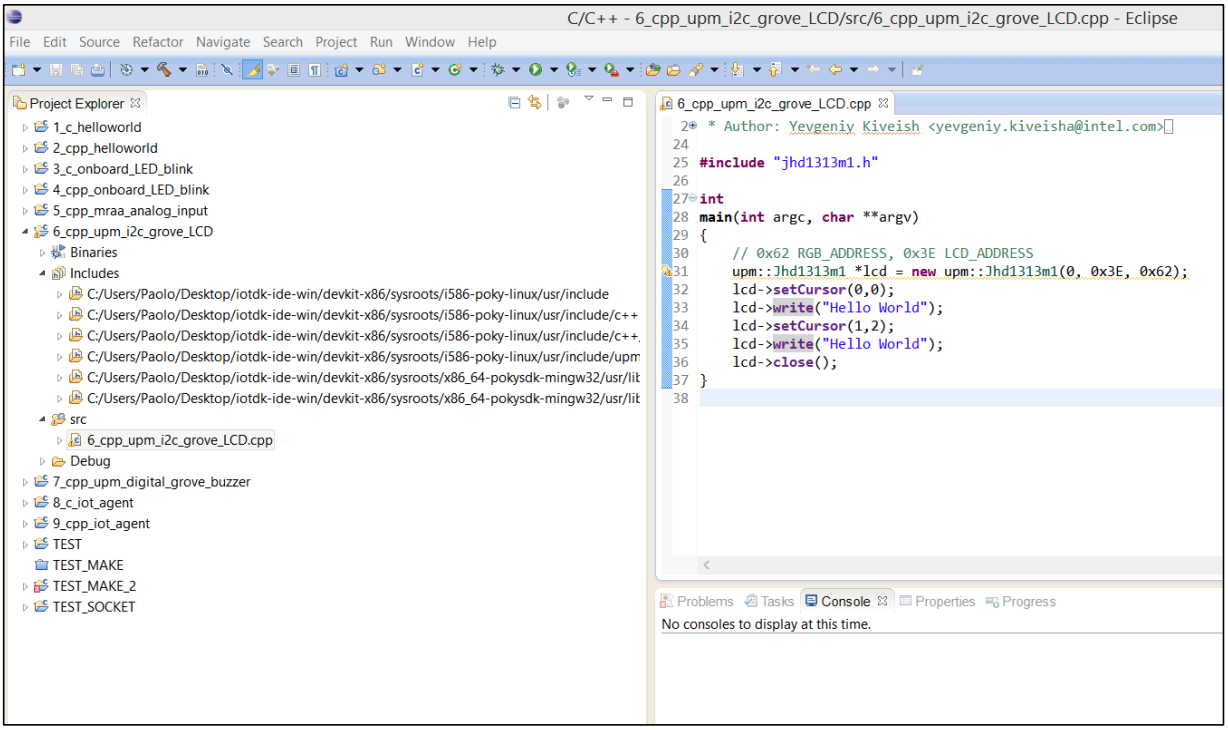

Figure 8-1. The Eclipse IDE for Intel Galileo

We tested the IDE, but it is at developing state and a some code, like the LCD code shown in fugure 8.1, runs with errors on the board. Furthermore, The Arduino API porting exposes an interface which differs from the standard Arduino API, thus deleting the Arduino software compatibility of the platform. Once again, the Arduino side of the board is the main way to program the microcontroller behaviour of the Intel Galileo system.

What we think is that the Linux side of the platform is the feature which actually pushes the board into the Internet-Of-Things world. This characteristic transforms the board from a simple hardware microcontroller to an advanced IoT device for implementing every kind of networking features. We proved our thesis in Chapter 7, where we implemented on the board a remote control software stack in the form of web application.

It is clear that the projects we chose to realize are very basic, but they were the perfect test cases to show all the interesting feature of the Intel Galileo board. In place of our temperature sensor we are now ready to use every kind of complex Arduino-compatible extension hardware. In the same way, our simple control web interface could become a rich web application.

As the complexity of the projects increases, the Internet-Of-Things becomes the new edge of designing in many large scale application areas.

One of these is the Environmental Monitoring. Environmental monitoring applications of the Internet-Of-Things typically use sensors to help in environmental protection by monitoring air or water quality, atmospheric or soil conditions, and could also include applications like the monitoring of wildlife movements and habitats. The IoT devices connected to the Internet actually have limited resources. Thus, they are usually connected to other systems like earthquake or tsunami early-warning systems which can also be used by emergency services to provide more effective aid. IoT devices in this application typically span a large geographic area and can also be mobile.

