Electrical Engineering
Electronics
Computers
Communications

הטכניון
מכון טכנולוגי
לישראל

The Lorry I. Lokey Interdisciplinary Center for Life Sciences and Engineering
NETWORK BIOLOGY RESEARCH LABORATORIES

הפקולטה להנדסת חשמל - המעבדה לחקר רשתות ביולוגיות

דו"ח סיכום פרויקט ב'

Characterization of the Intel® Galileo Board and Implementation of a Graphical User Interface

מבצע: גיל אייזנשטדט Gil Aizenshtadt

מנחה: חננאל חזן Hananel Hazan

סמסטר חורף תשע"ה

Credits

My guiding host and advisor for this project was Hananel Hazan. I want to thank Hananel for all the help and support he provided during the project. Hananel was very helpful, attentive to problems and willing to help.

Table of Contents

Table of Figures **5**

Table of Tables **6**

Abstract **7**

Introduction **8**

The Arduino controllers *8*

The Intel® Galileo controller *9*

Chapter 1: Description of the Intel® Galileo **10**

Storage *10*

Peripherals *10*

Digital Pins *11*

Analog Pins *12*

Additional Pins *12*

Physical Characteristics *13*

Chapter 2: Examining the Galileo's performance **14**

Measuring the Galileo's output speed at the Digital Pins *14*

First method: "OUTPUT" mode and digitalWrite() function 15

Second method: "OUTPUT_FAST" mode with digitalWrite() function 15

Third method: "OUTPUT_FAST" mode with fastGpioDigitalWrite() function 16

Fourth method: "OUTPUT_FAST" mode with fastGpioDigitalWriteDestructive() function 17

Measuring the Galileo's sampling rate of the Analog Input Pins *18*

Conclusions *18*

Chapter 3: The Graphical User Interface **19**

Connection options between the Galileo and the PC *19*

The Language Platform of the GUI *19*

The Intel® Galileo Graphical User Interface – IGG *20*

The Simulation tab 20

The Measurements Tab 22

The components of the Simulation and Measurements Tabs 24

The Monitor Tab and its Components 32

The "File" and "Options" Menus 37

The Interface Setup Graphical User Interface – sIGG *39*

The sIGG's Implementation 40

Tutorial – how to use the IGG and sIGG *42*

Setting up the Galileo 42

Using the Interface Setup GUI – the sIGG 44

Using the Intel® Galileo GUI – The IGG 44

Chapter 4: Summery **55**

Suggestions for further use *55*

References **56**

Appendices **57**

Appendix A – Making a code running on the Galileo on startup *57*

Appendix B – Downloading a file to the Galileo *59*

Appendix C – Sketches for examining the performance of the Galileo *60*

Appendix D – Templates and Code for all the Sessions (Simulation, Measurements and Monitor) *62*

Simulation Template 63

Measurements Template 67

Monitor Code 69

Table of Figures

Figure 1: The Arduino family boards 8
Figure 2: The Intel® Galileo board: front (left) and rear (right) views. 9
Figure 3: The Galileo board overview 11
Figure 4: The Galileo board's interface and connections scheme. 13
Figure 5: The Intel Galileo board connected to an oscilloscope for measuring the possible output frequencies. 14
Figure 6: The output of digital pin 2 in the case of "OUTPUT" mode and digitalWrite() function. 15
Figure 7: Output of digital pin 2 in the case of "OUTPUT_FAST" mode and digitalWrite() function. 15
Figure 8: Output of digital pin 2 in the case of "OUTPUT_FAST" mode and fastGpioDigitalWrite() function 16
Figure 9: The output of digital pin 2 in the case of "OUTPUT_FAST" mode and fastGpioDigitalWriteDestructive() function 17
Figure 10: The initial stage of the Simulation tab, offering to load a code and edit it in the text editor 21
Figure 11: The app's Simulation tab after the simulation started. A plot displays the data received from the Galileo, and the user can update the trigger or parameters while the simulation is running 21
Figure 12: The initial stage of the Measurements tab, offering to load a code and edit it in the text editor 23
Figure 13: The app's Measurements tab after the session ran and stopped. A graph displays the data received from the Galileo, and the user can either save the data from the session, or return to the code editing. 23
Figure 14: The Trigger Panel 24
Figure 15: The Code Loading Panel 25
Figure 16: The Parameters Panel with 4 visible parameters with set values and labels 26
Figure 17: The Code Text Area 27
Figure 18: The "Update" button when the session is running. 29
Figure 19: The Graph Panel displaying the first output channel in the plot. 30
Figure 20: The Monitor tab, offering the user to monitor the analog inputs of the Galileo, and also the data monitored to a file 32
Figure 21: The Monitor tab after the session was stopped. The user is offered to save the data and start another session. 32
Figure 22: The Channels Panel 33
Figure 23: The Graph Panel of the Monitor tab 34
Figure 24: The "Exit" option found in the "File" menu. 37
Figure 25: The "Add new code to stored codes..." option found in the "Options" menu. 37
Figure 26: The sIGG app for setting up the preferences and interfaces for the main app - the IGG 39
Figure 27: After saving the preferences file, the app asks the user whether to setup the monitor and network interfaces too 41
Figure 28: The Device Manager window. 42
Figure 29: The manual update of the Galileo Drive, the specified location is incorrect and should be the "hardware/intel/i586-uclibc" folder within the Arduino Galileo IDE 43
Figure 30: The Intel® Galileo Firmware updater. 43

Figure 31: The Simulation Tab, to load a code from the PC, select the "Choose from file option". 45
Figure 32: The File Browser. Locate your file on the PC and press "Open". The path cannot contain spaces. 45
Figure 33: (A) Choosing the shape of the wave. (B) Specifying the trigger fields for a Sine wave. (C) Specifying the trigger fields for a Pulse wave 46
Figure 34: Specify the parameters and their values 46
Figure 35: The Simulation is now running. You can change any field in the black rectangles. 47
Figure 36: After the simulation has stopped you can save the data and return to the code. 47
Figure 37: The Measurements Tab, to load a code from the PC, select the "Choose from file option" 48
Figure 38: The File Browser. Locate your file on the PC and press "Open". The path cannot contain spaces. 49
Figure 39: Specify the parameters and their values 49
Figure 40: The Simulation is now running. You can change any field in the black rectangles. 50
Figure 41: After the simulation has stopped you can save the data and return to the code. 50
Figure 42: The Monitor Session is starting up. 51
Figure 43: The Monitor session running with 3 analog inputs displayed - A0, A2, A5 52
Figure 44: The Monitor Session has stopped. You can now save the data from the session or start a new one. 52
Figure 45: Opening the "Options" window and selecting the "Add new code to stored codes..." option 53
Figure 46: A popup window asking where to store the code. 53
Figure 47: The File Browser window. Locate your code and press "Open" 53
Figure 48: Give your code a label. 54
Figure 49: After storing the code, it can be found the code-selection menu 54
Figure 50: The clupload_win.sh script. Used for downloading the sketch to the Galileo and starting it. 59
Figure 51: Four sketches for measuring the possible output frequency of digital pin 2 60
Figure 52: The code used to measure the sampling rate of the Galileo. 61
Figure 53: The MatLab code used to calculate the sampling rate of the Galileo 62

Table of Tables

Table 1: Average sampling frequencies of the analog input pins 18

Abstract

This project deals with a controller, made by Intel© Company, called the **Intel® Galileo Board.**
This controller was developed by Intel for education needs and for learning and experimenting with programming, as well as combining programming with hardware.

The controller is a part of a family of similar controllers, called Arduino controller, which create a cheap and comfortable environment of developing and running projects which combine software & hardware components.

This work was mainly based on two goals. The first goal is to examine the possibility of integrating the controller in the daily work of the laboratory workers, and even replace the existing instruments currently used in the laboratory. For that purpose, a characterization of the controller's abilities was performed. In addition, the performance of the controller was examined, especially it's processing rates and response time. The second goal is to build a user-friendly app to allow the laboratory workers to use the controller without any necessary professional knowledge in programming.
The program allows:

- Downloading and running source code files written in C++.
- Communication between the computer and the controller, in the form of sending & receiving strings of data.
- Showing the measurements from the controller's analog inputs on a graphical plot.
- Showing the output of the code running on the controller, on several sub-plots (one sub-plot for each output variable).
- Viewing and modifying the code before downloading it to the controller.

In addition to the program above, another secondary program was built, which helps create the infrastructure which is used by the main program. The Infrastructure includes creating a preferences file, turning on the Ethernet connection interface on the controller and downloading a compiled code to the controller for the monitoring state of the controller. A user guide was also written (and presented in the report) which explains what is needed to be set on the computer in order for the program to work properly. The guide also explains how to use the built programs.

<u>תקציר</u>
הפרויקט עוסק בבקר מחברת אינטל שנקרא Intel® Galileo. הבקר פותח ע"י חברת אינטל לצורכי חינוך ולמידה וניסוי בתכנות, כמו גם תכנות עם חומרה. הבקר הינו חלק ממשפחת בקרים דומים, הנקראים בקרי Arduino, אשר יוצרים סביבה נוחה וזולה לפיתוח וביצוע פרויקטים אשר משלווים רכיבי תוכנה וחומרה.

לפרויקט שתי מטרות עיקריות. הראשונה היא בדיקת התכנות של שילוב הבקר בעבודה יום-יומית של נסייני המעבדה, ואפילו החלפה של מכשירים קיימים במעבדה, בבקר. לצורך כך, בוצע אפיון של יכולות הבקר והביצועים שלו, בפרט מדידת קצב הוצאת אותות דיגיטליים וזמן הדגימה של אות פיסיקלי בכניסות האנלוגיות.
המטרה השניה של היא בניית ממשק משתמש גרפי ידידותי למשתמש, שיאפשר לנסייני המעבדה לעבוד עם הבקר בצורה קלה ללא צורך בידע מעמיק בתכנות.
התוכנה מאפשרת:

- הורדה של קבצי קוד בשפת C++ והרצתם על הבקר.
- תקשורת בין המחשב לבקר, בצורה של שליחה וקבלה של מחרוזות מידע.
- הצגה גרפית של דגימות הכניסות האנלוגיות של הבקר.
- הצגה גרפית של פלט הקוד, של המשתמש, שרץ על הבקר.
- הצגה ועריכה של הקוד של המשתמש לפני קימפול (הידור) והורדה של הקוד לבקר.

בנוסף לתוכנה הנ"ל נבנתה עוד תוכנת משנה אשר עוזרת ליצור את התשתית שבה תשתמש התוכנה הראשית (התשתית כוללת קובץ העדפות, הפעלת ממשק התקשורת דרך ה Ethernet על הבקר, והורדת קוד מקומפל לבקר לצורך מצב ניטור הכניסות האנלוגיות שלו), וגם נכתב מדריך למשתמש שמסביר מה דרוש כדי לעבוד עם הבקר דרך המחשב ובאמצעות התוכנה, וגם איך להשתמש בתוכנות שנבנו.

Introduction

To this date, several measurement instruments are available to the laboratory workers. These instruments are used for measuring physical signals, e.g. voltage & current, which allow the analysis of biological systems, including neural networks. Those measuring instruments, are usually very expensive and have a very specific function.

Lately, a community of developers and users of micro-controllers has begun to grow. These micro-controller boards are relatively cheap in price and small in size. The more popular are the Arduino micro-controllers, which are characterized by their capabilities to measure and process analog signals.

The mentioned above raises the question of whether an integration of these controllers with the laboratory's work is possible, i.e. can they be used for sampling the analog signals of a certain biological system, process them, and return a feedback to the system, in "Real-Time" rates.

This project in particular, focuses on the Intel® Galileo micro-controller, and whether it meets the necessary requirements as a measuring tool, or not. Moreover, this work tests if the controller can be used for monitoring signals and performing simulations of certain models.
In addition, a Graphical User Interface (GUI) program will be built, with the purpose of sparing the user of the controller the need of profound knowledge of the software/hardware environment, and allow a simple use of the controller for signal and data processing.

The Arduino controllers

The Arduino family is comprised of micro-controller boards which use either Atmel AVR micro-controllers, or 32-bit Atmel ARM processors. Some of the boards in the family can be seen in *Figure 1*. These boards provide sets of digital and analog I/O (input/output) pins that can be connected to various expansion boards, or "shields" and other circuits. Using the shields, or existing interfaces, the boards can connect to the internet, and exchange data with the network they are connected to. On most of the boards there are 14 digital I/O pins, 6 of which can produce Pulse-Width modulated (PWM) signals, and 6 analog inputs, which can also be used as digital I/O. On the software side, the Arduino platform provides an integrated development environment (IDE) based on the "Processing" project, which offers object-oriented syntax, similar to the Java and C++ languages (and more). The IDE offers writing a code in a simple syntax and then, before compiling it, the IDE adds the necessary code and converts the file into a C++ source file.

The main idea behind the Arduino world is offering an inexpensive and easy way for both novices and professionals to make projects and create devices which interact with their surroundings using sensors and actuators. Also, the hardware specifications of the boards are openly available (Open-Source), thus allowing the boards to be manufactured by anyone.

Figure 1: The Arduino family boards[1]

[1] This picture was taken from: http://www.robotshop.com/blog/en/arduino-robotics-projects-3666 in the "About Arduino" page, in the "RobotShop" site.

The Intel® Galileo controller

The Galileo (shown in *Figure 3*) is a micro-controller designed on a small electronic board. The controller uses an Intel Quark SoC (System-on-chip) x1000 processor, which has a very low power consumption.

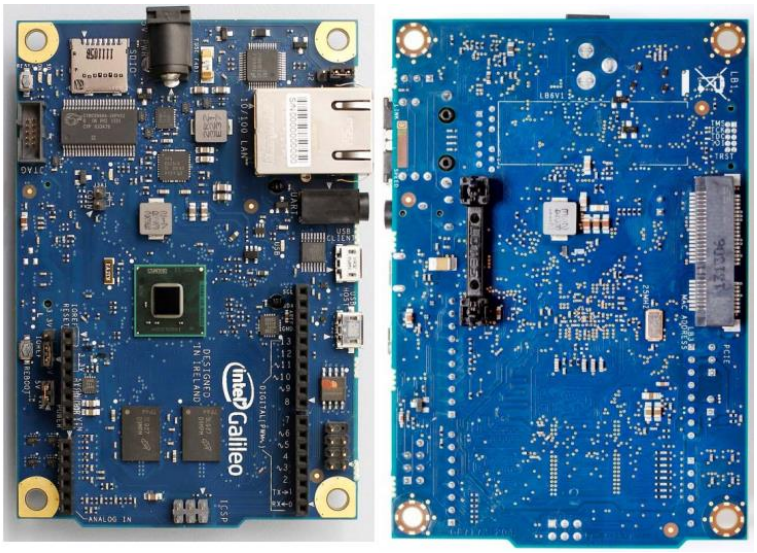

Figure 2: The Intel® Galileo board: front (left) and rear (right) views.[2]

This is the first controller based on hardware from the Intel© Company. It was designed in such way that supports the software environment which already exists and used in the Arduino controllers. The Galileo fits both the pins' arrangement and placement and the software developing environment, which are based on the Arduinos. In this way, the process of "getting to know" the Galileo, and also the use of it, don't require additional knowledge for those who are already familiar with the Arduino environment. In addition, for the new users, the learning process is very simple and quick. All of this allows the Galileo to fit in an existing and growing world of controllers, in a smooth and quick way.
A principle difference between the Arduino family and the Intel® Galileo lies in the processor architecture – while the Arduino boards use ARM-based processors, that uses a Reduced Instruction Set Computing (RISC) which is smaller and simpler, the Galileo uses an x86 processor that uses a Complex Instruction Set Computing (CISC) which is more complex but more powerful. This causes the Galileo to use more power than the Arduino, however an operation that takes a single clock cycle on the Galileo, can take 3 cycles on the Arduino, thus making the Galileo faster and more powerful than the Arduino.

[2] This picture was taken from: http://fabioangeletti.altervista.org/blog/intel-galileo/?doing_wp_cron=1440941783.8720419406890869140625 in the "Intel Galileo" page in the "Fabio's blog about electronics" site.

Chapter 1: Description of the Intel® Galileo

The Galileo board is powered either with the regulated 5V AC-to-DC adapter (recommended), or via the USB Client socket (not recommended).
As mentioned above, the Galileo has a 32-bit Quark SoC X1000 process. It is an x86-based, low-power embedded system-on-a-chip that can run at up to 400 MHz, and has a 512KB SRAM built-in. The processor has a single core and a 16KB L1 Cache, and uses a Complex Instruction Set Computing, which allows powerful and fast processing (on the expense of more hardware and higher power consumption).

Storage

The Galileo has an 8MB legacy SPI Flash whose main purpose is to store the firmware (or bootloader) and the latest sketch. Between 256KB and 512KB is dedicated for sketch storage, and the download happens automatically from the connected PC.
The Galileo board also has a 512KB embedded SRAM and a 256MB DRAM, enabled by the firmware by default. In addition, there is 11KB EEPROM which can programmed via the EEPROM library. The Galileo also supports external storage, via the USB connection (USB host), or via a µSD card with up to 32GB of storage (the µSD can also be used to run a Linux image on the board).

Peripherals

The Galileo has a series of peripherals:

- **Ethernet socket** – connects the Galileo to any Local-Area-Network (LAN) with a rate of 10/100 MB/s.
- **RS-232 UART** Port – a programmable speed 3.5mm jack port. Can be used to connect to the board's Linux terminal.
- **USB Client** – allows connecting the board to the computer and downloading a sketch using the Arduino IDE. Can also be used to control USB devices connected to it.
- **USB Host** – allows the board to act like a host for connected peripherals as mice, keyboards, smartphones, mass storages, etc. With a USB hub, up to 128 devices can be connected to this port.
- **µSD Slot** – supports up to 32GB SD cards. This option allows installing a "Linux" image and booting from it. Also can be used as a storage device and read from using the SD library (Arduino).
- Two buttons – one **Reboot button** which will restart the entire Galileo (including the Linux image), and one **Arduino Sketch Reset** button which will only restart the sketch running on the Galileo.
- **JTAG socket** – a 10-pin standard JTAG header for debugging.
- **Mini PCIe** (Peripheral Component Interconnect Express) **socket**[3] – which supports full size and half size (with adapter) mPCIe modules, such as WiFi, Bluetooth and Cellular. The socket also provides an additional USB host port via the mPCIe slot.
- **Coin cell pins** – These 2 pins offer the optional connection of a 3V "coin cell" battery for an integrated **Real Time Clock** (RTC), which allow operation between turn-on cycles.

[3] Note: Except the mPCIe slot, all mentioned peripherals are placed on the top side of the Galileo, and shown in *Figure 3*. The mPCIe is placed on the bottom side and shown in *Figure 2*.

Digital Pins

As mentioned above, the Galileo has 14 digital General-Purpose-Input-Output (GPIO) pins.
Of those pins two (pin 0 and pin 1) can be used for UART connection and six (pins: 3, 5, 6, 9, 10, 11) can be used as Pulse-Width-Modulated (PWM) outputs. All the pins can be controlled in the code downloaded to the Galileo, using the pinMode(), digitalWrite() and digitalRead() functions. Each pin can source (provide positive current) a maximum of 10 mA or sink (provide negative current) a maximum of 25 mA, and has an internal pull-up resistor (which is disconnected by default) in the range of $5.6{,}10\ K\Omega$.

When several pins act as sources/sinks, the max current source is 40 mA and the max current sink is 100 mA. When all the pins are acting as sources/sinks, the max current source is 80 mA and the maximum current sink is 200 mA.

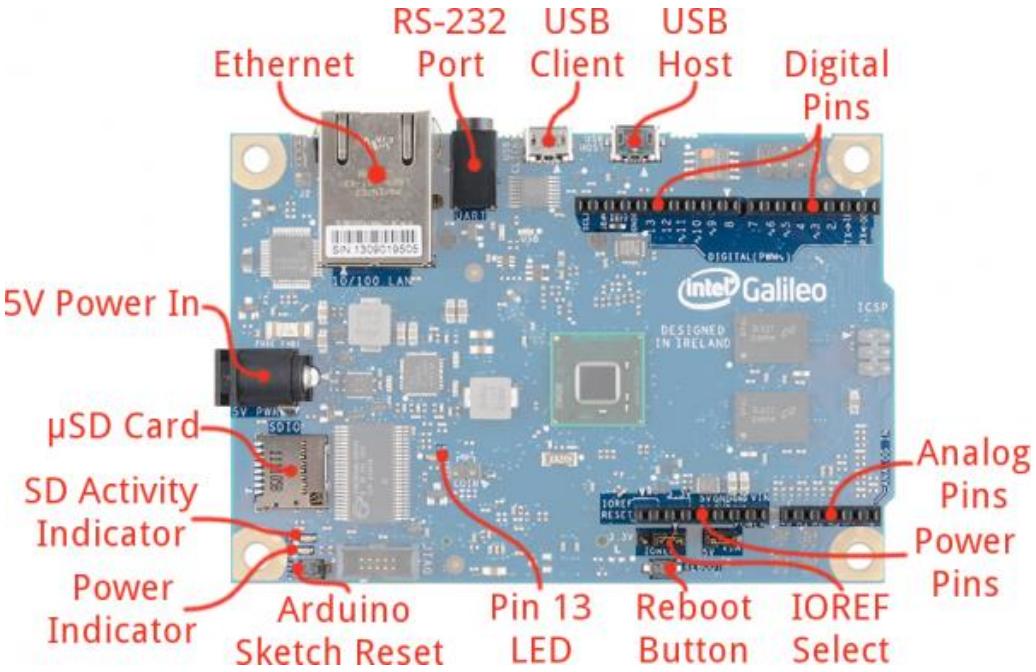

Figure 3: The Galileo board overview[4]

In the regular "OUTPUT" mode, the digital pins can change their outputs with a frequency of 230Hz, which is relatively slow. However, the Galileo has an option of increasing the frequency of the output in pins 2 & 3:

- When the pins are set to "OUTPUT_FAST" mode, the output frequency can increase to up to 477 KHz.
- When the pins are controlled with the fastGpioDigitalWrite() function, the output frequency increases to up to 680 KHz (with the "OUTPUT_FAST" mode).
- When the pins are controlled with the fastGpioDigitalWriteDestructive() function, the output frequency can increase to up to 2.93MHz (with the "OUTPUT_FAST" mode).

[4] This picture was taken from: https://learn.sparkfun.com/tutorials/galileo-getting-started-guide in the "Galileo getting started guide" page in the learn.sparkfun.com site.