Another large scale application area is the Infrastructure Management. Monitoring and controlling of urban and rural infrastructures like wind-farms, railway tracks and bridges is a key application of the IoT. The Internet-Of-Things infrastructure can be used for monitoring any events or changes in structural conditions that may affect safety and increase risk. It can also be utilized for planning activities of repairing and maintenance in an efficient manner, by coordinating tasks between the wide range of service provided and the users of these facilities. IoT devices can also be used to control critical infrastructure like bridges to allow access to ships. The use of IoT devices for infrastructure monitoring is supposed to improve incident management and forecasting. Also, emergency response coordination, quality of service and reduce costs of operation in all infrastructure related areas are the aims of the Internet-Of-Things application on the Infrastructure area.

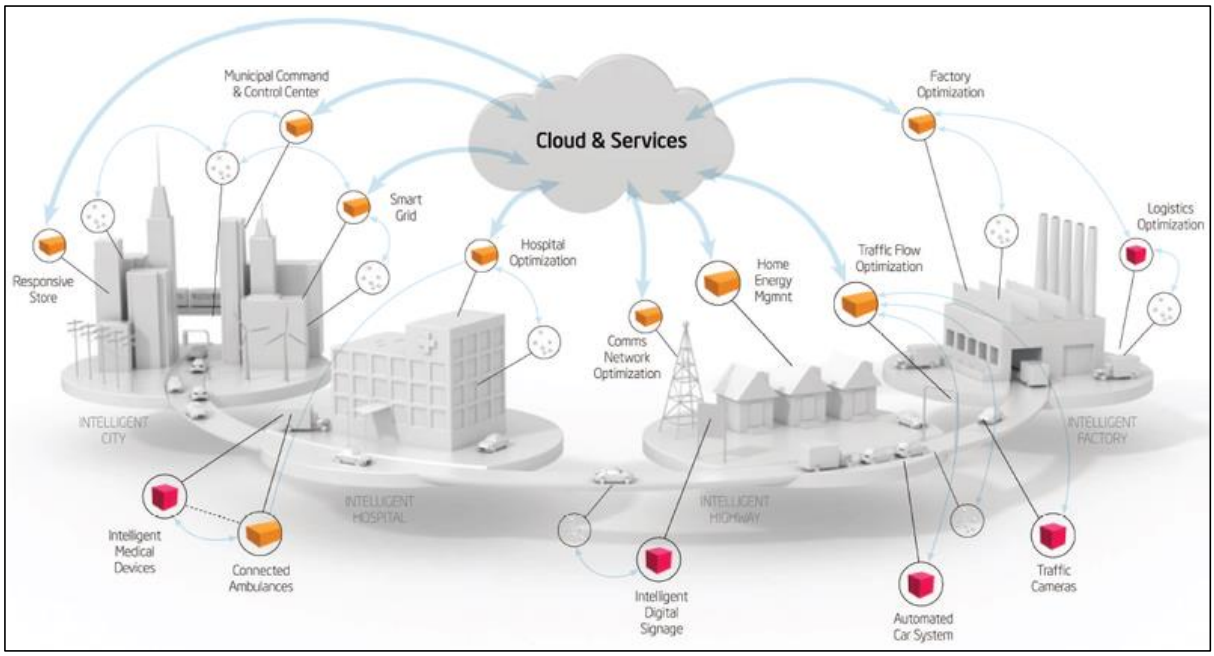

Figure 8-2. Infrastructure Management IoT

Linked to the infrastructure management there is the special case of Transportation. The IoT can help in the integration of communications, control and information processing through various transportation systems. Application of the IoT extends to all aspects of transportation systems, i.e. infrastructures, vehicles, and drivers (or users). Dynamic interaction between these components of a transport system enables smart parking, inter and intra vehicles communication, vehicle control, electronic toll collection systems, smart traffic control, fleet and logistical management, safety and road assistance.