Analog Pins

Pins A0 – A5 are the analog pins of the Galileo. While they can be used in the same way as the digital pins, these pins can also be used as analog input pins, using the AD7298 analog-to-digital (ADC) 12-bit converter[5]. By default, the analog pins measure from ground to 5 volts and with a 10-bit resolution, providing a resolution of $\sim 4.8\ mV$. Increasing the output to 12-bit gives a better resolution of $\sim 1.2\ mV$.

Additional Pins

There are 12 more pins in addition to the 14 digital pins and 6 analog pins:

- The AREF pin – unused on the Galileo.
- The SDA and SCL pins – used for connecting to the board's I^2C Bus.
- 3 GND pins – used as ground pins.
- The VIN pin – the input voltage to the Galileo board when it's using an external power source (as opposed to 5 volts from the regulated power supply connected at the power jack). The voltage can be supplied to the board through this pin, or accessed to through this pin, if the board is connected to power through the power jack.
- 5V output pin – This pin outputs 5V from the external source or the USB connector. The maximum current draw from this pin is 800 mA.
- 3.3V output pin – A 3.3 volt supply generated by the on-board regulator. The maximum current draw from this pin is 800 mA.
- The IOREF pin – allows an attached shield (external circuit board) with the proper configuration to adapt to the voltage provided by the board. The IOREF pin is controlled by a jumper on the board, i.e., a selection jumper which is used to select between 3.3V and 5V shield operation.
- The RESET pin – when the pin is in LOW, the sketch is reset. This is typically used to add a reset button to shields that block the one on the board.

Figure 4 shows the connection scheme of the Intel® Galileo board. It includes the connection and wiring interface between the processor and the peripheral sockets, connectors and headers, as well as the interface of the digital and analog pins, going through various multiplexers and sub-units (such as the ADC in the analog pins, and PWM unit in both the analog and digital pins).

[5] Note: the AD7298 has 8 channels as input, however only 6 of them are connected on the Galileo board. Also the ADC's rate is 1 MSPs, however it is much lower on the Galileo, since the ADC is connected through the I^2C Bus interface which works at 100KHz.

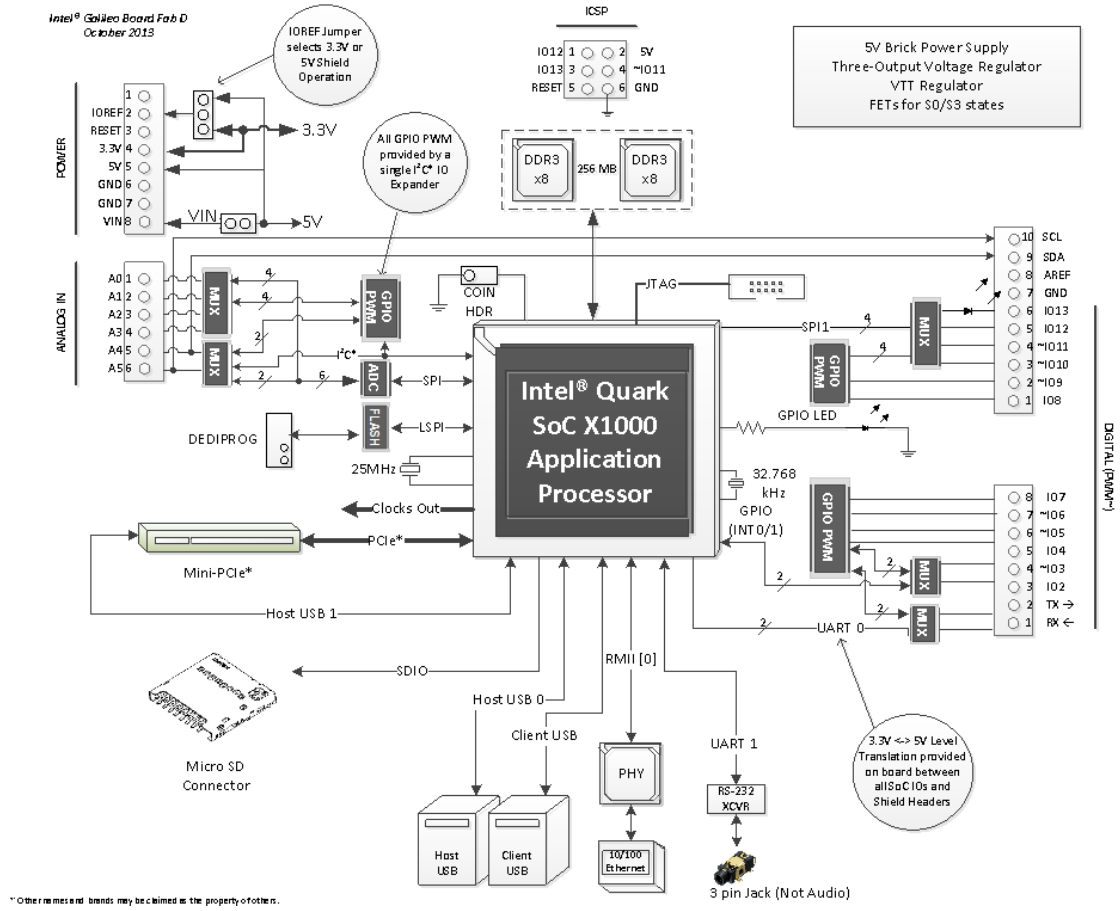

Figure 4: The Galileo board's interface and connections scheme.[6]

Physical Characteristics

The Galileo board is 10.6 cm (4.2 inches) long and 7.1 cm (2.8 inches) wide, but the USB connectors, UART jack, Ethernet connector and power jack are extending a bit beyond the these dimensions. Four screw holes allow the board to be attached to a surface or case.

Note: the distance between digital pins 7 & 8 is not an even multiple of the spacing between the other pins.

[6] This picture was taken from the galileo data-sheet at: http://www.intel.com/support/galileo/sb/CS-035174.htm

Chapter 2: Examining the Galileo's performance

In this chapter, the response time, or the frequency of the outputs of the Galileo, will be measured and analyzed. Using simple sketches, the Galileo will be configured to produce a pulse wave at the digital pin output and then, by connecting the output to an oscilloscope, the frequency will be measured. Also, an additional sketch will be used to measure the sampling rate of the Galileo.

Measuring the Galileo's output speed at the Digital Pins

As mentioned in chapter 1, there are 4 possible frequencies at which the Galileo can change its outputs:

- A frequency of 230 Hz, using a regular "OUTPUT" mode of the pin.
- A frequency of 477 KHz, using the "OUTPUT_FAST" mode, in pins 2 & 3.
- A frequency of 680 KHz, using the fastGpioDigitalWrite() function in pins 2 & 3.
- A frequency of 2.93 MHz, using the fastGpioDigitalWriteDestructive() function, in pins 2 &3.

These frequency were specified in the Intel community forum Q&A[7]. To verify the data specified, 4 sketches were used, where in each the Galileo was configured to change the output voltage in pin 2, and the frequency was measured by an oscilloscope. *Figure 5* shows the Galileo connected to the oscilloscope, through a matrix board.

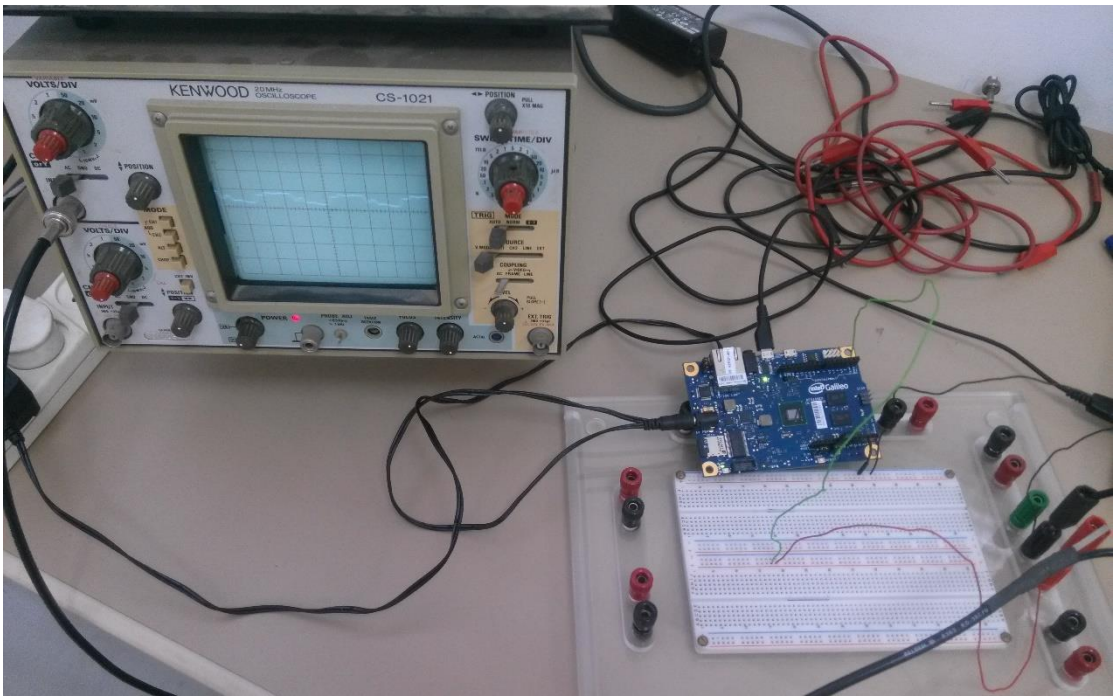

Figure 5: The Intel Galileo board connected to an oscilloscope for measuring the possible output frequencies.

[7] "I/O speeds?" Thread at the Intel community forum at: https://communities.intel.com/message/207904

First method: “OUTPUT” mode and digitalWrite() function

Note: The sketches that were used for this method and the other 3 are found in Appendix C.
In addition, for *Figures 6-9*, the yellow markers are the resolution of the scope. For x-axis it’s Sweep Time/Div and for y-axis it’s Volts/Div. Also, the figures show the time period length in terms of number of squares in the x-axis, painted in yellow as well.

In this method, the mode of the pin is set to be “OUTPUT” using the pinMode() function, and then the output is written using the digitalWrite() function. This method holds for all the GPIO pins on the board (while the next 3 methods hold only for digital pins 2 & 3).

The results are presented on the oscilloscope in *Figure 6*. As can be seen the time resolution is $2[ms]$ so the period is $T = 2\frac{1}{5} \cdot 2[ms] = 4.4[ms]$ and the frequency is $F = 1/T \cong 227[Hz]$, as expected.

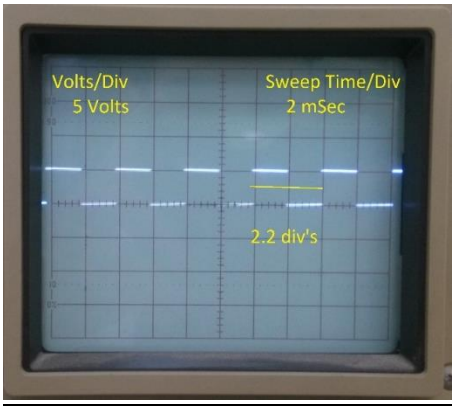

Figure 6: The output of digital pin 2 in the case of "OUTPUT" mode and digitalWrite() function.

Second method: “OUTPUT_FAST” mode with digitalWrite() function

In this method, the pin is set to be in “OUTPUT_FAST” mode using the pinMode() function, and then the output is written using the digitalWrite() function.
The results are presented on the oscilloscope in *Figure 7*. As can be seen the time resolution is $1[\mu s]$ so the period is $T = 2\frac{1}{5} \cdot 1[\mu s] = 2.2[\mu s]$ and the frequency is $F \cong 454[KHz]$, as expected. Notice that in this case the output is distorted since the time scale approaches the capacitive discharge/charge delay.

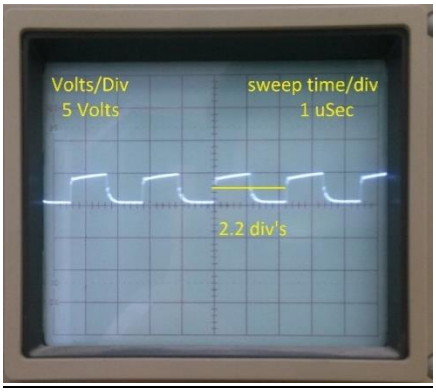

Figure 7: Output of digital pin 2 in the case of "OUTPUT_FAST" mode and digitalWrite() function.

Third method: “OUTPUT_FAST” mode with fastGpioDigitalWrite() function

Like in the previous two, in this method the pin is set to be in “OUTPUT_FAST” mode using the pinMode() function, however the writing of the output is performed with the fastGpioDigitalWrite() function. This function writes directly to the pin’s register, as opposed to the digitalWrite() function which performs additional operations which make it slower.

The results are presented on the oscilloscope in *Figure 8*. As can be seen the time resolution is $1[\mu s]$ so the period is $T = 1.5 \cdot 1[\mu s] = 1.5[\mu s]$ and the frequency is $F \cong 667[KHz]$, as expected. Notice the distortion in the output, which is similar to that from the previous case.

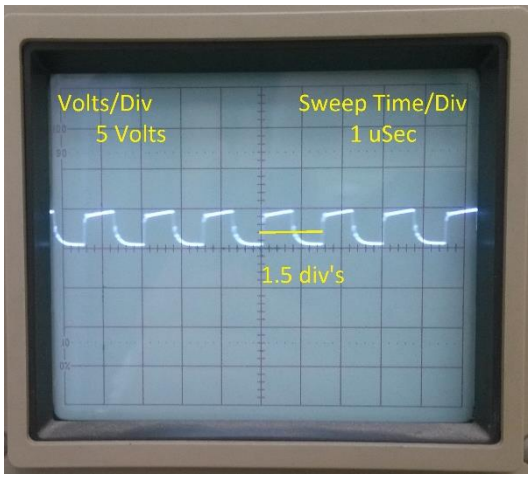

Figure 8: Output of digital pin 2 in the case of "OUTPUT_FAST" mode and fastGpioDigitalWrite() function

Fourth method: “OUTPUT_FAST” mode with fastGpioDigitalWriteDestructive() function

This method also uses the “OUTPUT_FAST” mode of the pin. The previous methods, when needed to write a value to the output, first read the current value of the register, then update it and finally write back the new value to the register (read-modify-write operation).
However, this method uses a destructive writing, i.e. writes the GPIO bits straight to the register with reading the stored value. This allows the method to modify the output up to twice as fast as the previous method’s speed (and four times as fast as the previous frequency). This method also uses the fastGpioDigitalLatch() function which allows the code to latch the initial state of the pin, before updating it.
The results are presented on the oscilloscope in *Figure 9*. As can be seen the time resolution is $0.1[\mu s]$ so the period is $T = 3.5 \cdot 0.1[\mu s] = 0.35[\mu s]$ and the frequency is $F \cong 2.87[MHz]$, as expected. In this case the output is totally distorted and noisy.

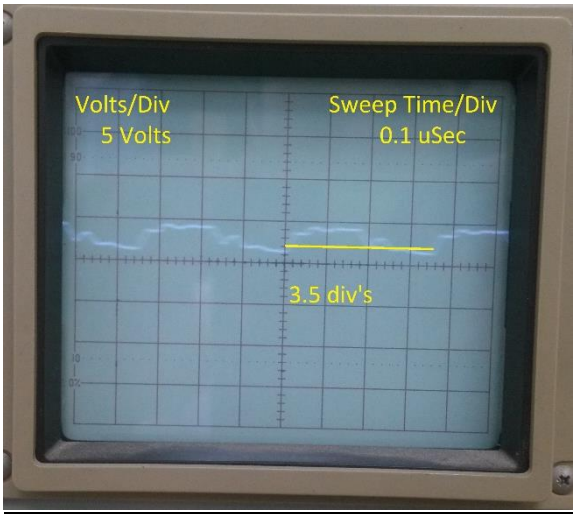

Figure 9: The output of digital pin 2 in the case of "OUTPUT_FAST" mode and fastGpioDigitalWriteDestructive() function

Measuring the Galileo's sampling rate of the Analog Input Pins

In order to consider the use of the Galileo in the everyday laboratory's work, the sampling rate at the analog inputs has to be measured. To do that, a sketch was used to configure the board to read the analog inputs to a variable, and measure the time taken to do that task. It was also configured to read several analog inputs in succession in order to check the board's performance in successive reading. The sketch used can be found in Appendix C.

The measurements were printed to the serial terminal (using the Serial.print() function), and then processed in Matlab, to give the average frequencies. The results are shown in *Table 1*.

Number of channels read	*Same channel read*	*Different channels read*
1	1.2 ± 0.196 *KHz*	1 ± 0.185 *KHz*
2	1.1 ± 0.17 *KHz*	1 ± 0.132 *KHz*
3	1.07 ± 0.069 *KHz*	1.04 ± 0.086 *KHz*
4	1.15 ± 0.085 *KHz*	1.1 ± 0.086 *KHz*
5	1.16 ± 0.065 *KHz*	1.13 ± 0.07 *KHz*
6	1.15 ± 0.05 *KHz*	1.15 ± 0.06 *KHz*

Table 1: Average sampling frequencies of the analog input pins

As can be seen from the results, the sampling frequency of the analog inputs is about the same, at ~ 1.1 *KHz* .

Conclusions

The requirements of the laboratory equipment are to be able to measure changes in the scales of milliseconds and lower (micro-seconds). Although with some adjustment the Galileo is able to send pulses in these time scales (as we saw in the case where the Galileo can output pulses with a frequency of 450KHz, 670KHz and 2.87 MHz), however the sampling performance borders on the limit of a millisecond, which might make it miss the very small changes in the system it measures. Moreover, when the output is set to high frequency, it becomes distorted, due to the capacitors in the output. Thus, without any external equipment the Galileo isn't fit for small time scale systems or sensitive systems. However, it can still be used for simulations and systems with a higher time scale (even several milliseconds).

Another disadvantage is the fact that the Galileo's working range is between ground and 5 volts (or ground and 3.3 volts), and it won't measure negative voltage. This problem can be solved with an external shield which shifts the voltage range to a positive range, so that the Galileo will be able to measure it.

To conclude, the Galileo is not fit for replacing the laboratory equipment, however it can be used for simulations and for measuring analog signals with a certain frequency limit. For that purpose and more, a Graphical User Interface is built.

Chapter 3: The Graphical User Interface

This chapter discusses the Graphical User Interface (GUI) application (app) that was built for using the Galileo in a simple way without any profound programming knowledge required. The connection options to the board are discussed, as well as the app itself – what does it offer, and how to use it. Another app which sets up the network and monitor interfaces, as well as the preferences file, is discussed.

Connection options between the Galileo and the PC

There are two main interfaces through which the personal-computer (PC) can connect to the Galileo:

1. A serial connection via the USB host socket – when the Galileo is connected to the computer through the USB host connection, sketches can be downloaded using the Arduino IDE software, or the GUI (will be discussed later). After the sketch is downloaded, a serial terminal can be initiated to allow receiving data from the board, as well as sending data to it (using the Serial.print() and Serial.read() functions in the sketch code).
2. A network connection – the Galileo can be connected to a local network (or PC) via the Ethernet socket or a Wi-Fi shield attached, and allow data exchange with the network at much higher speeds than the serial connection. Also, if a Linux image is present and the Galileo boots from it, then a terminal session to the Linux on-board can be opened.

Contrary to the USB serial connection, where a terminal can't be open while downloading sketches to the Galileo and only one terminal instance can run at a time, the network connection can support file and data exchange at the same time (also allowing several terminals to be opened and run simultaneously). Although the network connection is significantly better than the serial connection, the serial connection is still used, as the Arduino IDE itself provides an option which automatically compiles a sketch and then downloads it to the board through the USB (thus saving a lot of time and effort). Also, by slightly modifying the download script to pass commands to the board, the network interface can be established easily through the USB.

The Language Platform of the GUI

The language platform on which the GUI was built is Java. Java is a high-level, concurrent, class-based and object-oriented language and software-only platform, which is based on two main components. The first component is the Java Application Programming Interface (API), which is a library of Java command lines. The second component is the Java Virtual Machine (JVM) that interprets Java code into machine language. These allow the program built in the Java platform, to be independent from the underlying hardware and run on multiple platforms (Windows, MacOS, UNIX) without any notable modifications.

The apps were developed in the Java SE development kit 8 (update 31) using the NetBeans IDE (ver. 8.0.2), and run with Java 8 (update 45).

The Intel® Galileo Graphical User Interface – IGG

This is the main app, which allows the user to compile code and download it to the Intel® Galileo, and view the data received from the Galileo in a graphical form.

The app's window contains 3 tabs:

- The **Simulation** tab
- The **Measurements** tab
- The **Monitor** tab

Several features of the app include connecting to the Galileo through a network connection (by establishing a server connection), exchanging data with the Galileo while its code is running, and showing the received data on a plot. These features require the code to have a frame, which will handle them. For that purpose, templates of code are shown in the **Simulation** and **Measurements** tabs. For the app to run properly, it's highly advised to copy the templates to a new file and then insert the user's necessary code. The templates also show where the user should insert the code. Both sketch templates appear in Appendix D.

The Simulation tab

This tab offers the user to download a simulation sketch to the Galileo, with a trigger wave and a set of parameters. The user can also change the value of the parameters and the trigger wave's fields, as well as its shape, while the simulation is running. This allows control over the simulation without the need to re-compile and download a modified code to the sketch.

In the initial stage, the Simulation tab offers loading a code and showing it in a text editor. In this stage the window (shown in *Figure 10*) is built from several panels and components. The components are:

1. The Trigger Panel (marked by the black rectangle in *Figure 10*).
2. The Code Loading Panel (marked by the red rectangle in *Figure 10*).
3. The Parameters Panel (marked by the blue rectangle in *Figure 10*).
4. The Code Text Area (marked by the yellow rectangle in *Figure 10*).
5. The "Start" Button (found in the lower left corner of the window in *Figure 10*).

When the user starts the simulation, the code text area is replaced with a graph panel which displays the data received from the Galileo. In this stage the window (shown in *Figure 11*) contains all the components from the initial stage except the Code Text Area, in addition to 2 new components:

6. The "Update" Button (found in the lower left corner in a yellow rectangle in *Figure 11*).
7. The Graph Panel (marked by the red rectangle in *Figure 11*).

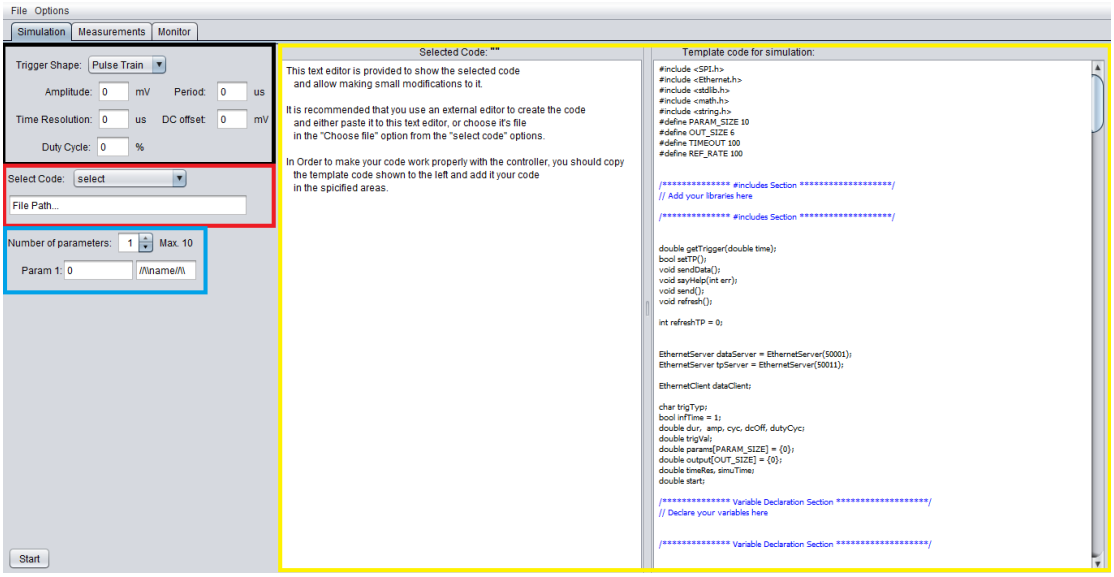

Figure 10: The initial stage of the Simulation tab, offering to load a code and edit it in the text editor

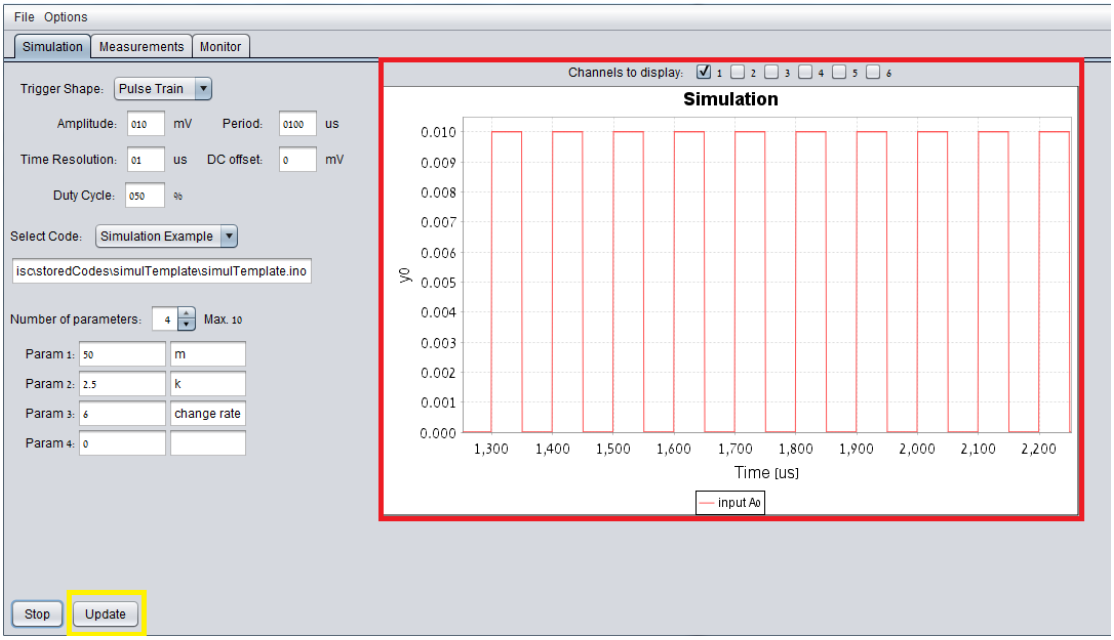

Figure 11: The app's Simulation tab after the simulation started. A plot displays the data received from the Galileo, and the user can update the trigger or parameters while the simulation is running

The Measurements Tab

This tab offers the user to download a model sketch to the Galileo, with a set of parameters. The user can also change the value of the parameters, while the code is running. This allows control over the code without the need to re-compile and download a modified code to the sketch.

The Measurements tab is very similar to the Simulation tab except several differences, which are due to the fact that the simulation session wasn't built for using external signals. This is contrary to the measurements session which supports the use of external signals. The differences which rise are:

- The measurements session supports the use of the Galileo's analog input and digital output pins in the code, while the simulation session doesn't support their use. This is due to a timing difference between the 2 sessions – while in the measurements session, the data is sent as soon as it is calculated, in the simulation session, there is a delay of approximately 20 milliseconds between each data sent. When one of the pins (analog or digital or both) will be used in the simulation session with a signal whose time period is lower than 20 millisecond, the code will not work as expected. Note that this difference lies in the templates written for each session, so by modifying the templates, you can eliminate the above delay, however the graphical display in the plot may not work as it should. With that said, the data can still be saved and processed with an external program.
- The trigger interface exists only in the simulation session. For the measurements, an external signal can be used as trigger.
- The time in the Measurements session is calculated using the system time on the Galileo, while in the Simulation session, it is calculated using the time resolution specified by the user.

In the initial stage, the Measurements tab offers loading a code and showing it in a text editor. In this stage the window (shown in *Figure 12*) is built from several panels and components. The components are:

2. The Code Loading Panel (marked by the red rectangle in *Figure 12*).
3. The Parameters Panel (marked by the yellow rectangle in *Figure 12*).
4. The Code Text Area (marked by the black rectangle in *Figure 12*).
5. The "Start" Button (found in the lower left corner of the window in *Figure 12*).

When the user starts the session, the code text area is replaced with a graph panel which displays the data received from the Galileo. In this stage the window (shown in *Figure 13*), contains all the components from the initial stage except the Code Text Area, in addition to 2 new components:

6. The "Update" Button (found in the lower left corner in a black rectangle in *Figure 13*).
7. The Graph Panel (marked by the red rectangle in *Figure 13*).

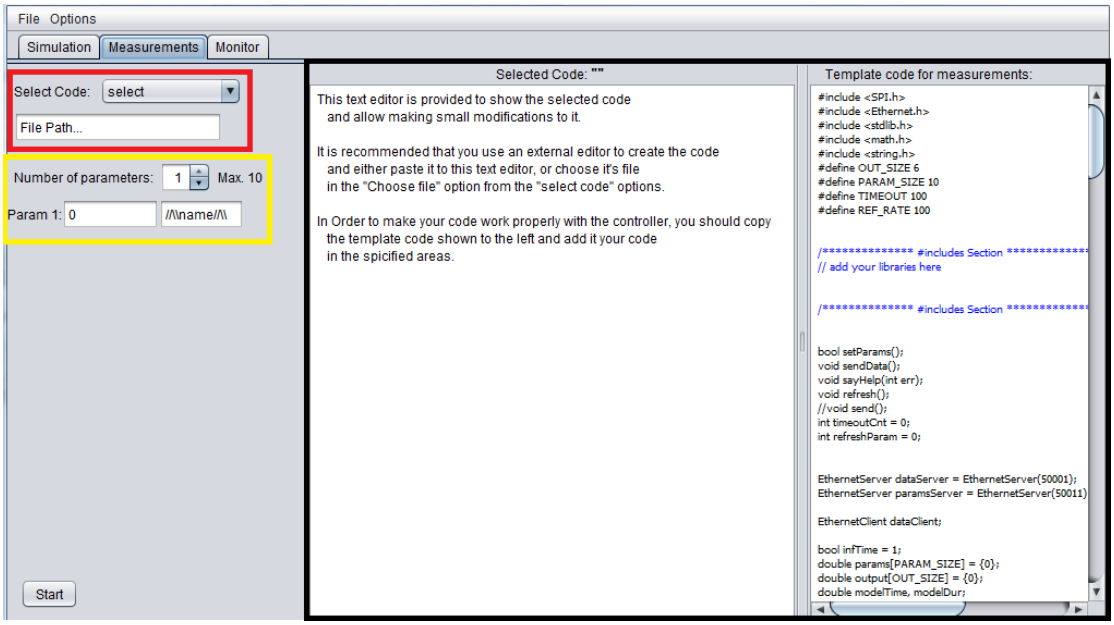

Figure 12: The initial stage of the Measurements tab, offering to load a code and edit it in the text editor

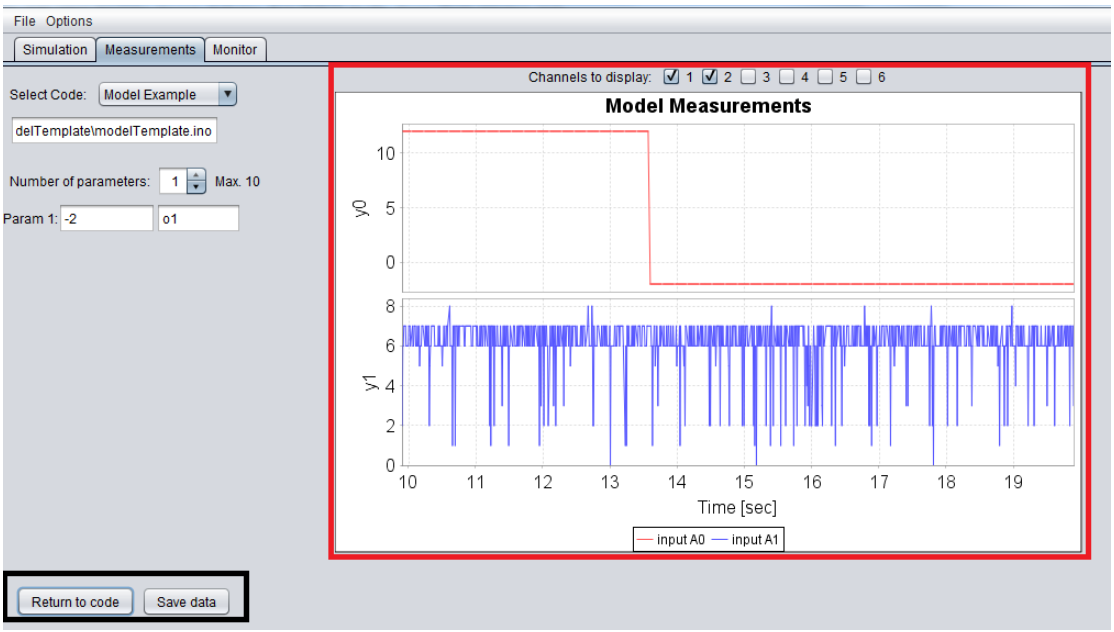

Figure 13: The app's Measurements tab after the session ran and stopped. A graph displays the data received from the Galileo, and the user can either save the data from the session, or return to the code editing.

The components of the Simulation and Measurements Tabs

The Trigger Panel (only in the Simulation tab)

This panel allows the user to choose from a set of trigger wave shapes and specify their parameters.
The available trigger shapes are:

- Pulse train waveform _-_-_-_-
- Sine waveform ~~~~~~
- Triangular waveform /\/\/\/\/\
- Ramp waveform /|/|/|/|/|

The parameters are:

- Amplitude – measured in units of mV .
- Period – measured in units of μs .
- Simulation Time Resolution – measured in units of μs .
- DC offset – measured in units of mV .
- Duty Cycle – measured in %, and available only in the case of the Pulse Train wave.

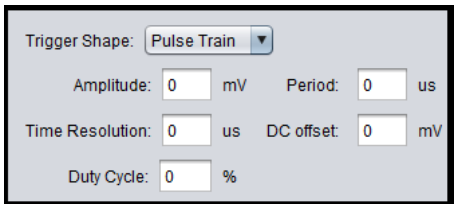

Figure 14: The Trigger Panel

Implementation – all the labels in the panel are javax.swing.JLabel components which allow showing a single-line text as a label. The choosing of the trigger shape was made by the javax.swing.JComboBox component which shows a menu to choose from. The trigger fields are javax.swing.JTextField components which allow writing a single-line text containing the value of the field. All the components in this panel were placed visually using the NetBeans IDE GUI builder.
When the app needs to send the trigger data to the Galileo, it first reads all the trigger fields strings, and verifies that these strings are valid values, i.e. can be parsed as variables of type Double. Afterwards, it builds a String of data to send to the Galileo. The String is of the following form:

"O-timeRes- -0- -trigType- -amp- -cyc- -dcOff- dutyCyc- "

The first character is the letter 'O' which tells the Galileo code that the following line that it will read is a "trigger line". After the letter 'O' comes the Time Resolution value. Then, separated by whitespace, comes the simulation duration value. When building the app, the initial thought was to have an additional field for the simulation duration, however this was later canceled due to implementation difficulty, and the value of the duration was set to constant 0, which tells the Galileo to run the simulation continuously. After the duration value, comes the Trigger Type of the waveform, which is a character. Based on the index of the selected shape in the JComboBox, the Trigger Type value is set to the appropriate character: P for **P**ulse train, S for **S**ine wave, T for **T**riangular wave and R for **R**amp wave. The next field is the Amplitude value field, followed by the Time Period, the DC offset, and the Duty Cycle, all read directly from the JTextField components.
The String must end with a whitespace, otherwise a problem will occur when parsing the data received from the PC.
Note that the values of the String are separated by whitespaces except the first char 'O' and the Time Resolution value.

The following example tries to make the trigger line setting clearer.
Suppose the Trigger shape selected is a Pulse Train, and the fields are set to the following values:

Amplitude=50.4 Period=30 Time Resolution=2.3 DC offset=0.5 Duty Cycle=64

Then the trigger line sent will be: `O2.3 0 P 50.4 30 0.5 64`

For any other shape than the Pulse Train, the duty cycle field can take any value, but will be unused by the code on the Galileo.

If any of the specified trigger fields isn't a valid value, the app will pop-up a message pointing to the invalid fields, and will terminate the data-sending process.

The Code Loading Panel

This panel offers the user to select a code from a stored list of codes, or load a code from a file. After the code is selected, its path is shown to the user, its name is shown above the text editor and its content is written to the text editor.

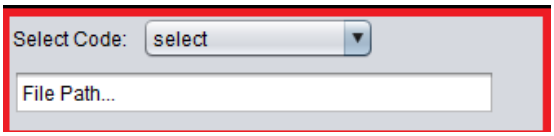

Figure 15: The Code Loading Panel

Implementation – The panel contains three components: a javax.swing.JLabel component; a javax.swing.JComboBox component, which shows a menu for selecting a code; and a javax.swing.JTextField, for showing the path of the loaded code.
The JComboBox component's menu is comprised of two hard-coded Strings - "select" and "choose from file…", and additional codes stored in the app's database. The "select" option is an idle one with no code behind it, and it's simply used as the initial selection of the menu. When the user selects the "choose from file…" option, a File Browser window pops up through which the user finds and selects the wanted code file. This is done by using the javax.swing.JFileChooser component that was built for the purpose of loading and saving files (in this instance – a "showOpenDialog" method is used). After the user has selected the file, the app checks the file's extension – if the extension is not ".ino" (which is supported by the Arduino IDE), then the user is notified and the app cancels the loading, returning to initial state. Otherwise, the file is read and loaded to the text editor. When the user selects a code which is stored in the app's database, the app loads the code from its library of stored codes, displays the code's label and path and updates the text editor to show the code. The stored codes' labels are added to the selection menu at the startup of the app, or when the user added a new code to the stored database.

The Parameters Panel

This panel shows a set of parameters which can be used in the code. When the code is already running, the user can change the value of one or more of the parameters and then update the session. A maximum of 10 such different parameters are available to use in the code. When the parameters are visible, their initial value is 0 and they can be set to any value by the user. When the user chooses to decrease the number of parameters, the parameters which are not used are hidden and their value resets to 0. When the app needs to send the parameters' values to the Galileo, the app first checks that all the values are valid numbers, and can be parsed as variables of type Double. If any of the parameters is not a valid value, the user is notified by a pop up window which points to the invalid parameter, and the app doesn't send the data. The data-sending is a String of the following form:

"K-par1- -par2- -par3- .... -par10- "

The first char of the String is the letter 'K' which tells the Galileo that the line that is read is the "parameters line". After 'K', come all the other parameters' values, separated by whitespace. Parameters not set by the user, are automatically set to 0.
Note that there isn't any whitespace between the letter 'K' and the first parameter value.

For convenience, an empty text field is placed near each of the parameters for the purpose of labeling. This allows the user to label the parameters and avoid confusing between parameters. The text fields have no other purpose, and the assigned labels are neither processed by the app, nor sent to the Galileo.

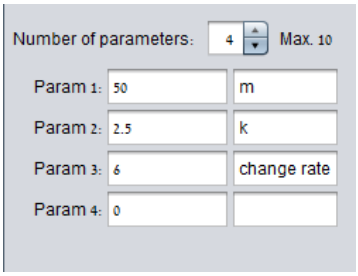

Figure 16: The Parameters Panel
with 4 visible parameters with set values and labels

Implementation – the panel contains: 2 javax.swing.JLabel components, to display the Strings "Number of parameters:" and "Max. 10"; a javax.swing.JComboBox which shows the current number of parameters; and a sub-panel that contains an array of 10 sub-units, which have 3 components each: a JLabel with the String "Param X" where $X = 1,2,3...,10$; a JTextField showing the value of the x-th parameter; and a JTextField for labeling the parameters.

The Code Text Area

This panel displays two text windows – one is an editable text area displaying the code selected by the user (the left text area), and the other is a non-editable text area that shows the Simulation or Measurements Template code (the right text area). The editable text area also allows the user to write a new code by copying the template code and adding additional code in the painted areas. Then, the user can save it, compile it and download the compiled code to the Galileo board.
To prevent confusion, two labels appear above the text areas – a left label showing the loaded code name (or "" if a new code is created) and a right label showing (constantly) "Template code for simulation/measurements".

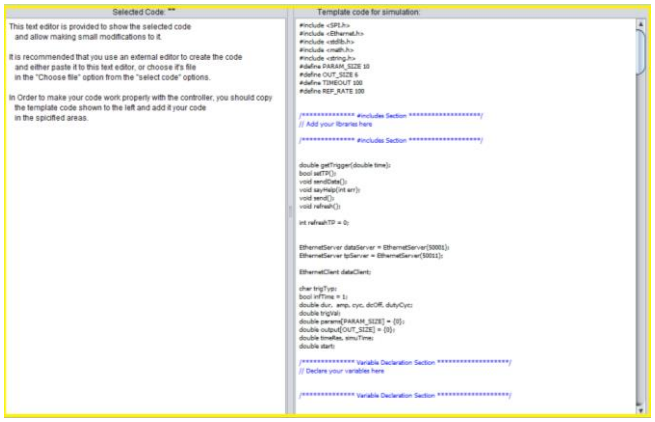

Figure 17: The Code Text Area

Implementation – the panel contains: 3 javax.swing.JLabel components which are the titles of the text areas (one for the template and two for the user's code), a JTextPane component which displays the template code and a javax.swing.JTextArea which displays the user's code. Each of the text components is placed inside a javax.swing.JScrollPane, to allow scrolling when the text area is bigger than the app's window. For each of the text areas (template and user's code), the Text component (inside a JScrollPane) was put inside a javax.swing.JPanel together with its labels. Then, both JPanels were placed inside a javax.swing.JSplitPane component, which allows a moveable border between the two JPanels.

Since the app allows modifying the user's code inside the JTextArea, the app needs to "know" whether the code was modified or not. For that purpose a Boolean flag is used together with a java.awt.event.DocumentListener interface. Each time the code text is modified, the Boolean flag is set to "True". Then, before the app will start the session, it will check the flag and if it's true, the app will offer to save the changes before starting. Otherwise the app will use the original code. This is due to the fact that when the app compiles and downloads the code, it uses the file itself instead of the actual code written in the text area.

The "Start" Button

When this button is pressed, the app starts the process of compiling the code, downloading it, and replacing the code text areas with the Graph Panel. Before the compilation, the app checks if the code loaded to the text area was modified by the user. If it was, the app offers the user to save to changes. Afterwards, the app calls a Bash script which compiles the code to an ".elf" source file, downloads it to the Galileo, and then restarts the sketch process on the Galileo to run the new sketch. The script's output is logged to a text file, and parsed by the app in order to see if any errors occurred during the compilation or the download. If some errors did occur, the app will notify the user and then show the output log for the user to understand the error. If no errors occurred, the app continues to start the server connection and the data exchange with the Galileo. If the Galileo isn't connected to the PC through the network (Ethernet), the user will be notified, and the process will be aborted, with app returning to initial stage.

If the code compilation and download was successful, and the network connection is running without errors, the plot starts to add data received from the Galileo, and the "Start" button changes its label to "Stop". In addition, an "Update" button becomes visible, allowing the user to update the parameters (or also trigger if the session is Simulation) on the running code.

When the session is running and the user clicks the button (now the "Stop" button), the plot update is stopped, the server connection is closed, and the "Stop" button changes its label to "Return to code". That is done to allow the user to view the plot before closing it or moving to another tab. In addition, the "Update" button changes its label to "Save" to offer the option of saving the data from the session.

When the user clicks the button again (now "Return to code" button), the app checks if the user saved the last session, and if not offers the user to save. Then, the button changes its label back to "Start", the "Save" button is changed back to "Update" and becomes hidden and the plot panel is replaced with the code text area, i.e. the app returns to the initial stage.

Implementation – the button is a javax.swing.JButton component and can have one of 3 labels, based on the current stage of the app. When the button is clicked, an ActionEvent is fired and a method responsible for managing the session is invoked. If the label is "Start", the method performs the following operations:

- Checking whether the code was modified in the text area and if it was, then offering the user to save the changes.
- Initiating the server connection interface (if it didn't exist previously) with the host IP address.
- Downloading the sketch code to the Galileo (after compiling it).
- Connecting to the Galileo via the server connection and sending the data (in Simulation – trigger and parameters, in Measurements – parameters). This task and the previous one are performed in the background while the user is asked to wait for them to finish.
- Verifying that no errors occurred in the previous two tasks. If there were any errors, the user is notified and the process is aborted.
- If there were no errors, the code area is replaced by the plot panel, the "Start" button changes its label to "Stop" and the "Update" button becomes visible.

If the label is "Stop", the method performs the following operations:

- Closing the server connection, and stopping the background update of the plot.
- Changing the label of the "Start" button to "Return to code".
- Changing the label of the "Update" button to "Save data".