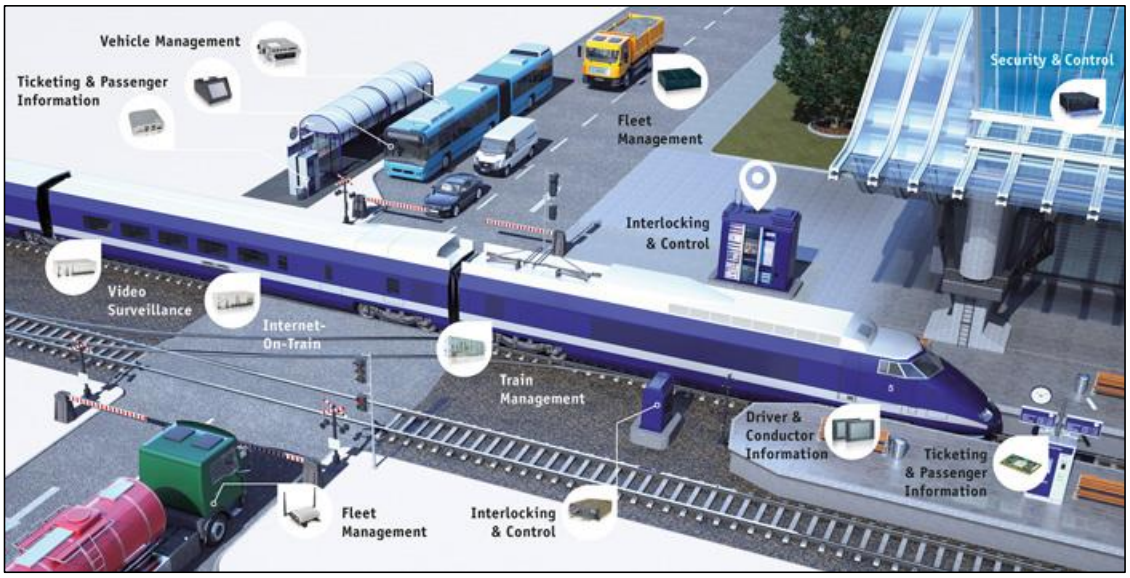

Figure 8-3. Trasportation IoT

Another important sector of application for the Internet-Of-Things concept is the Energy Management. Integration of Internet connected sensor and actuator systems is aimed to optimize energy consumption on a large scale. It is expected that IoT devices will be integrated into all forms of energy consuming devices (televisions, bulbs, power supplies, switches, etc.) and be able to communicate with the utility supply provider in order to effectively balance and energy usage and power generation. The Internet-Of-Things is especially relevant to the Smart Grid application area since it provides systems to gather data and act accordingly on energy and power-related information in an automated manner with the aim to improve the efficiency, reliability, economics, and sustainability of the production and distribution of electricity. Using the so called Advanced Metering Infrastructure (AMI), electric utility devices are connected to the Internet backbone and can not only collect data from end-users, but also manage distribution devices like transformers in an automatic manner.

Such devices would also offer the ability for users to remotely control their devices, or centrally manage them via a cloud based interface, and enable advanced functions like remote programming (e.g., remotely powering on or off heating systems, changing lighting conditions, etc.). At local and personal level, this concept is realized as Home Automation and Smart Home. IoT devices can be used to monitor and control the mechanical, electrical and electronic systems used in various types of buildings (e.g., public and private, institutions, industrial or residential). Home automation systems, like other building automation systems, are typically used to control heating, lighting, appliances, ventilation, air conditioning, communication systems, entertainment and home security devices to improve security, reliability, convenience and energy efficiency of the all the controlled installation in the house.

Also in Medical and Healthcare areas the Internet-Of-Things provide an IT revolution. IoT devices can be used to provide remote health monitoring and emergency notification systems. These health monitoring devices can range from blood pressure and heart rate monitors to advanced devices capable of monitoring specialized implants, such as pacemakers or advanced hearing aids. Specialized sensors can also be equipped within living spaces to monitor the health and general well-being of senior people, also ensuring that proper treatment is being provided to those people. The Internet-Of-Things application allows also to realize other consumer devices to encourage healthy living, such as wearable heart monitors. As we saw previously in "Chapter 2- The Board World", micro-devices like the Arduino LilyPad for clothes installation are currently on the market.