If the label is “Return to code” the method performs the following operations:

- Checking the label of the “Update” button. If it is “Save data”, it means the user didn’t save the data from the session, so the app will offer this option. If the label is “Saved” it means the user has already saved the data.
- Changing the label of the “Saved”/”Save data” button back to “Update” and setting this button invisible.
- Clearing the data structure used for storing the data from the session in the app.
- Changing the label of the “Return to code” button back to “Start”.
- Replacing the Graph Panel with the Code Text Areas.

The “Update” Button

When the session is running, pressing the “Update” button will send the updated values of the parameters (and trigger in Simulation) to the Galileo. This allows a partial update of session without the need to re-compile and download the code to the board. The update refers to the values of the parameters (and trigger in Simulation) only, and a change in the code itself requires to restart the whole session process.

When the session has stopped, the button changes its label to “Save data”, to allow the user to save the session data. Once the user has saved the data, the label is changed to “Saved” to notify the user, and clicking the button again will also open the save dialog.

When the session is over, and the app returns to the initial stage, the button is hidden, and becomes visible again when the session re-starts.

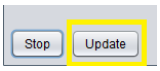

Figure 18: The "Update" button when the session is running.

Implementation – The button is a simple javax.swing.JButton component, and can have one of three labels, based on the current stage. When the button is clicked, an ActionEvent is fired and a method is invoked.

When the label is “Update” the method calls another method which reads the values of the parameters (and the trigger in Simulation) to an array of Strings, and then sends these Strings to the Galileo. If a problem occurred during the update process, the user is notified and the session process is aborted.

When the label is “Save data” or “Saved” the method calls another method which manages the save dialog, and saves the data in a tab-separated file, with 7 columns – first column is for the time of the row, and the other 6 are for the output values received from the Galileo at the given time. The file’s location is specified by the user. If the save process was successful, the label of the button changes to “Saved” (in the case where it was already “Saved”, then after the save is done, nothing happens). If problems occur during the save process, the user is notified.

The Graph Panel

This panel appears after the startup process of the session. It replaces the Code Text Area and contains a plot and a panel for selecting which outputs to display on the graph. When the session is running, the plot constantly receives data from the Galileo and displays it. Also, a data structure is used (a queue) to store all the points received from the Galileo. The plot displays the data in a fixed range on the x-axis. New points are added on the right and old points disappear to the left but are still included in the plot. When the number of included points in the plot increases, so does the delay needed to add a new point. Therefore, a limited number of included points is defined and when it's reached, a certain number of old points is deleted from the plot, thus keeping the delay low.

In the initial stage only the first output is displayed, however the user can select the channels to be displayed using the panel above the plot.

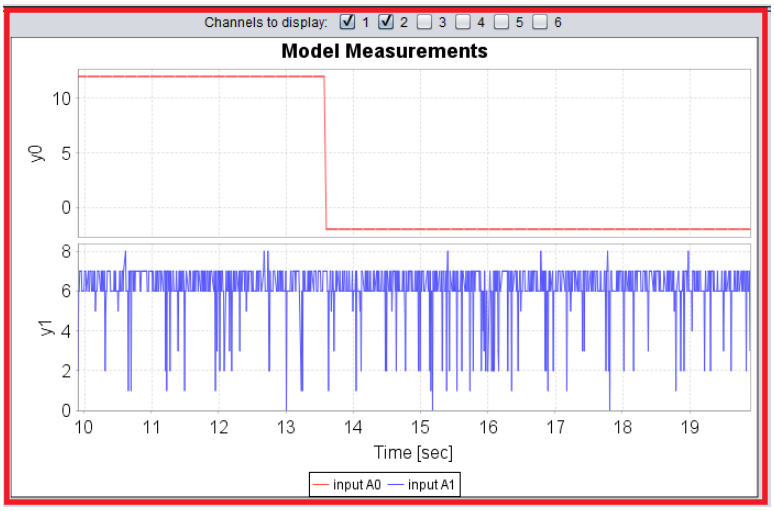

Figure 19: The Graph Panel displaying the first output channel in the plot.

Implementation – The panel contains two sub-panels.
The first panel contains the checkboxes for selecting the channels to display, which are 6 javax.swing.JCheckBox components, one for each of the 6 available outputs. There is an additional JLabel component with the label: "Channels to display". When a checkbox is checked or unchecked, an ActionEvent is fired and caught by an ActionListener class which handles the update process of the number of channels in the simulation and measurements tabs. The class receives the number of the channel and whether it is checked or unchecked, and subsequently notifies a different class which updates the plot accordingly. Then the data-update process of the plot itself, is restarted with the updated number of channels.
The second panel contains the plot, which can display up to 6 subplots, one for each output. The plot is a component called RealTimeGraph_SimuModel, which handles the plots in the Simulation and measurements tabs. This component is based on the JFreeChart library which offers a graphical display of different data in a variety of shapes. In addition to building the plot itself (with axes' types, subplots etc.), the component also uses 3 objects:

- A data structure (a queue) to store all the data received in the simulation/measurements sessions. This allows the user to save the data after the session is complete.
- An instance of a background update class which adds the received data to the plot without freezing the app, meanwhile allowing the user to use the app.
- An instance of a ServerInterface class which handles the server connection to the Galileo, as well as the data exchange between the PC and the Galileo.

The background update class uses the ServerInterface class to read the data from the Galileo, which is a String that contains 7 variables, separated by whitespace. The first variable is the time calculated by the code[8], and the other 6 are the output variables. After reading the data, the class parses it into an array of variables of type Double and adds them to the plot. Each time a point is added, the class checks whether the number of already placed points on the plots has reached a pre-defined limit. If it did, then a pre-defined number of old points is deleted from the plot.
This class is also responsible for calling the ServerInterface class to start the connection to the Galileo, and send data when an update is required.

[8] In the Simulation session the time is computed from the start using the time resolution specified by the user. In the Measurements session the time is computed using the subtraction of the initial system time from the current time at a given point.

The Monitor Tab and its Components

This tab offers the user to monitor the analog inputs of the Galileo. The user can choose to view only one input or several inputs, or all of them. The app also offers the user to stop the session, save its data to a file, and start a new session.

While the Simulation and Measurements sessions require both the network cord and the USB cord to be connected to the Galileo, the Monitor session requires only the network cord to be connected, since it doesn't download any files to the Galileo, but simply turns on an already existing sketch on the Galileo.

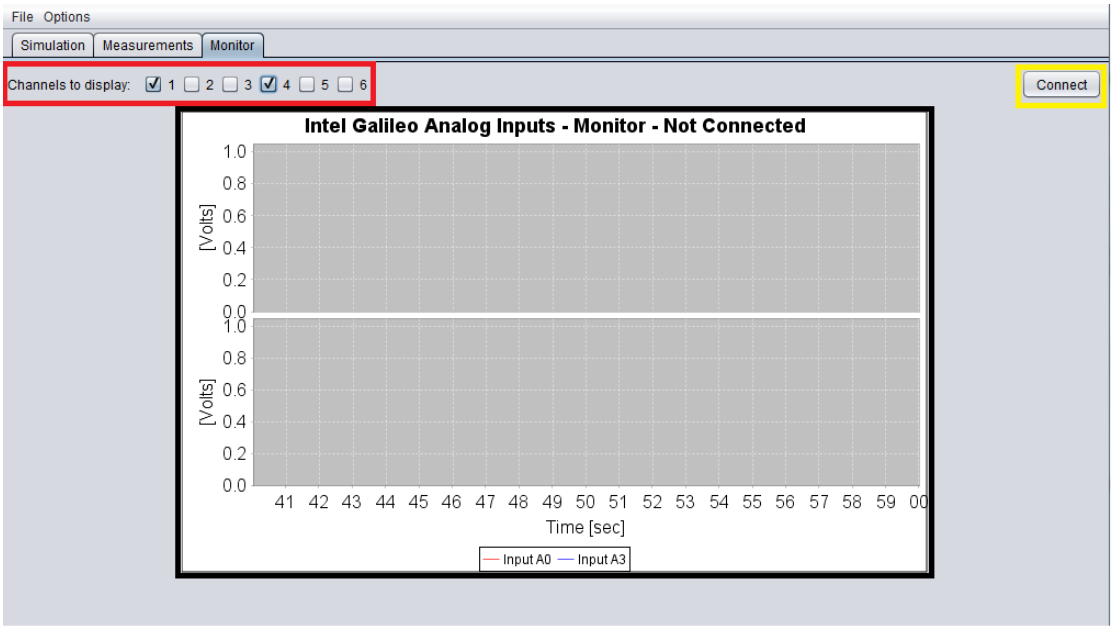

Figure 20: The Monitor tab, offering the user to monitor the analog inputs of the Galileo, and also the data monitored to a file

Figure 21: The Monitor tab after the session was stopped. The user is offered to save the data and start another session.

The components the Monitor tab contains are:

1. The Channels Panel (marked by the red rectangle in Figure 20).
2. The Graph Panel (marked by the black rectangle in Figure 20).
3. The "Connect" Button (marked by the yellow rectangle in Figure 20).
4. The "Save" Button (marked by the blue rectangle in Figure 21).

The Channels Panel

This panel is very similar to those of the Simulation and Measurements tabs. It allows the user to select which inputs will be displayed on the plot. When a checkbox is checked or unchecked, an ActionEvent is fired and a method, which updates the number of subplots displayed, is invoked. In addition, the background update of the plot is restarted with the updated channels to display.

Figure 22: The Channels Panel

Implementation – this panel contains the checkboxes for selecting the channels to display, which are 6 javax.swing.JCheckBox components, one for each of the 6 available analog inputs. There is an additional JLabel component with the label: "Channels to display". When a checkbox is checked or unchecked, an ActionEvent is fired and method which updates the number of channels displayed is invoked. The method receives the number of the channel and whether it is checked or unchecked, and subsequently notifies a different class which updates the plot accordingly. Then, a String is sent to the Galileo, in the following form:

"X X X X X X "

Where X can have the value of 1 or 0. The index of X in the String represents the analog input with the same index. The value 1 tells the Galileo to measure the analog input, and 0 tells it to skip that analog input (the value is set to 0). E.g. if the user wants only inputs A0, A4 and A5 to be measured, the data sent is:

1 0 0 0 1 1

Finally, the data-update process of the plot itself, is restarted with the updated number of channels. If problems occur during the restart, the user is notified and the Monitor session is stopped, allowing the user to fix the problem and connect again.

The Graph Panel

This panel contains the plot of the tab. When the session is running, the app constantly reads data received from the Galileo and adds it to the plot. Also, a data structure is used (a queue) to store all the points received from the Galileo. The plot displays the data in a fixed range on the x-axis. New points are added on the right and old points disappear to the left but are still included in the plot. When the number of included points in the plot increases, so does the delay needed to add a new point. Therefore, a limited number of included points is defined and when it's reached, a certain number of old points is deleted from the plot, thus keeping the delay low.

When the user switches tabs to the Monitor tab for the first time, the session is started automatically with the first output displayed on the plot. The user can then select other inputs to display, or stop the session and save the data.

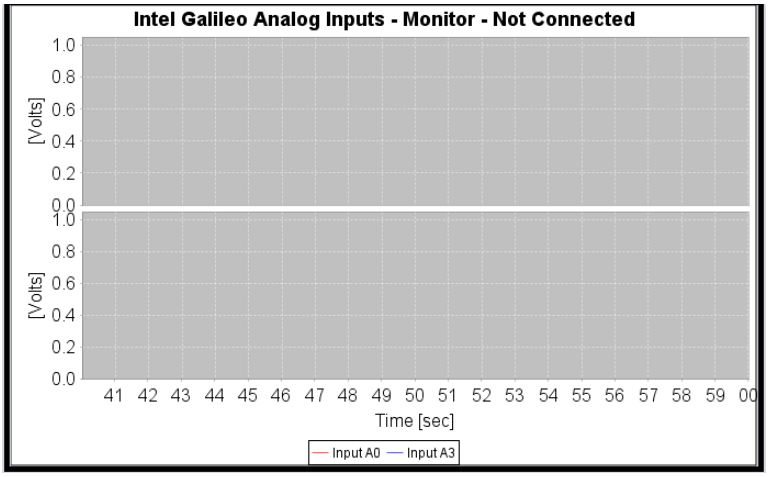

Figure 23: The Graph Panel of the Monitor tab

Implementation – this panel contains the plot, which can display up to 6 subplots, one for each analog input. The plot is a component called RealTimeGraph_Monitor, which handles the plot in the Monitor tab. This component is similar to that of the Simulation and Measurements tabs and it's also based on the JFreeChart library. The difference between these 2 components is that the ports to which the Galileo and PC are connected are different from those of the Simulation/Measurements sessions. Also, the data sent in the Monitor session is different. Furthermore, since the received data is the analog inputs, and has a value between 0 and 4096 (the ADC output), the app converts it to Voltage before displaying on the plot. In the Simulation and Measurements sessions, no conversion is made, since the app doesn't know which output is an analog input. In addition to building the plot itself (with axes' types, subplots etc.), the component also uses 3 objects:

- A data structure (a queue) to store all the data received in the Monitor session. This allows the user to save the data after the session is complete.
- An instance of a background update class which adds the received data to the plot without freezing the app, meanwhile allowing the user to use the app.
- An instance of a ServerInterafce class which handles the server connection to the Galileo, as well as the data exchange between the PC and the Galileo.

The background update class uses the ServerInterface class to read the data sent from the Galileo, which is a String that contains 6 variables, which are the analog inputs, separated by whitespace. After reading the data, the class parses it into an array of variables of type Double and adds them to the plot. Each time a point is added, the class checks whether the number of already placed points on the plots has reached a pre-defined limit. If it did, then a pre-defined number of old points is deleted from the plot. This class is also responsible for calling the ServerInterface class to start the connection to the Galileo, and send data when an update is required.

The "Connect" Button

When this button is pressed, the app starts the Monitor session. First, it runs a Bash script which replaces the existing running sketch on the Galileo with the already stored Monitor sketch, and then restarts the sketch process on the Galileo by simply killing it and its parent process. This causes the parent's parent process to restart the parent process which subsequently restarts the sketch process. If any problems occur during the startup of the session, the user is notified and session is aborted. If no errors occurred, the app continues to start the server connection and the data exchange with the Galileo.
If the startup was successful, and the network connection is running without errors, the plot starts to add data received from the Galileo, and the "Connect" button changes its label to "Stop".

When the session is running and the user clicks the button (now the "Stop" button), the plot update is stopped, the server connection is closed, and the "Stop" button changes its label back to "Connect". In addition, a "Save" button becomes visible allowing the option of saving the data from the session.

Implementation – the button is a javax.swing.JButton component and can have one of 2 labels, based on the current stage of the app. When the button is clicked, an ActionEvent is fired and a method responsible for managing the session is invoked. If the label is "Connect", the method performs the following operations:

- Offering the user to save the data from the previous session before starting a new one.
- Clearing the data structure used for storing the data of the previous session, as well as the objects in which the plot stores its displayed data.
- Making the "Save" button invisible.
- Initiating the server connection interface (if it didn't exist previously) with the host IP address.
- Replacing the current sketch file on the Galileo with the Monitor sketch, and restarting the sketch process.
- Connecting to the Galileo via the server connection and sending data describing which analog inputs should be displayed. This task and the previous one are performed in the background while the user is asked to wait for them to finish.
- Verifying that no errors occurred in the previous two tasks. If there were any errors, the user is notified and the process is aborted.
- If there were no errors, the "Connect" button changes its label to "Stop".

If the label is "Stop", the method performs the following operations:

- Changing the label of the "Stop" button to "Connect".
- Making the "Save" visible.
- Closing the server connection, and stopping the background update of the plot.

The "Save" Button

The "Save" button appears only when the session is not running, and offers the user to save the data from the session to a file in a location specified by the user. Once the user has saved the data, the label is changed to "Saved" to notify the user, and clicking the button again will also open the save dialog. When the session starts over, the button is hidden, and becomes visible again when the session is over.

Implementation – The button is a simple javax.swing.JButton component, and can have one of two labels, based on the current stage. When the button is clicked, an ActionEvent is fired and a method is invoked.

When the label is "Save" or "Saved" the method calls another method which manages the save dialog, and saves the data in a tab-separated file, with 7 columns – first column is for the time of the given row, and the other 6 are the analog input values in [Volts]. The file's location is specified by the user. If the save process was successful, the label of the button changes to "Saved" (in the case where it was already "Saved", then after the save is done, nothing happens). If problems occur during the save process, the user is notified.

The “File” and “Options” Menus

Two menus appear in the top left corner of the app’s window:

- The “File” Menu which offers the “Exit” option.
- The “Options” Menu which offers the “Add new code to stored codes…” option.

The “Exit” option

When this option is selected, the app offers to save unsaved changes and sessions and then exit.
Note that if a session is still running, the app won’t offer to stop and save it, and will just exit, with the data collected from the session lost.

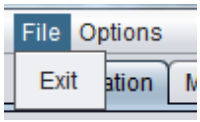

Figure 24: The "Exit" option found in the "File" menu.

Implementation – the “Exit” option is a javax.swing.JMenuItem component, and when it’s selected, an ActionEvent is fired and invokes a method. That method checks for unsaved changes made to any of the 2 codes loaded in the Simulation and Measurements tabs, as well as unsaved data from stopped sessions in any of the 3 tabs. If there is an unsaved change or data, the user is offered to save it. If the User cancels the process of saving, the method won’t exit the app. After the user choice is made and the data needed to be saved, is indeed saved, then the method will exit the app. The same method is invoked when the user closes the window.

The “Add new code to stored codes…” option

This option allows the user to store a code with its own label in the app’s database. The stored code can be in the Simulation tab, the Measurements tab, or both. After completing the process, the newly stored code’s label will appear in the “Select Code” menu, allowing the user to quickly load the code without searching for it on the PC. All stored codes are copied to the app’s “.\src\Misc\storedCodes” folder, so if the user wants to modify a stored code, he can either load it to the text area and modify it, or simply go to its location and use an external editor to modify it. In addition to copying the code to the app’s folder, the label and location of the code are stored on a list which is then saved to an external file. This allows the app to “remember” the list of stored codes, so the next time it is run, it will read the file and load the stored codes to the selection code menu.

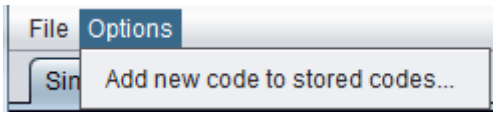

Figure 25: The "Add new code to stored codes..." option found in the "Options" menu.

Implementation – like the “Exit” option, this option is also a JMenuItem component, and when it’s selected, an ActionEvent is fired and invokes a method, which handles the process of storing the code. This method performs several operations:

- Asking the user in which tab he would like the code to be stored – the Simulation tab, the Measurements tab, or both.
- Opening the load dialog and allowing the user to locate the code in the system.
- Notifying the user if a code with the same name is already stored, and offering the user to overwrite the stored code. If the user chooses to overwrite the code, then the existing code label and path will be deleted from the list of stored codes. Also, when the code will be copied it will replace the old code. If the user doesn’t choose to overwrite the process is aborted.
- Copying the code file from the specified location to the app’s folder.
- Asking the user to label the code, which will “represent” the code in the code-selection-menu.
- Storing the code’s label and location in the app’s list.
- Updating the code-selection-menu to include the newly added code.
- Saving the app’s list of stored codes to a file.

When the user chooses to store the code in both tabs, the same operations (described above) are performed to both lists of stored codes – the Simulation list and the Measurements list.

The Interface Setup Graphical User Interface – sIGG

This app sets up the following objects, used by the main app (the IGG):

- The Preference File used by the main app (the IGG), which includes the IP of the Galileo, the path to the Arduino IDE, the COM port number to which the Galileo is connected and the type of the board for the code compilation (can either be Intel® Galileo or Intel® Galileo Gen2).
- The Ethernet interface on the Galileo, which will allow the data exchange through the network.
- The download of the Monitor Sketch, which will allow the IGG to simply turn it on instead of compiling and downloading it each time it's required.

In addition to the preferences required for the IGG, the sIGG requires two fields to be filled:

a. The path of the IGG – after the preferences are set, a file will be written and saved in the directory of the IGG (which is specified) containing the above preferences.
b. The Netmask of the Galileo's IP address – used for setting up the Ethernet interface on the board.

When all fields are set and valid, the sIGG proceeds and performs the following operations:

1. Saving the preferences required by the IGG in its directory.
2. Generating a script for turning on the network interface on the board's startup. The script includes the IP and Netmask addresses of the board.
3. Downloading the network script (from 2) to the board, moving it to a directory and symbolically linking it to allow the board to run the script on startup. After the download is complete the script is run to start the network interface (saves the need to reboot the Galileo).
   Note: the instructions to set script to run on startup are specified in Appendix A.
4. Downloading the monitor sketch (which is found in the IGG directory) to the Galileo.

Note: the instructions regarding the download process of a file to the board are specified in Appendix B.