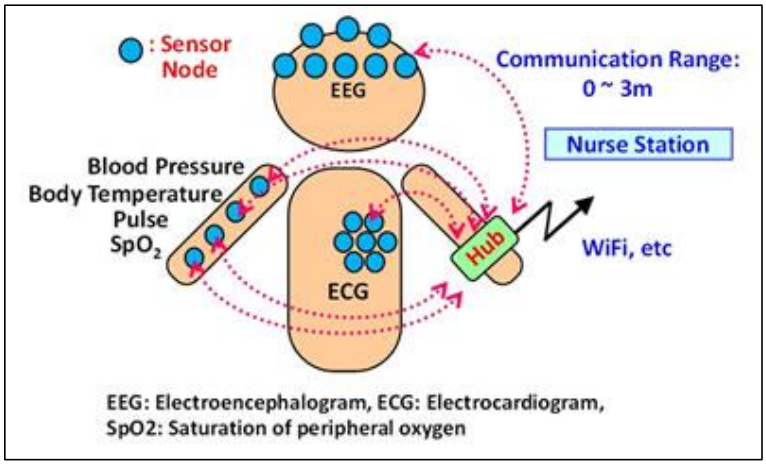

Figure 8-4. Medical and Healthcare IoT

At this point the reader has the instruments to exploit the full Internet-Of-Things power of the Intel Galileo board to realize any of the just described IoT application. On the Arduino side, the maker could build any complex extension electronic circuit around the board. The Arduino community is full of examples, from the remote controlled robots to the advanced systems for energy management in Smart Homes. Furthermore, on the Linux side the experienced programmer could implement any kind of service. The web application we proposed is a simple RESTful web application, but on a Linux platform could be implemented advanced web systems like SOAP Web Services. As we said in the previous chapters of this paper, the Linux system offers to the developer possibilities without limitation.

Bibliography

Arduino. (n.d.). Retrieved from http://www.arduino.cc/.

BeagleBoard. (n.d.). Retrieved from http://beagleboard.org/.

Building Yocto Applications using Intel® C++ Compiler with Yocto Project Application Development Toolkit. (n.d.). Retrieved from https://software.intel.com/en-us/articles/building-yocto-applications-using-intel-c-compiler.

Clay and Galileo. (n.d.). Retrieved from http://www.hofrock.com/.

Download Software for the Intel® IoT Developer Kit v1.0. (n.d.). Retrieved from https://software.intel.com/en-us/iot/downloads.

Efficient communication between Arduino and Linux native processes. (n.d.). Retrieved from https://software.intel.com/en-us/blogs/2014/09/22/efficient-communication-between-arduino-and-linux-native-processes.

Galileo Gen2 : GPIO Control Using Linux. (n.d.). Retrieved from https://anandvetcha.wordpress.com/2014/09/28/galileo-gen2-gpio-control-using-linux/.

Getting Started for C/C++ (Eclipse) - Galileo & Edison. (n.d.). Retrieved from https://software.intel.com/en-us/getting-started-for-c-c-plus-plus-eclipse-galileo-and-edison.

Intel. (2013). *Intel Quark SoC X1000 Board Support Package (BSP) Build Guide.*

Intel. (2013). *Quick Start Guide For Using Intel Compiler To Build Applications Targeting Quark SoC On Galileo Board.*

Intel. (2014). *Intel Galileo and Intel Edison Release Notes.*

Intel. (2014). *Intel Galileo Board and Intel Galileo Gen 2 Board Shield Testing Report.*

Intel. (2014). *Intel Galileo Board User Guide.*

Intel. (2014). *Intel Galileo Gen 2 Datasheet.*

Intel. (2014). *Intel Galileo Gen2 Intel Quark X1000 Schematic.*

Intel. (2014). *Intel Galileo I/O Mappings.*

Intel. (2014). *Intel Galileo Software.*

Intel. (2014). *Intel Galileo Software Release Notes.*

Intel. (2015). *Intel Galileo and Intel Edison Release Notes (Revision 008).*

Intel Galileo. (n.d.). Retrieved from http://arduino.cc/en/ArduinoCertified/IntelGalileo.

Intel Galileo. (n.d.). Retrieved from http://www.intel.com/content/www/us/en/do-it-yourself/galileo-maker-quark-board.html.

Intel Galileo Community. (n.d.). Retrieved from https://communities.intel.com/community/makers/galileo.

Intel Galileo Official Software Repository. (n.d.). Retrieved from http://repo.opkg.net/galileo/.

Intel® Galileo Software Package. (n.d.). Retrieved from https://downloadcenter.intel.com/download/24272.

Raspberry Pi. (n.d.). Retrieved from http://www.raspberrypi.org/.

Sergey's Blog. (n.d.). Retrieved from http://www.malinov.com/Home/sergey-s-blog.

SQLite. (n.d.). Retrieved from https://www.sqlite.org/.

UDOO. (n.d.). Retrieved from http://www.udoo.org/.

Working with Galileo without Arduino IDE. (n.d.). Retrieved from https://communities.intel.com/message/224718.