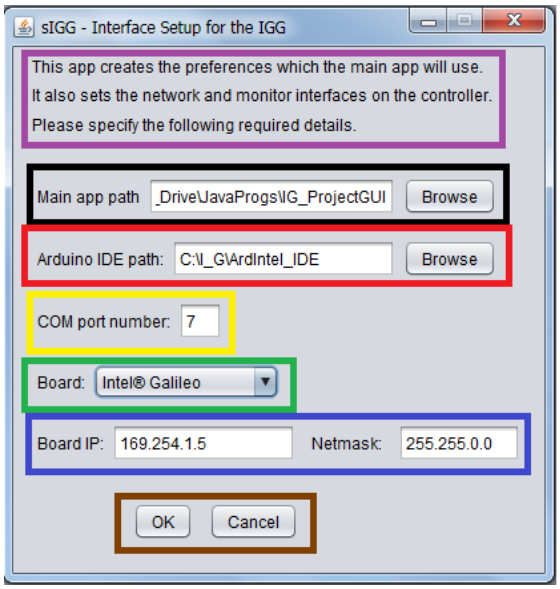

Figure 26: The sIGG app for setting up the preferences and interfaces for the main app - the IGG

The sIGG's Implementation

The sIGG is a single-window app with no tabs, and 7 sub-panels which are:

1. **The welcome text panel.**
   Marked by the purple rectangle in *Figure 26*, this panel contains a set of three JLabel components which specify the app's purpose and request the user to fill the necessary details (this panel serves no other purpose).
2. **The Main App Path Panel.**
   Marked by the black rectangle in *Figure 26*, this panel contains 3 components. The first is a JLabel component with the label: "Main app path". The second is a JTextField which contains the path specified by the user. The last component is a JButton with the label: "Browse" which when clicked opens a File Browser to allow the user to locate the main folder of the IGG (the main app).
3. **The Arduino IDE Panel.**
   Marked by the red rectangle in *Figure 26*, this panel contains 3 components. The first is a JLabel component with the label: "Arduino IDE path". The second is a JTextField which contains the path specified by the user. The last component is a JButton with the label: "Browse" which when clicked opens a File Browser to allow the user to locate the main folder of the Arduino IDE.
4. **The COM Port Panel.**
   Marked by the yellow rectangle in *Figure 26*, this panel contains 2 components. The first is a JLabel component with the label: "COM port number". The second is a JTextField which contains the number of the COM port to which the Galileo is connected.
5. **The Board Type Panel.**
   Marked by the green rectangle in *Figure 26*, this panel contains 2 components. The first is a JLabel component with the label: "Board". The second is a JComboBox which offers the user to choose between two options: "Intel® Galileo" and "Intel® Galileo Gen2". If the first choice is selected then the app writes to the preferences file the following line: "BOARD = Intel:i586-uclibc:izmir_fd"[9] and if the second choice is selected then the app writes the following line: "BOARD = Intel:i586-uclibc:izmir_fg".
6. **The Board's Network Panel.**
   Marked by the blue rectangle in *Figure 26*, this panel contains 4 components. The first is a JLabel component with the label: "Board IP". The second is a JTextField which contains the Galileo's IP, specified by the user. The Third component is a JLabel with the label: "Netmask". The last component is a JTextField which contains the Netmask of the IP address, also specified by the user.
7. **The Buttons Panel.**
   Marked by the brown rectangle in *Figure 26*, this panel contains 2 JButton components. The first is the "OK" button which starts the sIGG's work after the user specified all the necessary details, and the second is the "Cancel" button, which simply exits the app when clicked.

[9] The string written after the "BOARD = " string specifies the controller's type as used by the Arduino IDE. The type is parsed in the following form: VENDOR:ARCHiTECTURE:BOARD_NAME. In this case the manufacturer is Intel, the architecture is "i586-uclibc" and the board's name is: "izmir_fd" which refers to the Intel® Galileo. For more information, see: https://github.com/arduino/Arduino/blob/master/build/shared/manpage.adoc

When the “OK” button is hit, the app performs the following operations:

a. The board’s type is set to the choice made by the user.
b. All the other fields are validated. If one of the fields is invalid, e.g. the path doesn’t exist, or the specified COM port isn’t a number, or one of the IP and Netmask addresses is not a valid IP address[10], then the app notifies the user and points to the invalid fields, and then aborts the process.
c. If all the fields are valid, then a preferences file is saved in the IGG’s directory, in the “.\src\PrjGui” folder. Another preference file containing all the fields (including the Netmask and the IGG’s path) is saved in the sIGG’s folder, so next time the sIGG is run, it can load the preferences from this file, and allow the user to modify one of the fields, instead of filling up again all the fields.
d. The user is asked whether they would like to set up the network and monitor interfaces, and if they would, then they should connect the USB cord to the PC and the Galileo, and press “Yes”.

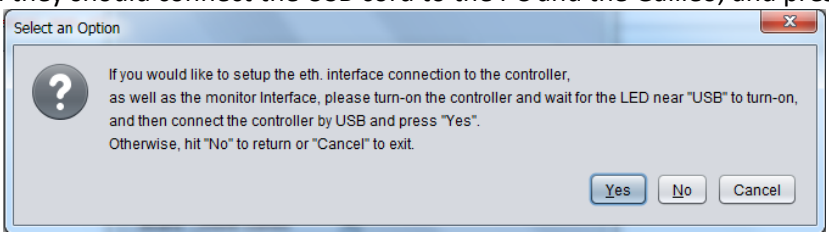

Figure 27: After saving the preferences file,
the app asks the user whether to setup the monitor and network interfaces too.

e. When the interface setup proceeds, the network script (responsible for turning on the Ethernet interface on startup) is generated, using a Batch script and the IP and Netmask addresses.[11]
f. After the network script is generated, the app downloads it to the controller, moves it to specific folder, and then symbolically links it to other folder, so the next time the controller starts up, it will run the script.
g. The App then downloads the monitor sketch to the controller, and moves it to the \sketch\ folder. This will allow the IGG to simply rename the file to “sketch” and restart the sketch process, all the while saving the need the compile and download it to the controller, each time it’s required.
h. If a problem occurred, the user is notified and the process aborts to allow the user to try again.
i. If no problems occurred the user is notified and then the app exits.

[10] In order to check the validity of the paths specified (the main app path and the Arduino IDE path), the app checks if the given path is an existing directory.
In order to check the validity of the COM port number, the app checks if it can parse it to a variable of type Integer.
In order to check the validity of the IP and Netmask addresses, the app calls a class which checks if the address is of the form: “X:X:X:X” with X being an Integer between 0 and 255.

[11] An example of the network script contents can be found in Appendix A.

Tutorial – how to use the IGG and sIGG

If you haven't yet set up the Galileo drivers on the PC, then follow the "Setting up the Galileo" section, and then proceed to the other sections. If the Galileo's drivers are already set up, you can skip the "Setting up the Galileo" section and jump to the "Using the Interface Setup GUI – the sIGG" section.

Setting up the Galileo

The following instructions help you set the Galileo drivers on the PC.

1. Download the "Arduino IDE 1.6.0 – Intel 1.0.4" program from the following link: http://www.intel.com/support/galileo/sb/CS-035101.htm (Intel® Galileo boards software downloads).
2. Connect the board (first plug in the power cord and then connect the USB cord to the board and PC). The OS will try to automatically install the drivers for the board, but it will fail, so you will have to supply the drivers manually (next instruction).
3. Go to Start -> type "run" at the search and hit Enter -> type "devmgmt.msc" in the opened window.
4. When the Device Manager Window opens, find the "Gadget Serial v2.4" device, under the "Other devices" tree. Right-click that device and select "Update Driver Software".

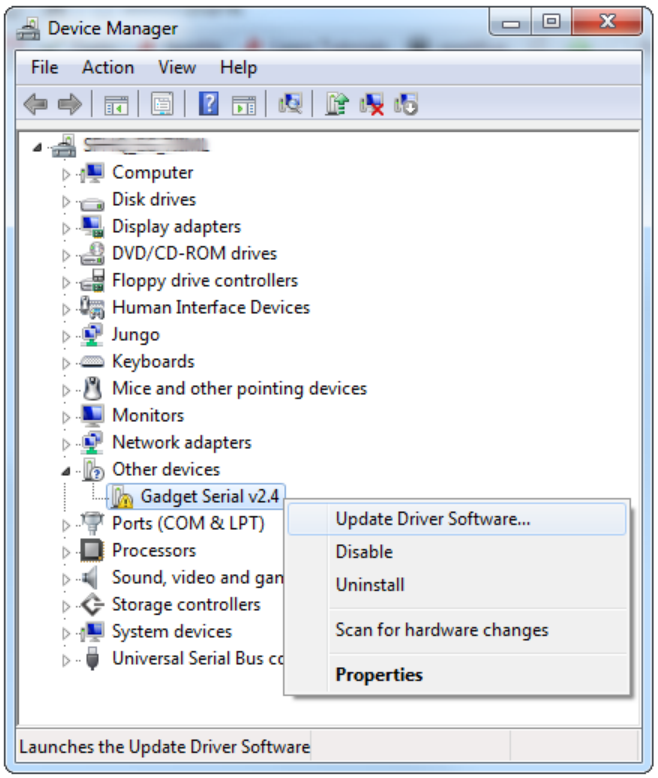

Figure 28: The Device Manager window.

5. On the first window that pops up hit "Browse my computer for driver software", and on the next page select "Browse…" and navigate to the "hardware/intel/i586-uclibc" folder within your Arduino Galileo IDE you downloaded in 1. Click "Next". (Make sure the "Include subfolders" option is checked. A similar window is shown in *Figure 28*, however the location specified in the window is incorrect.

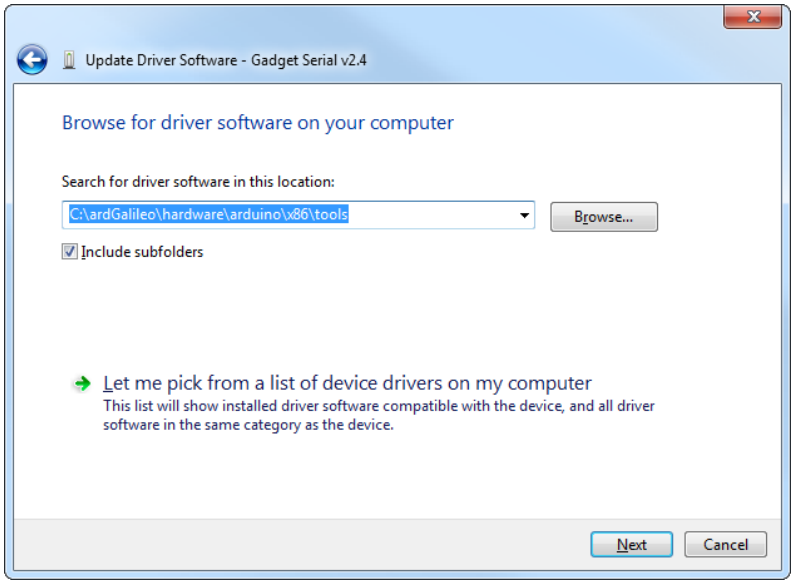

Figure 29: The manual update of the Galileo Drive, the specified location is incorrect and should be the "hardware/intel/i586-uclibc" folder within the Arduino Galileo IDE

6. Click "Install" on the next "windows security" window that pops up, and after a while the installation will be complete.
7. Look back at the Device Manager, this time under the "Ports" tree. There should be an entry for “Galileo (COM#)”. Remember which COM# your Galileo is assigned.
8. Now that the drivers are installed, you need to update the firmware of the Galileo. Download the "Intel Galileo Firmware Updater Tool 1.0.4" from the same link given in 1.
9. Before you run the tool, you need to reboot the Galileo, i.e. unplug the USB, then unplug the Power, wait a few seconds and make sure no SD card is inserted in the board before turning it on. Then plug in the power cord and finally the USB.
10. Run the Firmware updater tool, and make sure you select the correct COM port. The tool should recognize the Galileo connected to the PC, and then show its firmware version. If the version matches the one of the updater, no need to update, otherwise, press update, and then follow the process and wait until the firmware is updated. For a more detailed guide about updating the firmware follow the link: https://downloadmirror.intel.com/24748/eng/IntelGalileoFirmwareUpdaterUserGuide-1.0.4.pdf

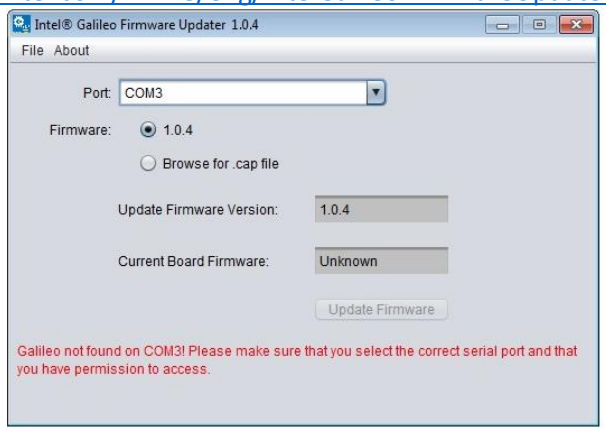

Figure 30: The Intel® Galileo Firmware updater.

11. After the firmware update is finished you should be able to use the setup app – the sIGG and then the main app – the IGG.

Using the Interface Setup GUI – the sIGG

1. **Specify the path of the Main App – the IGG**. You can either write the path manually to the provided field, or press “Browse” button next to it and using a File Chooser find the location of the IGG’s main directory.
2. **Specify the path of the Arduino IDE.** You can either write the path manually to the provided field, or press “Browse” button next to it and using a File Chooser find the location of the IDE’s main directory.
3. **Specify the COM port number.** This is the number which appears in the “Galileo (COM#)” device (instead of the #), which can be found in the Device Manager window (to see it, go to Start -> type "run" at the search and hit Enter -> type "devmgmt.msc" in the opened window. The device should be under the “Ports” tree).
4. **Choose the type of the controller board you are using.** It can be either Intel® Galileo, or Intel® Galileo Gen2.
5. **Write the IP address of the board.** If you connect the controller to a network with several users, you should specify the IP that will be assigned to the board. Note that this is a Static IP.
6. **Write the Netmask of the network to which the Galileo will be connected.**
7. **Press the “OK” button.** This will start the process of saving the preferences to a file in the IGG’s directory, as well as the downloading the monitor sketch and the network script.
8. **When asked if you would like to setup the network and monitor interfaces,** first plug in the power cord to the controller to turn it on. Then, plug in the USB cord to the controller (and make sure it’s connected to the right socket on the PC). And finally, **press the “Yes” button,** and wait until the sIGG has finished downloading the files to the controller.
9. **Done.** You can now use the IGG.

Using the Intel® Galileo GUI – The IGG

The Simulation Tab

Writing a simulation code using the given template –

In the provided template (found in the simulation tab under the “template code for simulation”) the areas for inserting the user’s code are marked with blue for #includes and variable declaration sections and red for the setup code and loop code sections. The Simulation template provides 4 variables for the user:

- A Double-type variable: “trigVal” which simulates the trigger wave based on the specified data determined by the user (sent from the PC) and the time of the current loop iteration.
- A Double-type array of 10 elements: “params [0...9]” which hold the values of the parameters specified by the user and sent from the PC.
- A Double-type array of 6 elements: “output [0...5]” which hold the values that will be sent to the PC to display on the plot.
- The time of the simulation: “simuTime” which is updated each loop-iteration using the time resolution value specified by the user.

The code language is similar to C/C++ (object oriented). For the available libraries and functions see the following site: https://www.arduino.cc/en/Reference/HomePage#

How to use the Simulation tab

This section will help you load a code to the text area and then download it to the Galileo. Also it will show you how to change a parameter or trigger and update the simulation:

1. Run the IGG.jar. After a while the app's window shall appear with the simulation tab as the current tab.
2. To load a code from the system, at the code-selection menu, select the "Choose from file…"

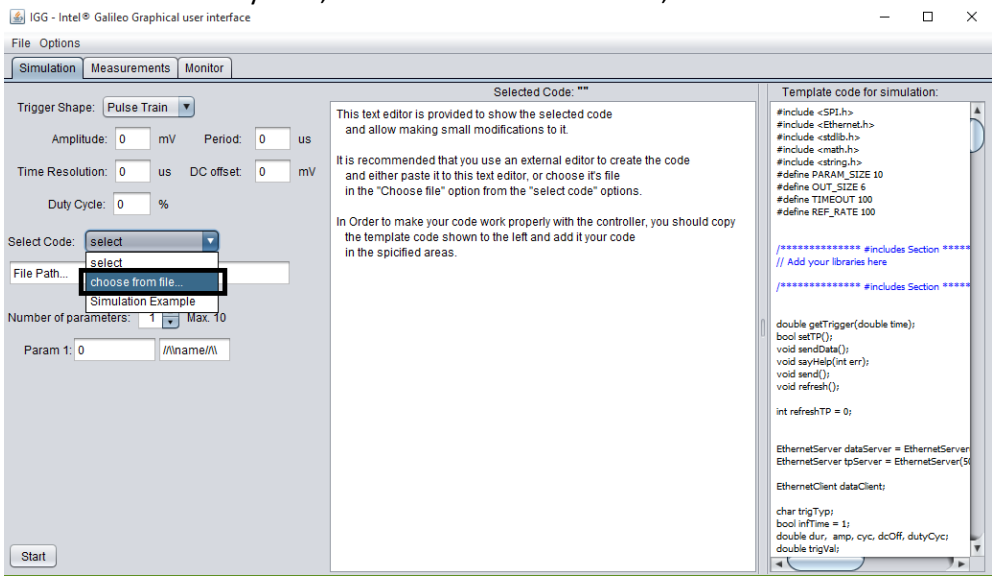

Figure 31: The Simulation Tab, to load a code from the PC, select the "Choose from file option".

3. A File Browser will open and allow you to locate your code file. After you have located your file, press "Open". The code should be displayed by now in the text area, allowing you to view it and change it if necessary.
   Note: the path of the file cannot contain names with space, as the download process won't work.

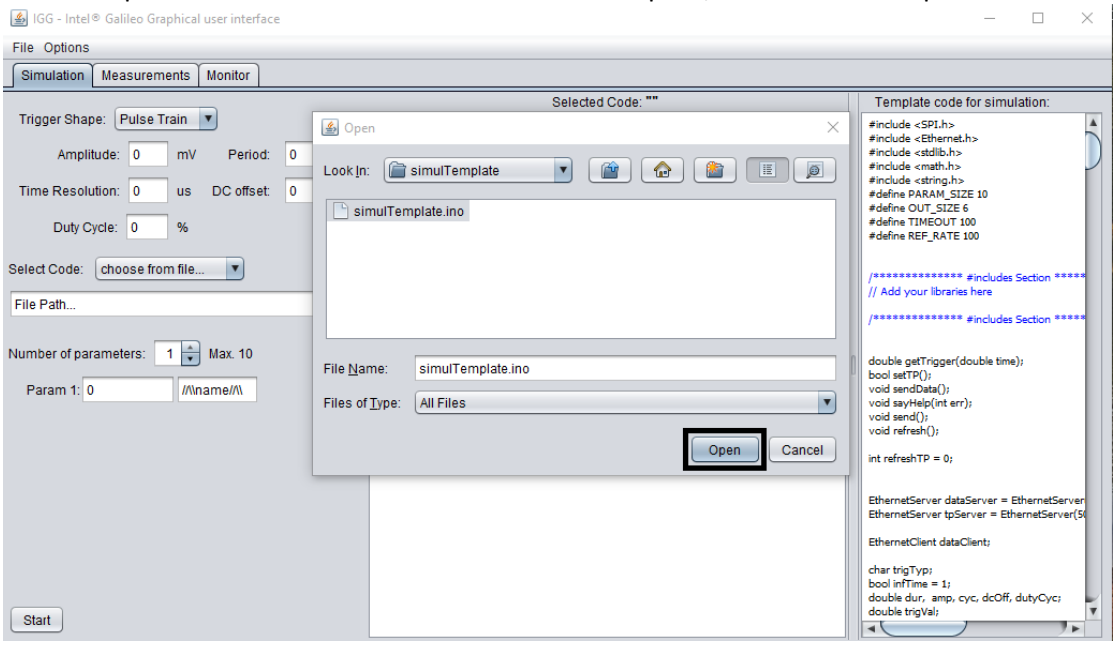

Figure 32: The File Browser. Locate your file on the PC and press "Open".
The path cannot contain spaces.

4. Select the trigger shape from the trigger shape menu (available shapes are pulse wave, sine wave, triangular wave and a ramp wave). Then, specify the trigger's fields (amplitude in mV, period in uS, time resolution in uS, DC offset in mV and in the case of a pulse wave, duty cycle in %).

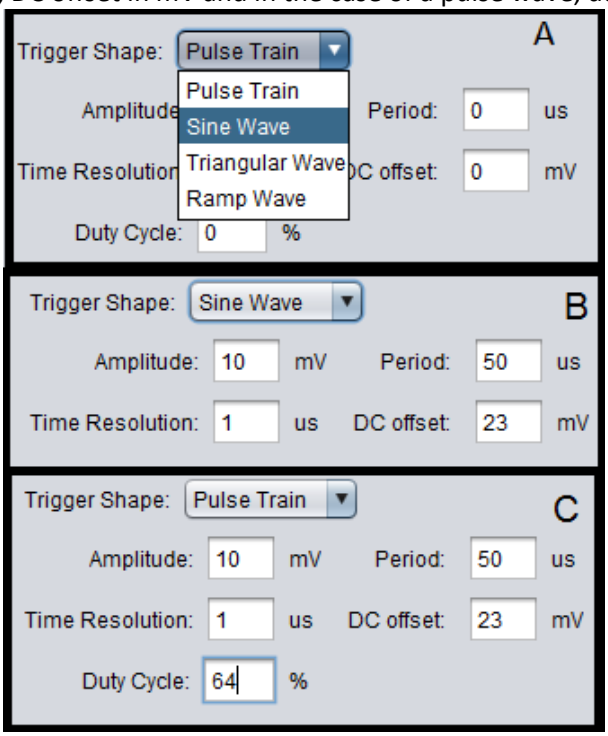

Figure 33: (A) Choosing the shape of the wave. (B) Specifying the trigger fields for a Sine wave. (C) Specifying the trigger fields for a Pulse wave

5. In addition to specifying the trigger, you can also specify up to 10 parameters to be used in the code and changed from the app while the code is running.
To increase the number of parameters press the ▲ arrow near the number of parameters displayed. To decrease it press the ▼ arrow accordingly. You can also label the parameters for your convenience (it won't have any effect on the code).

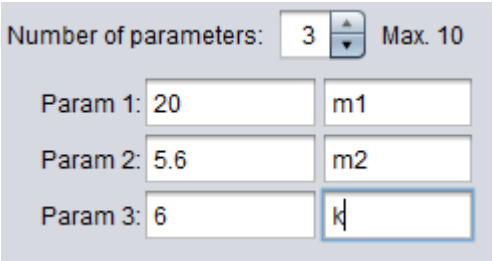

Figure 34: Specify the parameters and their values

6. After specifying all the values your code is ready to be compiled and downloaded to the Galileo. Make sure that the USB and Ethernet cords are connected to the Galileo, and press the "Start" button (in the lower left). The IGG will now compile the code, and then download it to the Galileo and will automatically start displaying the data on the plot.

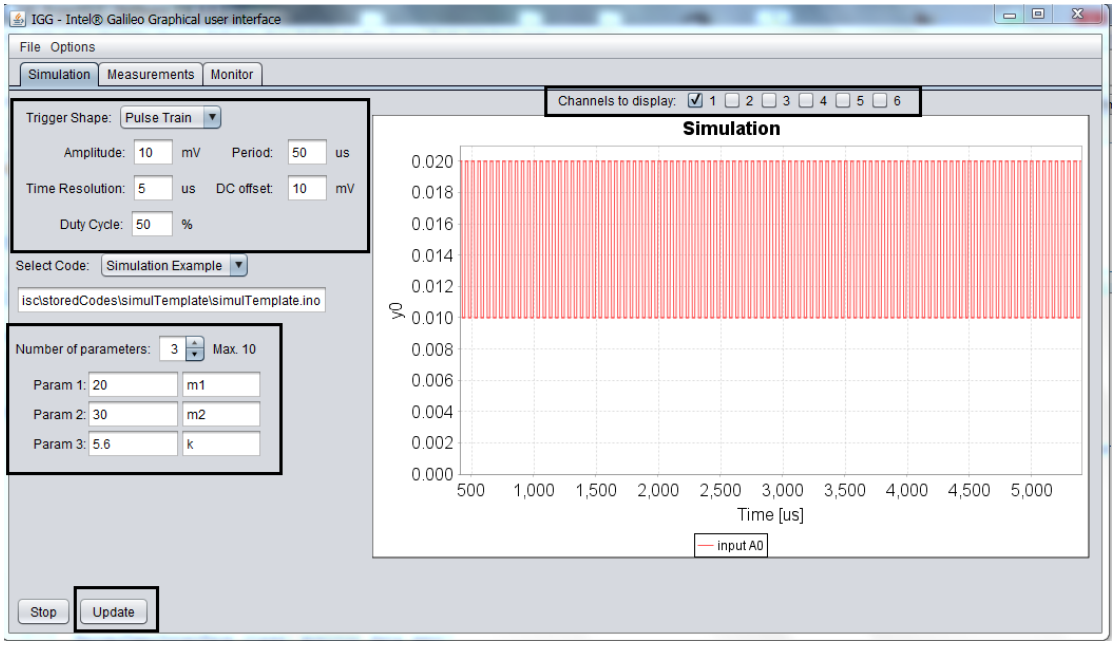

Figure 35: The Simulation is now running. You can change any field in the black rectangles.

7. While the simulation is running you can select which output you want to display, by checking the checkboxes above the plot. Once you checked/unchecked one of the outputs the plot automatically changes accordingly. You can also change the trigger shape and values as well as the parameters' values and by pressing "Update" the app will send the new data to the Galileo which will update the Simulation.
8. Pressing the "Stop" button will stop the updating of the plot and abort the simulation.

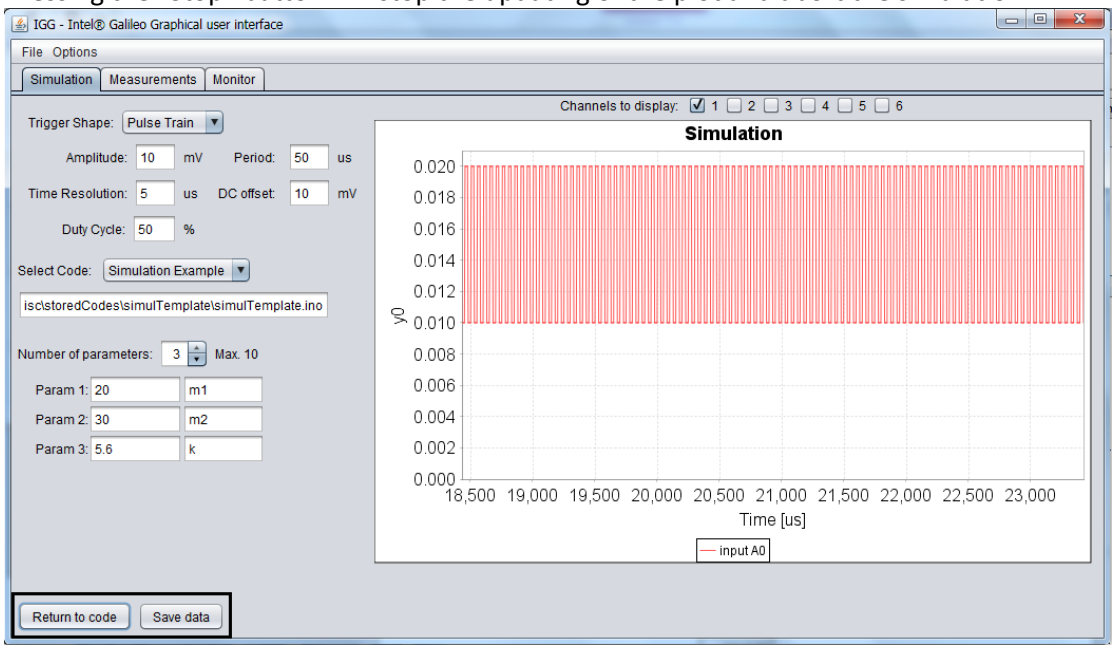

Figure 36: After the simulation has stopped you can save the data and return to the code.

9. After the simulation has stopped, you can press the "Save data" button which will open a File Browser and allow you to find a location to save the data (when you located the folder, write the file name in the "File Name:" text field and press "Save"). By pressing "Return to code" you return to the text area and now you can load a new code or change the existing one.

The Measurements Tab

Writing a model code using the given template –

In the provided template (found in the measurements tab under the “template code for measurements”) the areas for inserting the user’s code are marked with blue for #includes and variable declaration sections and red for the setup code and loop code sections. The Measurements template provides 3 variables for the user:

- A Double-type array of 10 elements: “params [0...9]” which hold the values of the parameters specified by the user and sent from the PC.
- A Double-type array of 6 elements: “output [0...5]” which hold the values that will be sent to the PC to display on the plot.
- The time of the code: “modelTime” which is updated in each loop-iteration using the total time that has passed since the code has started.

The code language is similar to C/C++ (object oriented). For the available libraries and functions see the following site: https://www.arduino.cc/en/Reference/HomePage#

How to use the Measurements tab

This section will help you load a code to the text area and then download it to the Galileo. Also it will show you how to change a parameter or trigger and update the code:

1. Run the IGG.jar. After a while the app’s window shall appear with the simulation tab as the current tab. Switch to the Measurements tab.
2. To load a code from the system, at the code-selection menu, select the “Choose from file…”

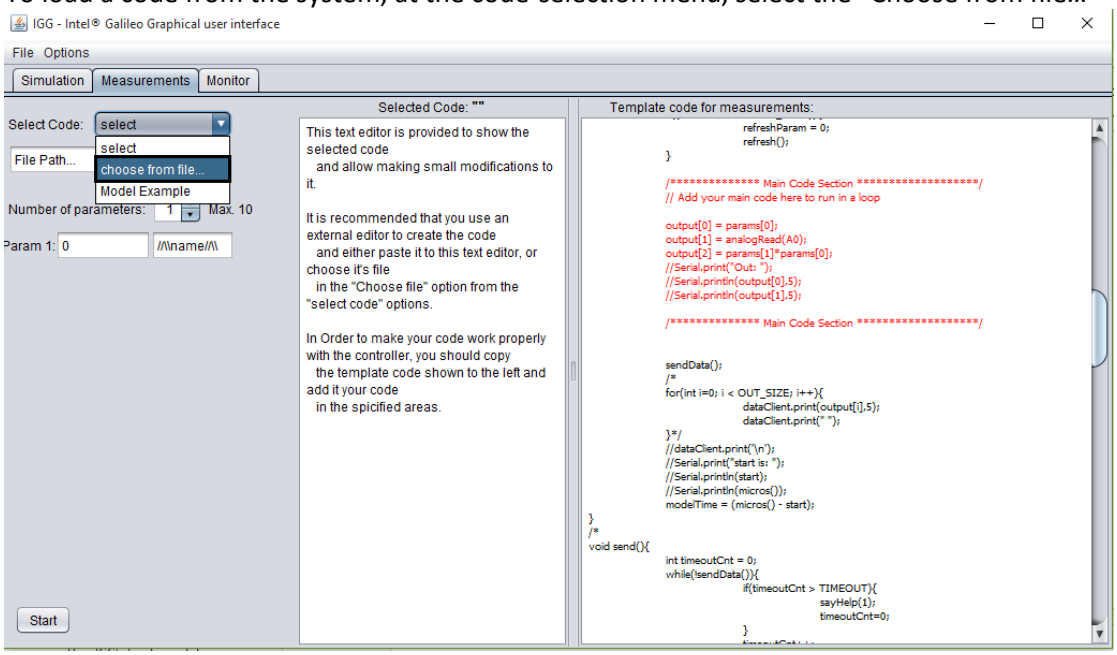

Figure 37: The Measurements Tab, to load a code from the PC, select the "Choose from file option".

3. A File Browser will open and allow you to locate your code file. After you have located your file, press “Open”. The code should be loaded by now on the text area, allowing you to view it and change it if necessary.
   Note: the path of the file cannot contain names with space, as the download process won’t work.

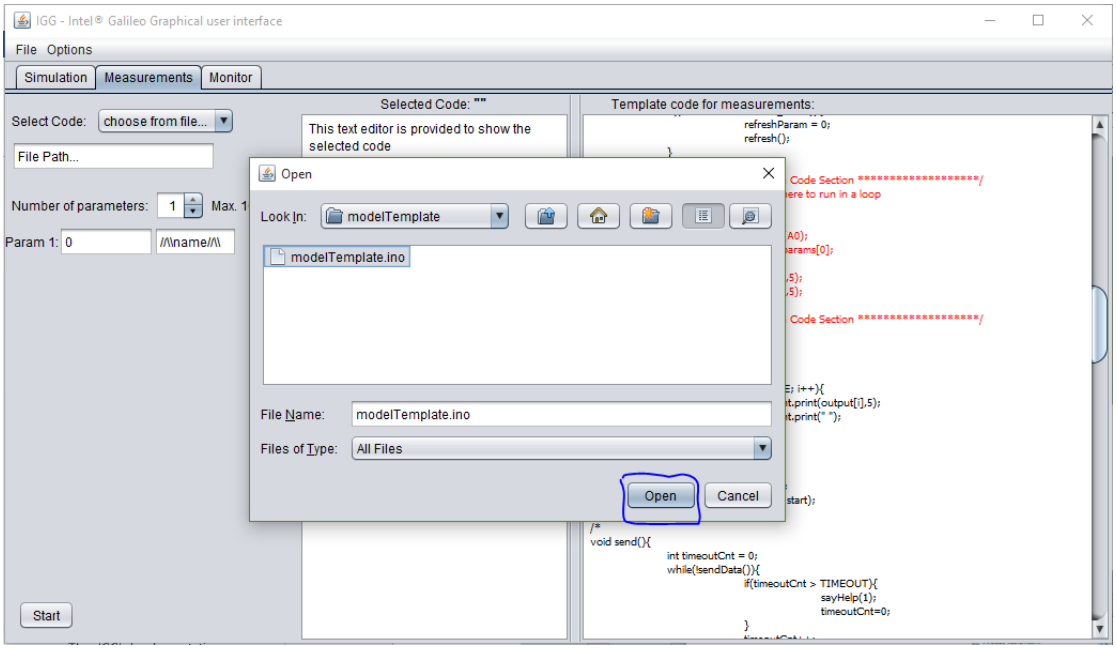

Figure 38: The File Browser. Locate your file on the PC and press "Open".
The path cannot contain spaces.

4. You can specify up to 10 parameters to be used in the code and changed from the app while the code is running.
   To increase the number of parameters press the ▲ arrow near the number of parameters displayed. To decrease it press the ▼ arrow accordingly. You can also label the parameters for your convenience (it won't have any effect on the code).

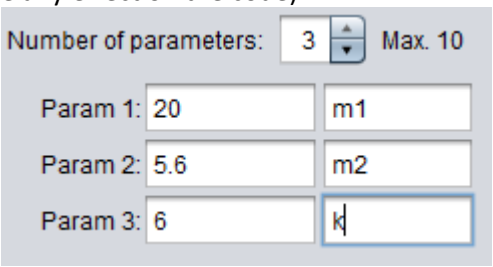

Figure 39: Specify the parameters and their values

5. After specifying all the values your code is ready to be compiled and downloaded to the Galileo. Make sure that the USB and Ethernet cords are connected to the Galileo, and press the "Start" button. The IGG will now compile the code, and then download it to the Galileo and will automatically start displaying the data on the plot.

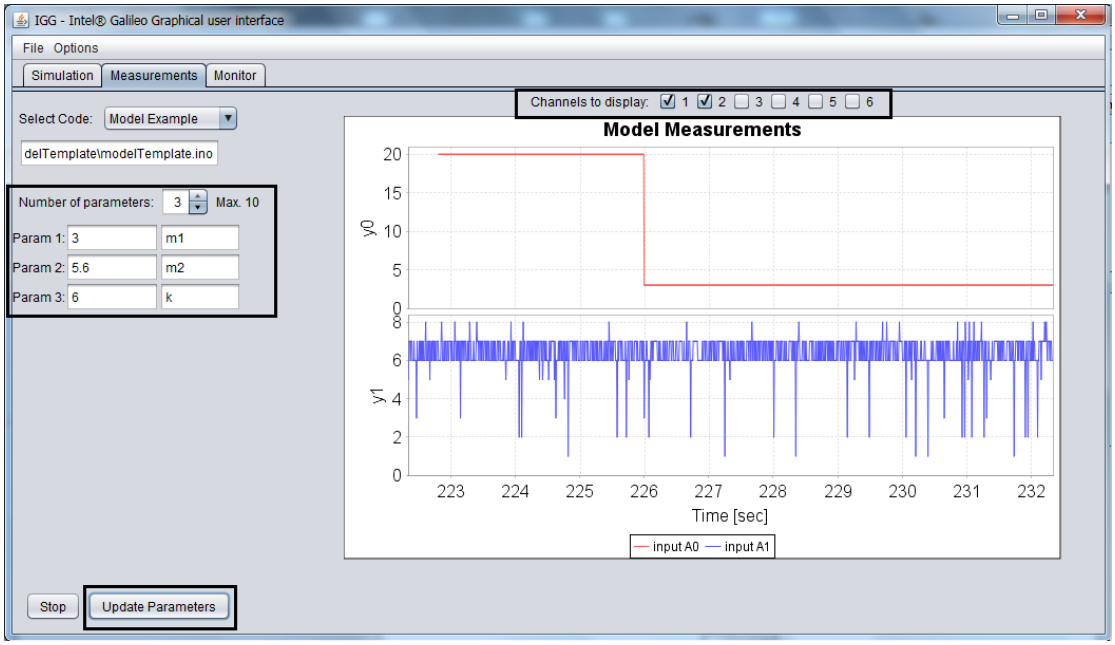

Figure 40: The Simulation is now running. You can change any field in the black rectangles.

6. While the code is running you can select which output you want to display, by checking the checkboxes above the plot. Once you checked/unchecked one of the outputs, the plot automatically changes accordingly. You can also change the parameters' values and by pressing "Update" button the app will send the new data to the Galileo which will update the Session.
   In *Figure 40* the code used sets the first output as the first parameters, so at the beginning it had the value of 20, and then it was updated to the value of 3.
7. Pressing the "Stop" button will stop the updating of the plot and abort the session.

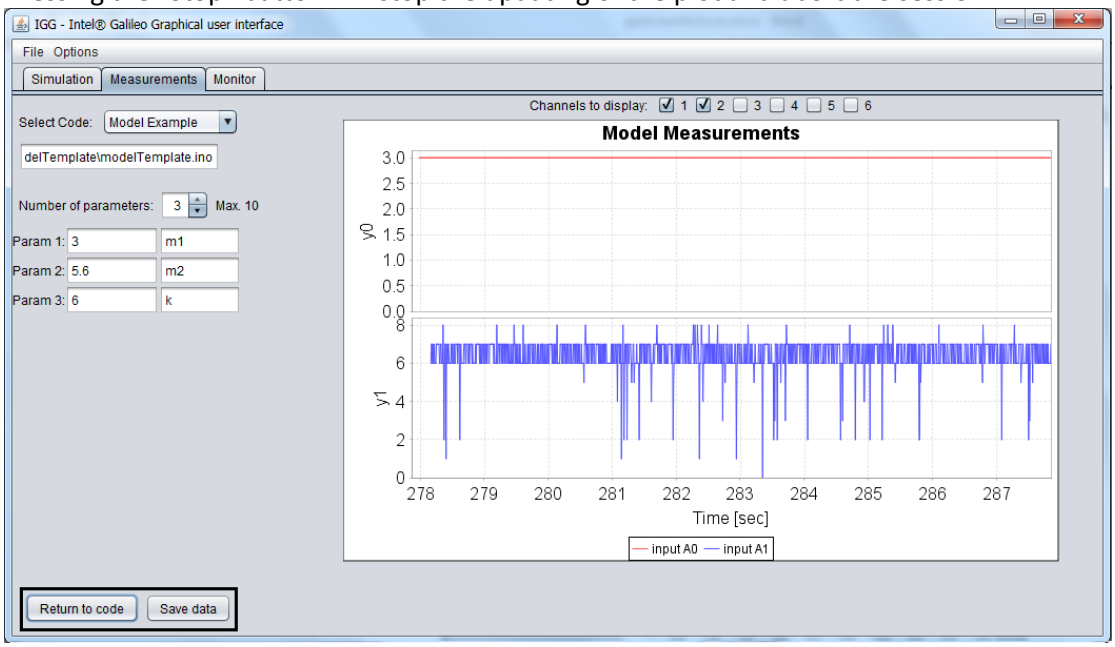

Figure 41: After the simulation has stopped you can save the data and return to the code.

8. After the session has stopped, you can press the "Save data" button which will open a File Browser and allow you to find a location to save the data (when you have located the folder, write the file name in the "File Name:" text field and press "Save"). By pressing "Return to code" you return to the text area and now you can load a new code or change the existing one.

The Monitor Tab

This section will show you how to use the monitor tab which displays the analog inputs of the Galileo.

1. Run the IGG.jar. After a while the app's window shall appear with the simulation tab as the current tab. Switch to the Monitor tab.
2. When you switch to the monitor tab for the first time, the app will automatically start the monitor session, and will try to connect to the Galileo through the network, so you need to make sure that the Ethernet cord is connected before you run the IGG. The connection will take some time to start, and a message will appear until the session starts.

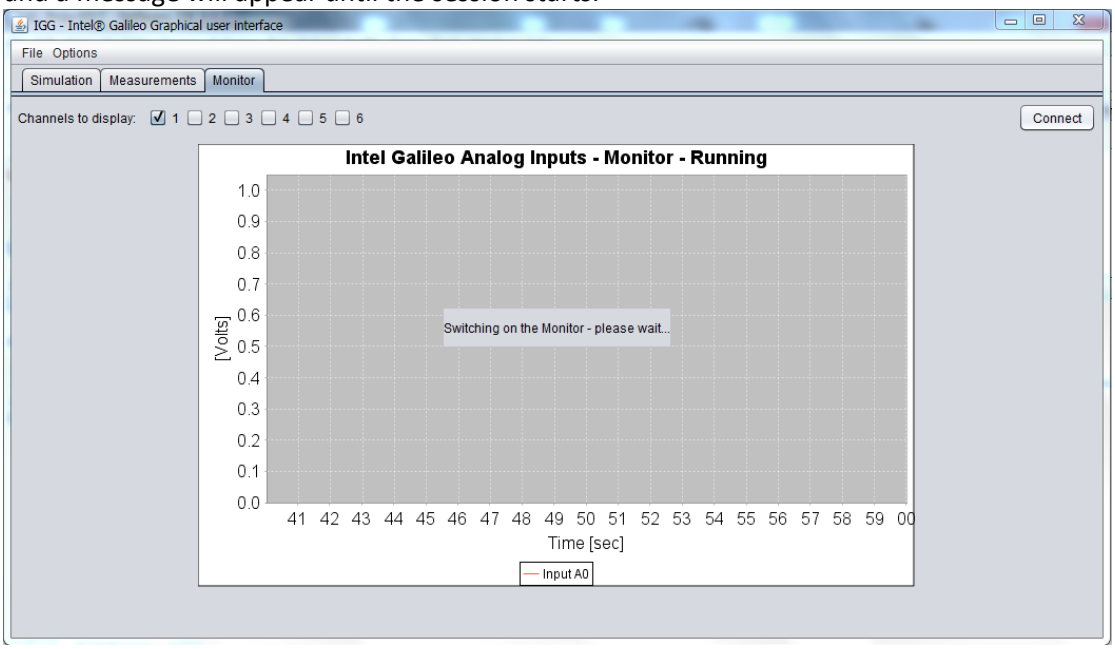

Figure 42: The Monitor Session is starting up.

3. If the Galileo is running and connected to the network the plot will automatically start receiving the data from the analog inputs of the Galileo.
4. Once the session is running you can choose which channels (inputs) you want the app to display and which not, by checking and unchecking the checkboxes in the left corner above the plot. Once a channel was checked/unchecked, the plot automatically updates itself.

Figure 43: The Monitor session running with 3 analog inputs displayed - A0, A2, A5

5. To stop the session, press the stop button on the right corner above the plot. This will stop the update process of the plot.
6. After the session has stopped, you can press the "Save" button which will open a File Browser and allow you to find a location to save the data (when you located the folder, write the file name in the "File Name:" text field and press "Save").

Figure 44: The Monitor Session has stopped.
You can now save the data from the session or start a new one.

7. To restart the Monitor session, simply press the "Connect" button and wait for the session to start.

Adding a Code to the Stored Codes List of the IGG

This section will guide through the process of storing a code in the IGG, so next time it will run, you will be able to easily load the code, instead of searching it on the PC.

1. Run the IGG.jar.
2. Once the window has opened, open the “Options” menu and select the “Add new code to stored codes…” option.

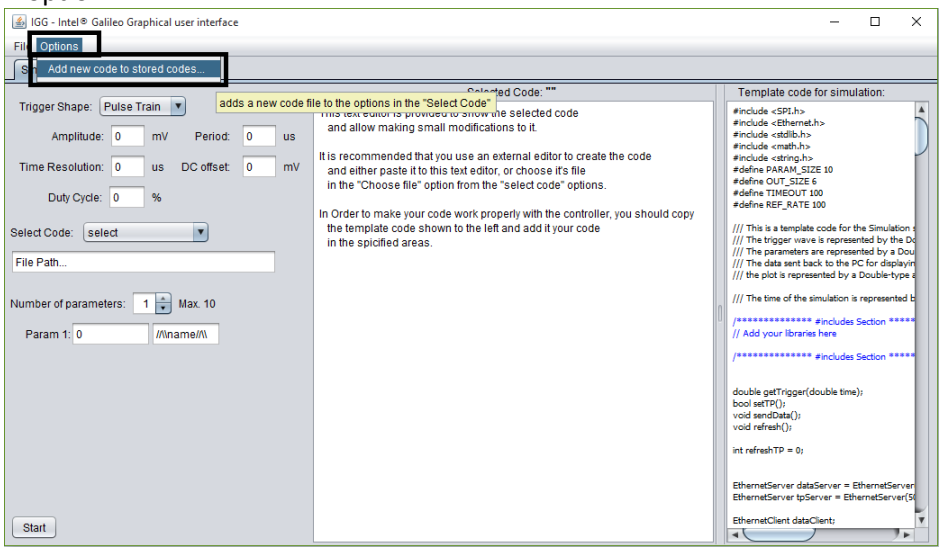

Figure 45: Opening the "Options" window and selecting the "Add new code to stored codes..." option

3. A window will open asking you if you want to store the in only one of the sessions (Simulation or Measurements) or both. For this demonstration choose “both”.

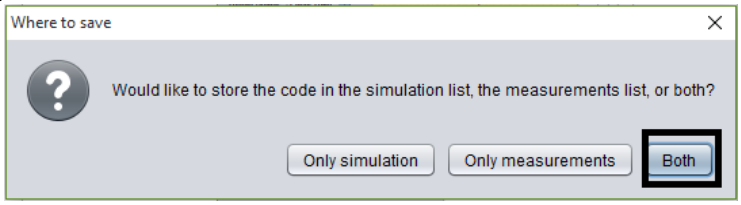

Figure 46: A popup window asking where to store the code.

4. A File Browser Window is opened. Locate the code you wish to store and press the “Open” button.

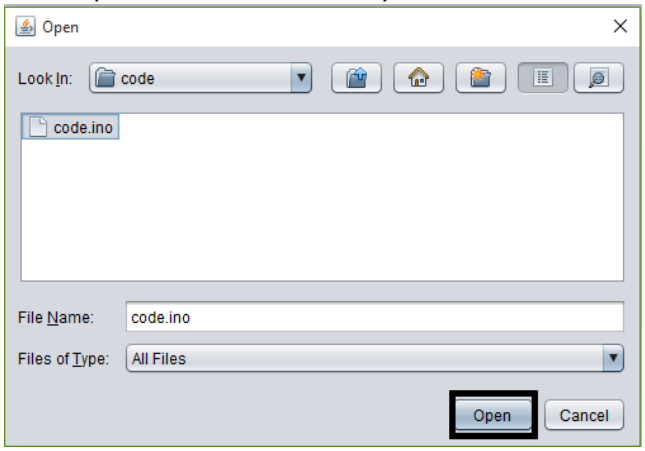

Figure 47: The File Browser window. Locate your code and press "Open"

5. After the file browser is closed another window pops up asking for the code’ label. Give the code your label and press the “OK” button.

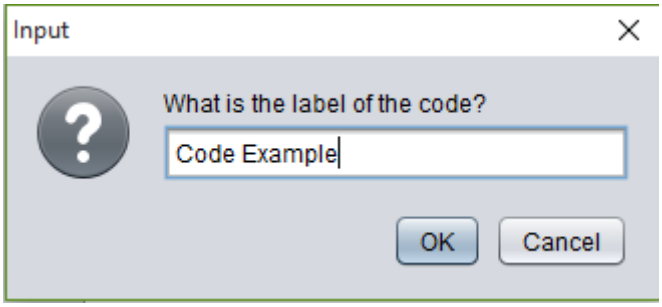

Figure 48: Give your code a label.

6. Done. Now the code is stored and you can load it by choosing its label in the code-selection menu in the session it was stored in (in this case in both sessions – Simulation and Measurements).

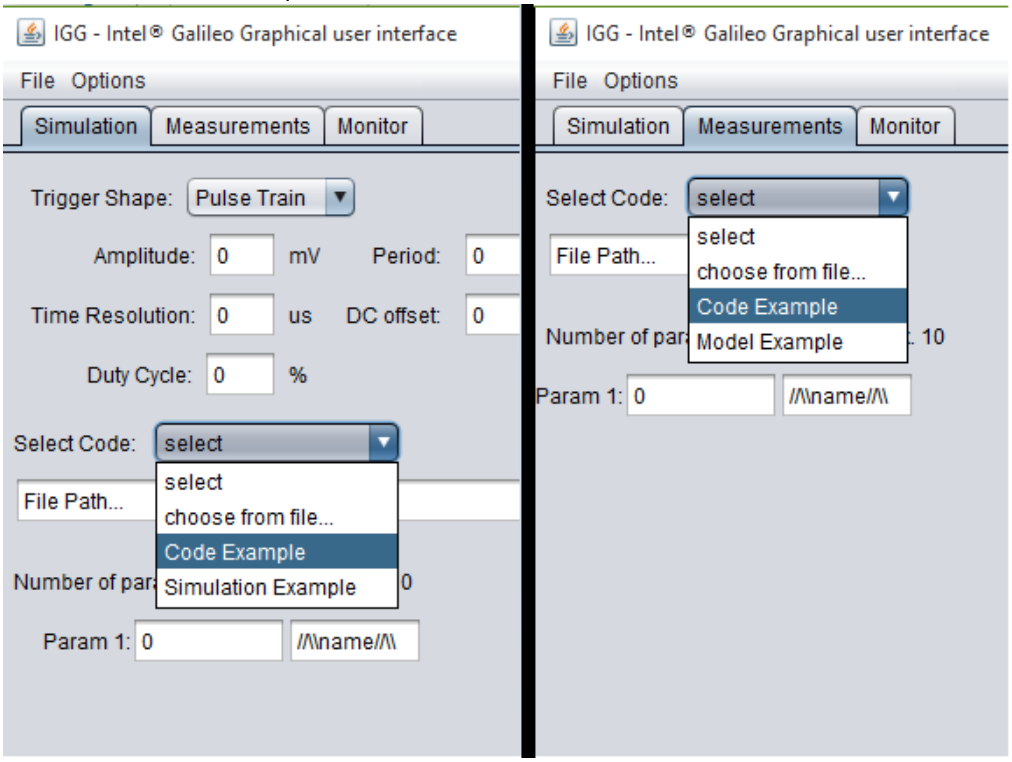

Figure 49: After storing the code, it can be found the code-selection menu

Note: all the stored codes are saved in one folder, so if a code is stored in both sessions, modifying it in one session will also modify it in the other session.

Note – A Problem Which May Occur in the IGG

The JFreeChart library used for displaying data on a plot doesn't support concurrency, which is partially used in the app's code. This may cause the plot to freeze and stop updating itself while the code is running. The app itself doesn't freeze and still functions. This happens very rarely and the solution for it is simple: if the problem occurs in the Simulation or Measurements tabs, press the "stop" button and then "save" to save your data from the session until the moment the plot stopped. Then press "return to code" and you can restart the session and the plot will work. If the problem occurs in the Monitor tab, press the "stop" button, then "save" to save the data from the session, and then restart the app.

The fact that the JFreeChart library doesn't support concurrency was discovered in a late phase in the building of the IGG. Considering the fact that the problem which rises happens very rarely, it was decided to keep using the library, as it is better than the other options which are available.

Chapter 4: Summery

In this project the Intel® Galileo Board controller was examined and characterized, and graphical user interface was built to allow a simple use of the Galileo without a profound knowledge in programming. The main work with the Galileo and the building of the GUI was based on open-source code and information found on the internet.

The first intent was to try and integrate the Galileo as a measuring equipment in the laboratory. However, after examining its performance, it was understood that the measuring capabilities are somewhat poor relatively to the existing measuring tools in the lab. Nevertheless, it was decided to build the GUI and use the Galileo for simple tasks, such as simulations and simple measurements of large time scale systems (of the order of milliseconds).

Suggestions for further use

- The GUI can be expanded to support the use of 2 or more controllers together, to simulate a group of elements and their interactions with each other.
- The current GUI uses the USB serial interface for downloading the code to the controller (based on a script used by the Arduino IDE). It can be changed and configure the GUI the download the code through the Ethernet and then will require only the Ethernet to be connected (or the board to be connected to the network).
- This work is based on the Intel® Galileo Board. However the Intel© company has also designed the Intel® Galileo Gen2 which is the successor of the Galileo board. The performance of the Intel® Galileo Gen2 should be examined and analyzed to see whether it's better than its predecessor. Also, other micro-controller boards, with different specifications and abilities should be considered for integration with the lab's work.
- The controller, is connected to the PC by an Ethernet cord. A WiFi card can attached to the board and allow it to connect to the Local-Area-Network wirelessly, so any computer can access it (this will require only modifying the setup of the board, and the templates used by the IGG).

References

- A background on the Arduino controllers:
  https://en.wikipedia.org/wiki/Arduino
- The Arduino site, which provides support for the developing platform of the Arduino controllers and everything concerning them (including the Galileo):
  www.arduino.cc
- A “Getting Start” guide for the Intel® Gelileo (disregard the updating firmware part on this page):
  https://learn.sparkfun.com/tutorials/galileo-getting-started-guide
- A thread in the Intel support forum regarding the I/O speeds of the Galileo:
  https://communities.intel.com/message/207904#207904
  (title: “I/O speeds?”)
- A guide about the Java Language Platform:
  http://www.oracle.com/technetwork/topics/newtojava/downloads/index.html
- A page for downloading the Intel© firmware updater tool, with a guide:
  https://software.intel.com/en-us/installing-drivers-and-updating-firmware-for-arduino-windows
- A help page about running the Arduino IDE from the command line:
  https://github.com/arduino/Arduino/blob/master/build/shared/manpage.adoc
- The Stack OverFlow forum where many questions about the GUI building were answered:
  http://stackoverflow.com/

Appendices

Appendix A – Making a code running on the Galileo on startup

This section requires a terminal to the Galileo's Linux OS image, and the use of a file transfer program like WinSCP.

In order to add a script or process to the Galileo system so that it would run on startup, follow the next instructions:

1. Write a Bash script of the following form:

```
#!/bin/sh
###BEGIN INIT INFO
#Provides: PROGNAME                                  # PROGNAME is the name of the program/script you want to run
# Should-Start: console-screen dbus network-manager
# Required-Start: $all                              # specifies the set of facilities that should be started before
                                                    # starting this service. In our case we want to start it last so we put
"$all"
# Required-Stop: $all                               # same deal - specifing the fac. that should be stopped AFTER
                                                    # stopping this service. a list downunder explains what every option
means
# Default-Start: 2 3 4 5                            #  defines the run-levels in which the service has to be started
# Default-Stop: 0 1 6                               #  defines the run-levels in which the service has to be stopped
# Short-Description: start PROGNAME at boot time
###END INIT INFO
#

PATH="/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin" # This specifies the DIRs where the necessary func. are
found.

###
# the startup script has a input argument for the system to start it or stop it - "$1"
#this case specifies what the script will do according to the input it has (start/stop)

case "$1" in
start)
            #### write your code here to start a process or a script or just some code
            #### if you have a program/script you want to run then
            #### under PATH line define <SCRIPT="script_path">  and here write <$SCRIPT>
            #### (without the <> of course)
    ;;
stop)
            #### write your code here to end a process or a script or just some code
            #### if you have a program running you want to stop,
            #### define under PATH line <PROGNAME="program_name">
            #### and then write here <skill $PROGNAME>
    ;;
esac

exit 0
```

2. Move the script to the Galileo to "/etc/init.d/" folder.
3. Make the script executable using the following command in the terminal:
   chmod +x /etc/init.d/SCRIPT.sh where **SCRIPT.sh** is the script you worte.
4. Symbolically link the script to the "/etc/rc#.d/" folders (# is 1…6) with the following commands:
   ln –s /etc/init.d/SCRIPT.sh /etc/rc#.d/S##SCRIPT.sh
   ln –s /etc/init.d/SCRIPT.sh /etc/rc#.d/K##SCRIPT.sh where ## is 00..99. the ## number will tell the system in which order to **S**tart the script and in which order to **K**ill the script. 99 means last started or last killed. For example, with a script named "netcon.sh":
   ln -s /etc/init.d/netcon.sh /etc/rc2.d/S99netcon.sh; *
   ln -s /etc/init.d/netcon.sh /etc/rc3.d/S99netcon.sh;
   ln -s /etc/init.d/netcon.sh /etc/rc4.d/S99netcon.sh;
   ln -s /etc/init.d/netcon.sh /etc/rc5.d/S99netcon.sh;
   ln -s /etc/init.d/netcon.sh /etc/rc6.d/S99netcon.sh;
   ln -s /etc/init.d/netcon.sh /etc/rc1.d/K99netcon.sh; **
   * This command symbolically links the script to rc2.d folder and tells the system to start it after all services have started.

**This command symbolically links the script to rc1.d folder and tells the system to end it after all services have ended.

5. Done! Now your script will run every time the Galileo stars up.

For more information see: “How to Write Linux Init Scripts Based on LSB Init Standard”
http://www.thegeekstuff.com/2012/03/lsbinit-script/

Appendix B – Downloading a file to the Galileo

The download process is based on a script called “clupload_win.sh” which can be found in the \hardware\intel\i586-uclibc\tools\izmir\ folder within the Arduino IDE directory. This script downloads a compiled sketch to the Galileo, moves it to the /sketch/ folder, renames the current sketch to “sketch.old.elf” and the new sketch to “sketch.elf” and finally makes the sketch executable. The script is shown in *Figure 50*. You can modify it to download files to the Galileo and perform additional commands. A modified script was used to make the network script run on startup (in Appendix A), using the sIGG app. Note that for some reason, when the script is modified and additional commands are added, the script won’t work unless the following line is executed before every command to the Galileo: “`echo "~sketch downloadGalileo" > $tty_port_id:`”.

```
#!/bin/sh

# clupload script to invoke lsz
# Copyright (C) 2014 Intel Corporation
#
# This library is free software; you can redistribute it and/or
# modify it under the terms of the GNU Lesser General Public
# License as published by the Free Software Foundation; either
# version 2.1 of the License, or (at your option) any later version.
#
# This library is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
# Lesser General Public License for more details.
#
# You should have received a copy of the GNU Lesser General Public
# License along with this library; if not, write to the Free Software
# Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA  02110-1301  USA
#

echo "starting download script"
echo "Args to shell:" $*

# ARG 1: Path to lsz executable.
# ARG 2: Elf File to download
# ARG 3: COM port to use.

#path contains \ need to change all to /
path_to_exe=$1
fixed_path=${path_to_exe//\\/\/}

#COM ports are not always setup to be addressed via COM for redirect.
#/dev/ttySx are present. Howwever, COMy -> /dev/ttySx where x = y - 1

com_port_arg=$3
com_port_id=${com_port_arg/COM/}
echo "COM PORT" $com_port_id
tty_port_id=/dev/ttyS$((com_port_id-1))
echo "Converted COM Port" $com_port_arg "to tty port" $tty_port_id

echo "Sending Command String to move to download if not already in download mode"
echo "~sketch downloadGalileo" > $tty_port_id

#Move the existing sketch on target.
echo "Deleting existing sketch on target"
"$fixed_path/lsz.exe" --escape -c "mv -f /sketch/sketch.elf /sketch/sketch.elf.old" <> $tty_port_id 1>&0
# Execute the target download command

#Download the file.
host_file_name=$2
"$fixed_path/lsz.exe" --escape --binary --overwrite $host_file_name <> $tty_port_id 1>&0

#mv the downloaded file to /sketch/sketch.elf
target_download_name="${host_file_name##*/}"
echo "Moving downloaded file to /sketch/sketch.elf on target"
"$fixed_path/lsz.exe" --escape -c "mv $target_download_name /sketch/sketch.elf; chmod +x /sketch/sketch.elf"
<> $tty_port_id 1>&0
```

Figure 50: The clupload_win.sh script. Used for downloading the sketch to the Galileo and starting it.

Appendix C – Sketches for examining the performance of the Galileo

The following 4 sketches in *Figure 51* are used to measure the possible output frequencies of the Galileo on digital pin 2. These are ".ino" sketches which were used in the Arduino IDE.

```
/**
* This code uses the simple digitalWrite() function on pin 2 on the Intel(R) Galileo,
* which produces an output frequency of ~230 Hz (which is rather slow)
*/
{
    void setup() {
      pinMode(2,OUTPUT);

    }

    void loop() {
      register int x=0;
      while(1){
        digitalWrite(2,x);
        x = !x;
      }
    }
}

/**
* This code adjusts the frequency of the output of pin 2
* on the Intel(R) Galileo to ~450 KHz,
* using the OUTPUT FAST mode of pin 2 and the digitalWrite() function
*/
{
    void setup() {
      pinMode(2,OUTPUT_FAST);
    }

    void loop() {
      register int x=0;
      while(1){
        digitalWrite(2,x);
        x = !x;
      }
    }
}

/**
* This code adjusts the frequency of the output of pin 2
* on the Intel(R) Galileo to ~670 KHz,
* using the OUTPUT_FAST mode of pin 2 and the fastGpioDigitalWrite() function
*/
{
    void setup() {
      pinMode(2,OUTPUT_FAST);
    }

    void loop() {
      register int x = 0;
      while(1){
        fastGpioDigitalWrite(GPIO_FAST_IO2,x);
        x = !x;
      }
    }
}

/**
* This code adjusts the frequency of the output of pin 2
* on the Intel(R) Galileo to ~2.9 MHz (the fastest available),
* using the OUTPUT FAST mode of pin 2 and the fastGpioDigitalWriteDestructive() function
* the code latches the value on the register connected to the pin and writes driectly to it.
*/
{
    uint32_t latchValue;

    void setup() {
      pinMode(2, OUTPUT_FAST);
      latchValue = fastGpioDigitalLatch();
    }

    void loop() {
      while(1){
        fastGpioDigitalWriteDestructive(latchValue);
        latchValue ^= GPIO_FAST_IO2;
      }
    }
}
```

Figure 51: Four sketches for measuring the possible output frequency of digital pin 2

The following code in *Figure 52* is the code used for measuring the maximum sampling rate of the Galileo. The code measured the time taken to sample the analog inputs, and from that data the maximum frequency was derived.

```
  int sensor_value1;
  int sensor_value2;
  int sensor_value3;
  int sensor_value4;
  int sensor_value5;
  int sensor_value6;
  unsigned long time1;
  unsigned long time2;


void setup(){
  Serial.begin(9600);
}

void loop(){

  analogReadResolution(12);

  // Time taken to read 1 channel
  time1 = micros();
  sensor_value1 = analogRead(A0);
  Serial.print(micros()-time1);
  Serial.print("\t");

  // Time taken to read 2 different
channels
  time1 = micros();
  sensor_value1 = analogRead(A0);
  sensor value2 = analogRead(A1);
  Serial.print(micros()-time1);
  Serial.print("\t");

  // Time taken to read 3 different
channels
  time1 = micros();
  sensor value1 = analogRead(A0);
  sensor_value2 = analogRead(A1);
  sensor_value3 = analogRead(A2);
  Serial.print(micros()-time1);
  Serial.print("\t");


  // Time taken to read 4 different
channels
  time1 = micros();
  sensor_value1 = analogRead(A0);
  sensor_value2 = analogRead(A1);
  sensor_value3 = analogRead(A2);
  sensor_value4 = analogRead(A3);
  Serial.print(micros()-time1);
  Serial.print("\t");

  // Time taken to read 5 different
channels
  time1 = micros();
  sensor_value1 = analogRead(A0);
  sensor value2 = analogRead(A1);
  sensor value3 = analogRead(A2);
  sensor_value4 = analogRead(A3);
  sensor_value5 = analogRead(A4);
  Serial.print(micros()-time1);
  Serial.print("\t");

  // Time taken to read 6 different
channels
  time1 = micros();
  sensor_value1 = analogRead(A0);
  sensor_value2 = analogRead(A1);
  sensor_value3 = analogRead(A2);
  sensor_value4 = analogRead(A3);
  sensor_value5 = analogRead(A4);
  sensor value6 = analogRead(A5);
  Serial.print(micros()-time1);
  Serial.print("\t");

//////////////////////////////////////
//////
  // Time taken to read 6 times the
same channel
  time1 = micros();
  sensor_value1 = analogRead(A0);
  sensor value2 = analogRead(A0);
  sensor value3 = analogRead(A0);
  sensor_value4 = analogRead(A0);
  sensor_value5 = analogRead(A0);
  sensor_value6 = analogRead(A0);
  Serial.print(micros()-time1);
  Serial.print("\t");

  // Time taken to read 5 times the
same channel
  time1 = micros();
  sensor_value1 = analogRead(A0);
  sensor_value2 = analogRead(A0);
  sensor_value3 = analogRead(A0);
  sensor_value4 = analogRead(A0);
  sensor value5 = analogRead(A0);
  Serial.print(micros()-time1);
  Serial.print("\t");

  // Time taken to read 4 times the
same channel
  time1 = micros();
  sensor value1 = analogRead(A0);
  sensor_value2 = analogRead(A0);
  sensor_value3 = analogRead(A0);
  sensor_value4 = analogRead(A0);
  Serial.print(micros()-time1);
  Serial.print("\t");

  // Time taken to read 3 times the
same channel
  time1 = micros();
  sensor_value1 = analogRead(A0);
  sensor_value2 = analogRead(A0);
  sensor_value3 = analogRead(A0);
  Serial.print(micros()-time1);
  Serial.print("\t");

  // Time taken to read 2 times the
same channel
  time1 = micros();
  sensor_value1 = analogRead(A0);
  sensor_value2 = analogRead(A0);
  Serial.print(micros()-time1);
  Serial.print("\t");

  // Time taken to read a channel
  time1 = micros();
  sensor_value1 = analogRead(A0);
  Serial.print(micros()-time1);
  Serial.print("\t");

  // Time taken for the code around the
sampling lines.
  time1 = micros();
  Serial.print(micros()-time1);
  Serial.print("\t");
  Serial.print("\n");
}
```

Figure 52: The code used to measure the sampling rate of the Galileo.

The code measures the time taken for a sampling of one channel, 2,3,4,5 and 6 sequential samplings for different channels, and 2,3,4,5 and 6 sequential samplings from the same channel. The data printed to the serial terminal is then collected and analyzed to calculate the maximal sampling rate. The MatLab function used to calculate the sampling rate is shown in *Figure 53*.

```
function [freqs, stdVal] = analogReadTimeValsAnalysis(fileName)
    %% The fileName is a tab seperated ".txt" file with 13 columns
    %% The function returns:
    %%% freqs - the frequencies for each sampling
    %%% stdVal - the standard deviation for each frequency
    data = importdata(fileName);

    all = data.data;
    channels = (all(:,1:(end-1)));

    % avrage time taken for the code around the sampling to run
    timeOfMeas = mean(all(:,end)).*ones(size(channels));

    % average time without the code around runtime
    channels = channels - timeOfMeas;

    % each column has its own number of channels
    channelNumber = repmat([1:6,6:-1:1],size(channels,1),1);

    % Time taken for the sampling of one channel in each column
    timePerChannel = channels ./ channelNumber;

    periods = timePerChannel*2*1e-6;  % the periods in microseconds
    freqs = 1./periods; % the frequencies
    stdVal = std(freqs); % the standard deviation of the frequencies
    freqs = mean(freqs); % the average of the frequencies
    freqs = [freqs(1:6);freqs(end:-1:7)];
    stdVal = [stdVal(1:6);stdVal(end:-1:7)];
end
```

Figure 53: The MatLab code used to calculate the sampling rate of the Galileo

Appendix D – Templates and Code for all the Sessions (Simulation, Measurements and Monitor)

The code presented in this appendix is an extra-commented code, and not shown in the templates in the app itself. It can be found in the “src\Misc\TemplatesWithComments” folder.

Simulation Template

The following code is the template code used for the Simulation session:

```
/// Including necessary libraries
#include <SPI.h>
#include <Ethernet.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
////////////////////////////////////////
/// Defining Constants such as size of output paramteres array, size of parameters array that
/// can be changed from the IGG, Refreshing time after which the Galileo accepts any updates.
#define PARAM_SIZE 10
#define OUT_SIZE 6
#define TIMEOUT 100
#define REF_RATE 100
////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////
/// This is a template code for the Simulation session.
/// The trigger wave is represented by the Double-type variable "trigVal"
/// The parameters are represented by a Double-type array of 10 elements  - "params[]"
/// The data sent back to the PC for displaying it on
/// the plot is represented by a Double-type array of 6 elements - "output[]"
/// The time of the simulation is represented by the Double-type variable "simuTime"
////////////////////////////////////////////////////////////////////////////////////////////

/************** #includes Section ******************/
// add your libraries here as well as #define constants and function declarations
// Any additional functions you add have be declared before their first call in the code
/************** #includes Section ******************/
////////////////////////////////////////////////////////////////////////////////////////////
/// Declarations of functions used in the code
double getTrigger(double time);
bool setTP();
void sendData();
void refresh();
////////////////////////////////////////////////////////////////////////////////////////////
/// Variables used in the code
////////////////////////////////////////////////////
/// Variables responsible for server connection interface between the PC and the Galileo
EthernetServer dataServer = EthernetServer(50001);  /// server for sending data to the PC
EthernetServer tpServer = EthernetServer(50011);    /// server for reading data from the PC
EthernetClient dataClient;                          /// client for writing the data to the dataServer.
////////////////////////////////////////////////////
/// Varibales representing the trigger:
/// duration(dur - unused), amplitude (amp), period (cyc),
/// DC offset (dcOff) and the duty cycle (dutyCyc), Time Resolution (timeRes),
/// the type of the trigger (trigTyp), and finally the value of trigger at a given time.
char trigTyp;
double dur,  amp, cyc, dcOff, dutyCyc;
double trigVal, timeRes;
//////////////////////////////////////////////////////////////////////
/// Additional Variables
//bool infTime = 1; // for future uses of adding the option of limited session time
double params[PARAM_SIZE] = {0};  ///  parameters array which the user can change its values during the runtime of the code.
double output[OUT_SIZE] = {0};        ///  output array which the Galileo sends to the PC to display on the graphes.
double simuTime;                      ///  used for storing the time of the session. and the duration of the simulation.
double start;                         ///  stores the time at which the session started for future reference. (unused)
int refreshTP = 0;                    ///  counting variable used to count till the REF_RATE contant

/************** Variable Declaration Section ******************/
// Declare your variables here


/************** Variable Declaration Section ******************/
////////////////////////////////////////////////////
/*
This function runs only once at the begining of the code, and is used for setting up the code.
In this template, it performs the following tasks:
  - sets the resolution of the analog readings to 12 bits.
  - start the Serial interface for debugging.
  - starts the servers which exchange data with the PC.
  - sets the "simuTime" var to 0.
  - updates the paramteres values sent from the PC (using the startUp function).
  - performs the user setup code.
*/
void setup() {
  analogReadResolution(12);
  Serial.begin(9600);
  tpServer.begin();
  dataServer.begin();

  simuTime = 0;

  startUp();

  /************** Setup Code Section ******************/
  // Add your setup code to run once
  // Your paramters are stored in an array called params.
  /************** Setup Code Section ******************/
}
```

```
/*
This function runs repetitvely and used for the main code tasks.
In this template, it performs the following tasks in an infinite loop:
  - add to the "refreshTP" counter and if the count reached the "REF_RATE"
    constant's value then call the "refresh" function and reset the counter.
  - perform the user code
  - send the output array to the PC using the "sendData" function.
  - update the time of the session in the "simuTime" var.
*/
void loop() {
  refreshTP++;

  //if((!infTime)&&(simuTime > dur)){exit(0);} // for future uses
  if((refreshTP > REF_RATE)){
    refreshTP = 0;
    refresh();
  }

  // start = micros(); //unused
  trigVal = 0.001*getTrigger(simuTime); /// setting the trigger value in the given time

  /************** Main Code Section ******************/
  // Add your main code here to run in a loop
  // To display data on the plot in the IGG, put it in the output[] array, and
  // at the end of each loop-iteration the data will be sent
  // to the PC using the send() function
  /************** Main Code Section ******************/

  sendData();
  simuTime += timeRes;
}


/*
This function starts up the server connection, and sets the parameters values and the trigger fields.
It waits for the PC to start its sending server and then reads and sets the parameters' and trigger values.
This is done by callin the "setTP" function.
It also waits for the PC to start its recieving server.
*/
void startUp(){
  Serial.println("StartUp: Updating Trigger & Parameters");
  while(!setTP()){
    Serial.println("StartUp: didn't update TPs");
    delay(2000);
  }

  dataClient = dataServer.available();
  // waiting for the PC to connect to the data server
  //(and allow sending data to the PC)
  while(!dataClient){
    delay(1000);
    Serial.println("Waiting for data Client at startup");
    Serial.println(dataClient);
  }
}
/*
This function calls the "setTP" to read data from
the tpServer update the parameters values.
*/
void refresh(){
  Serial.println("Updating Trigger & Parameters");
  if(!setTP()){
    //Serial.println("Didn't update TP");
  }
  if(!dataClient){
    //Serial.println("dataClient not connected");
  }
}
/*
This function calculates the trigger at a given time based
on the trigger fields sent from the PC finally it returns its value.
*/
double getTrigger(double time){
  switch(trigTyp){
    case 'T': {  // triangle trigger   /\/\/\/\/\/\/
      double pos = fmod(time ,cyc);  // position relative to a cycle
      double m = 4*amp/cyc; // slope
      if(pos<(cyc*0.25)) {
        return dcOff + m*pos;
      }
      else if((pos < (0.75*cyc))&&(pos >= (0.25*cyc))) {
        return dcOff + amp - m*(pos - cyc*0.25);
      }
      else {
        return dcOff - amp + m*(pos - 0.75*cyc);
      }
    }
```

```
    case 'S': {   // sine trigger   ~~~~~~~~~~~~
      return dcOff + amp*sin(2*PI/cyc*time);
    }
    case 'R':{ // ramp trigger  /|/|/|/|/|/|/|/|
      double m = amp/cyc; // slope
      double pos = fmod(time,cyc);  // position relative to a cycle
      return dcOff + pos*m;
    }
    case 'P':{  // square wave trigger  -_-_-_-_-_-_-_
      double pos = fmod(time,cyc);
      double highTime = dutyCyc*cyc;  // duty cycle of '1'
      return dcOff + ((pos >= highTime)? 0 : amp);
    }
    default: return 0;
  }
}

/*
This function reads the data sent from the PC and
updates the trigger and parameters values.
If it was successful it returns true, otherwise false
*/
bool setTP(){
/*
The computer will send 2 lines -
  First line - "O" + trigger line
  Second line - "K" + parameters line
*/
  char c;
  String tLine = "";
  String pLine = "";
  bool tLineRead = false;
  bool pLineRead = false;

  int j = 0;
  int l = 0;
  int i = 0;
  int spaceCnt = 0;
  int oldCnt = 0;

/// the client that reads the data from the server lives and dies in this function.
  EthernetClient tpClient = tpServer.available();


  Serial.println("Enter Check");
  if(tpClient){
    if(tpClient.available()) { // if client data is available to read.

      // Read the trigger line
      c = tpClient.read(); // read 1 byte (char) from client
      if(c == 'O'){ // the trigger line is read now
        c = tpClient.read(); // read 1 byte (char) from client
        while(c != '\n'){
          tLine += (char)c;
          c = tpClient.read();
        }
        tLineRead = true;
      }
       // Read the parameters line
      c = tpClient.read(); // read 1 byte (char) from client
      if(c == 'K'){ // the parameters line is read now
        c = tpClient.read(); // read 1 byte (char) from client
        while(c != '\n'){
          pLine += (char)c;
          c = tpClient.read();
        }
        pLineRead = true;
      }

      tpClient.flush();
    }
    else{
      tpClient.stop();
      return false;
    }
  }
  else{
    tpClient.stop();
    return false;
  }
```

```
  tpClient.stop();
  // Parse the trigger line
  if(tLineRead){
    while(spaceCnt < 7){
      char buf[50] ="";
      j=0;
      // read the chars until space and then based on the count of spaces set the fitting field.
      while(i<tLine.length()){
        if((tLine[i] == ' ')&&(i>0)){
          i++;
          switch (spaceCnt){
            case 0: {
              timeRes = atof(buf); // in micros
              break;
            }
            case 1: {
              dur = 1000000.0*atof(buf); //in micros
              break;
            }
            case 2:{
              trigTyp = buf[0]; // char
              break;
            }
            case 3:{
              amp = atof(buf); // in millivolts
              break;
            }
            case 4:{
              cyc = atof(buf); // in micros
              break;
            }
            case 5:{
              dcOff = atof(buf); // in millivolts
              break;
            }
            case 6:{
              dutyCyc = (atof(buf))/100.0; // in percent
              break;
            }
            default: break;
          }

          spaceCnt++;
          break;
        }

        buf[j] = tLine[i];
        j++;
        i++;
      }
    }
  }
  //infTime = (dur == 0) ? 1 : 0;

  // Parse the parameter line
  if(pLineRead){
    for (int j = 0; j < PARAM_SIZE; j++){
      int k = 0;
      char buf[50] ;
      // read the chars untill space " " and then set the parameters accordingly
      while((l < pLine.length())&&(pLine[l] != ' ')){
        buf[k] = pLine[l];
        l++;
        k++;
      }
      l++;
      params[j] = atof(buf);
    }
  }

  return true;
}

/*
This function sends the data in the output array to
the PC along with the session time in simuTime.
A line is sent - first the simuTime and then the output values,
all seperated by whitespaces (the end of the line is also white space, before newline).
*/
void sendData(){
  delay(20);
  dataClient.print(simuTime);
  dataClient.print(" ");
  for(int i=0; i < OUT_SIZE; i++){
    dataClient.print(output[i],7);
    dataClient.print(" ");
  }
  dataClient.print("\n");
}
```

Measurements Template

The following code is the template code used for the Measurements session.

```
/// Including necessary libraries
#include <SPI.h>
#include <Ethernet.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
////////////////////////////////////////
/// Defining Constants such as size of output paramteres array, size of parameters array that
/// can be changed from the IGG, Refreshing time after which the Galileo accepts any updates.
#define OUT_SIZE 6
#define PARAM_SIZE 10
#define REF_RATE 100
////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////
/// This is a template code for the Measurements session.
/// The parameters are represented by a Double-type array of 10 elements  - "params[]"
/// The data sent back to the PC for displaying it on the plot is represented
/// by a Double-type array of 6 elements - "output[]"
/// The time of the session is represented by the Double-type variable "modelTime"
////////////////////////////////////////////////////////////////////////////////////////

/************** #includes Section ******************/
// add your libraries here as well as #define constants and function declarations
// Any additional functions you add have be declared before their first call in the code
/************** #includes Section ******************/
////////////////////////////////////////////////////////////////////////////////////////
/// Declarations of functions used in the code
bool setParams();
void sendData();
void refresh();
////////////////////////////////////////////////////////////////////////////////////////
/// Variables used in the code
////////////////////////////////////////////////////
/// Variables responsible for server connection interface between the PC and the Galileo
EthernetServer dataServer = EthernetServer(50001);    /// server for sending data to the PC
EthernetServer paramsServer = EthernetServer(50011);  /// server for reading data from the PC
EthernetClient dataClient;                            /// client for writing the data to the dataServer.
////////////////////////////////////////////////////
/// Additional Variables
//bool infTime = 1; // for future uses of adding the option of limited session time
double params[PARAM_SIZE] = {0};  ///  parameters array which the user can change its values during the runtime of the code.
double output[OUT_SIZE] = {0};    ///  output array which the Galileo sends to the PC to display on the graphes.
double modelTime; //modelDur;     ///  used for storing the time of the session. and the duration of the simulation.
double start;                     ///  stores the time at which the session started for future reference.
int refreshParam = 0;             /// counting variable used to count till the REF_RATE contant

/************** Variable Declaration Section ******************/
// Declare your variables here
/************** Variable Declaration Section ******************/
////////////////////////////////////////////////////


/*
This function runs only once at the begining of the code, and is used for setting up the code.
In this template, it performs the following tasks:
  - sets the resolution of the analog readings to 12 bits.
  - start the Serial interface for debugging.
  - starts the servers which exchange data with the PC.
  - stores the current system time in the "start" var.
  - sets the "modelTime" var to 0.
  - updates the paramteres values sent from the PC (using the startUp function).
  - performs the setup user code.
*/
void setup() {
  analogReadResolution(12);
  Serial.begin(9600);
  paramsServer.begin();
  dataServer.begin();

  start = micros();
  modelTime = 0;

  startUp();

  /************** Setup Code Section ******************/
  // Add your setup code to run once
  // Your paramters are stored in an array called params.
  /************** Setup Code Section ******************/
}
```

```
/*
This function runs repetitvely and used for the main code tasks.
In this template, it performs the following tasks in an infinite loop:
  - add to the "refreshParam" counter and if the count reached the "REF_RATE"
    constant's value then call the "refresh" function and reset the counter.
  - perform the user code
  - send the output array to the PC using the "sendData" function.
  - update the time of the session in the "modelTime" var.
*/
void loop() {
  refreshParam++;
  //if(modelTime > modelDur){exit(0);}  // for future use
  if((refreshParam > REF_RATE)){
    refreshParam = 0;
    refresh();
  }

  /************** Main Code Section ******************/
  // Add your main code here to run in a loop
  // To display data on the plot in the IGG, put it in the output[] array, and
  // at the end of each loop-iteration the data will be sent to the PC using the send() function
  /************** Main Code Section ******************/

  sendData();
  modelTime = (micros() - start);
}

/*
This function starts up the server connection, and sets the parameters values
It waits for the PC to start its sending server and then reads and sets the parameters' values. This is done by callin the
"setParams" function.
It also waits for the PC to start its recieving server.
*/
void startUp(){
  while(!setParams()){
    Serial.println("Parameters not set at startup");
    delay(2000);
  }

  dataClient = dataServer.available();
  // waiting for the PC to connect to the data server
  //(and allow sending data to the PC)
  while(!dataClient){
    Serial.println("DataClient not connected at startup!");
    delay(2000);
  }
}

/*
This function calls the "setParams" to read data from the paramsServer update the parameters values.
*/
void refresh(){
  Serial.println("Updating parameters"); /// for debugging
  if(!setParams()){
    //Serial.println("parameters not updated"); /// for debugging
  }
  if(!dataClient){ /// for debugging
    //Serial.println("dataClient not connected");
  }
}

/*
This function reads the data sent from the PC and updates the parameters values.
If it was successful it returns true, otherwise false.
*/
bool setParams(){
  char c;
  String line = "";
  int i=0;

  EthernetClient paramsClient = paramsServer.available(); /// the client that reads the params from the server lives and
dies in this function.
```

```
  Serial.println("Enter Check");
  if(paramsClient){                 // if cleint is connected
    if(paramsClient.available()) { // if client data is available to read.
      c = paramsClient.read(); // read 1 byte (char) from client - will be the 'K' letter for the
parameters string data
      c = paramsClient.read(); // read 1 byte (char) from client
      while(c != '\n'){
        line += (char)c;
        c = paramsClient.read();
      }
    }
    else{
      paramsClient.stop();
      return false;
    }
  }
  else{
    paramsClient.stop();
    return false;
  }

  paramsClient.stop();

  for (int j = 0; j < PARAM_SIZE; j++){
    int k = 0;
    char buf[50] ;
    // read the chars untill space " " and then set the parameters accordingly
    while((i < line.length())&&(line[i] != ' ')){
      buf[k] = line[i];
      i++;
      k++;
    }
    i++;
    params[j] = atof(buf);
  }

  /// for future use
  //modelDur = 1000000*((params[0] == 0) ? 0 : params[0]);
  //infTime = !(params[0] == 0);

  return true;
}

/*
This function sends the data in the output array to
the PC along with the session time in modelTime.
A line is sent - first the modelTime and then the output values,
all seperated by whitespaces (the end of the line is also white space, before newline).
*/
void sendData(){
  delayMicroseconds(500);
  dataClient.print(modelTime);
  dataClient.print(" ");
  for(int i=0; i < OUT_SIZE; i++){
    dataClient.print(output[i],5);
    dataClient.print(" ");
  }
  dataClient.print("\n");
}
```

Monitor Code

The following code is the used for the Monitor session.

```
/// Including necessary libraries
#include <SPI.h>
#include <Ethernet.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
//////////////////////////////////////
/// Defining Constants such as size of output paramteres array, size of parameters array that
/// can be changed from the IGG, Refreshing time after which the Galileo accepts any updates.
#define OUT_SIZE 6
#define PARAM SIZE 6
#define TIMEOUT 100
#define REF_RATE 1000
//////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////
/// This code is used for the Monitor session, and based on the ModelTemplate. It simply reads and sends the analog input
/// measurements to the PC, according to the channels sleceted by the user to display,
/// If a channel shouldn't be displayed its input won't be measured
////////////////////////////////////////////////////////////////////////////////////////////
/// Declarations of functions used in the code
bool setParams();
bool sendData();
void refresh();
void send();
////////////////////////////////////////////////////////////////////////////////////////////
/// Variables used in the code
//////////////////////////////////////////////////////
/// Variables responsible for server connection interface between the PC and the Galileo
EthernetServer dataServer = EthernetServer(50021);      /// server for sending data to the PC
EthernetServer paramsServer = EthernetServer(50031);    /// server for reading data from the PC
EthernetClient dataClient;                              /// client for writing the data to the dataServer.
//////////////////////////////////////////////////////
int refreshParam = 0;                 ///  counting variable used to count till the REF RATE contant
int analogPins[] = {0,1,2,3,4,5};     ///  analog pins indices used in the main code.
double params[PARAM_SIZE] = {0};  ///  parameters array which the user can change its values during the runtime of the code.
double output[OUT_SIZE] = {0};        ///  output array which the Galileo sends to the PC to display on the graphes.
double modelTime; // modelDur         ///  used for storing the time of the session. and the duration of the simulation.
double start;                         ///  stores the time at which the session started for future reference.
//////////////////////////////////////////////////////
/*
This function runs only once at the begining of the code, and is used for setting up the code.
In this code, it performs the following tasks:
  - sets the resolution of the analog readings to 12 bits.
  - start the Serial interface for debugging.
  - starts the servers which exchange data with the PC.
  - stores the current system time in the "start" var.
  - sets the "modelTime" var to 0.
  - updates the paramteres values sent from the PC (using the startup function).
*/
void setup() {
  analogReadResolution(12);
  Serial.begin(9600);
  paramsServer.begin();
  dataServer.begin();
  start = micros();
  modelTime = 0;
  startup();
}
/*
This function runs repetitvely and used for the main code tasks.
In this code, it performs the following tasks in an infinite loop:
  - delays the work for a half millisecond.
  - add to the "refreshParam" counter and if the count reached the "REF_RATE".
    constant's value then call the "refresh" function and reset the counter.
  - perform the main code of measuring the analog inputs and sending the data to the PC:
    - first send the time of the measure.
    - then for each analog input, if the user wishes to display it, measure and send the value.
    - otherwise send the value of 0 and coontinue.
    - The data is sent in a form of line with each value separated by space
      (at the end there is space as well before newline).
  - send the output array to the PC using the "sendData" function.
  - update the time of the session in the "modelTime" var.
*/
void loop() {
  delayMicroseconds(500);
  refreshParam++;
  if((refreshParam > REF RATE)){
    refreshParam = 0;
    refresh();
  }
  /************** Main Code Section ******************/
  //start = micros();
  dataClient.print(micros()-start);
  dataClient.print(" ");
  for(int i = 0; i < OUT_SIZE; i++ ){
    if(params[i]){
      dataClient.print(analogRead(analogPins[i]));
      dataClient.print(" ");
    }
```

```
    else{
      dataClient.print("0");
      dataClient.print(" ");
    }
  }
  dataClient.print("\n");
  /************** Main Code Section ******************/
}

/*
This function calls the "setParams" to read data from the paramsServer update the parameters values.
*/
void refresh(){
  Serial.println("Updating parameters");
  if(!setParams()){
    Serial.println("Didn't update params...");
  }
  if(!dataClient){
    Serial.println("dataClient not connected");
  }
}

/*
This function starts up the server connection, and sets the parameters values
It waits for the PC to start its sending server and then reads and sets the parameters' values. This is done by callin the
"setParams" function.
It also waits for the PC to start its recieving server.
*/
void startup(){
  Serial.println("Updating parameters");
  while(!setParams()){
    Serial.println("Didn't update params...");
    delay(1000);
  }

  dataClient = dataServer.available();
  while(!dataClient){
    delay(1000);
    Serial.println("Waiting for data Client at startup");
  }
}

/*
This function reads the data sent from the PC and updates the parameters values.
If it was successful it returns true, otherwise false
*/
bool setParams(){
  char c;
  String line = "";
  int i=0;

  EthernetClient paramsClient = paramsServer.available();
  Serial.println("Enter Check");
  if(paramsClient){
    if(paramsClient.available()) { // if client data is available to read.
      Serial.println("Reading...");
      c = paramsClient.read(); // read 1 byte (char) from client
      Serial.println("Read a char");
      while(c != '\n'){
        line += (char)c;
        c = paramsClient.read();
      }
    }
    else{
      Serial.println("paramsClient not available");
      paramsClient.stop();
      return false;
    }
  }
  else{
    paramsClient.stop();
    Serial.println("paramsClient not true. Trying again...");
    return false;
  }

  paramsClient.stop();
  Serial.print("line is: ");
  Serial.println(line);
  for (int j = 0; j < PARAM_SIZE; j++){
    int k = 0;
    char buf[50] ;
    while((i < line.length())&&(line[i] != ' ')){
      buf[k] = line[i];
      i++;
      k++;
    }
    i++;
    params[j] = atof(buf);
  }

  return true;
}
```