Creating a Wiring Framework for Embedded GUI Programming Course

Csaba Zoltán Kertész
Department of Electronics and Computers
Transilvania University
Brasov, Romania
csaba.kertesz@unitbv.ro

***Abstract*—This paper presents the results of implementing the Wiring unit for Embedded Wizard, which offers the possibility of using functions familiar from Arduino projects inside a high level GUI building and prototyping environment. The unit allows students for easier microcontroller interfacing of their embedded GUI applications, so they can focus on studying UI and UX design concepts and doing online or offline prototyping of their applications without the need for extensive hardware testing.**

During the development of the Wiring-like framework also a large-scale study was performed about the usage of the framework in Arduino projects. Almost 200,000 Arduino source code files were analyzed from public GitHub repositories, and the findings about the topmost used functions and libraries are also presented in the paper.

***Index Terms*—embedded systems, GUI, Arduino, Embedded Wizard, online education, Arduino function usage**

I. Introduction

Embedded graphical user interface (GUI) programming modules are becoming and important part of embedded software and microcontroller programming course syllabuses [1]. User interface (UI) and user experience (UX) design are increasingly in demand for embedded electronics applications, so teaching at least some basic concepts is a must in an Electronics Engineering curricula.

In this paper, I am presenting a software framework which was developed for Embedded Wizard, the GUI development environment of choice for the new GUI programming module. This framework mimics the Wiring framework made popular by the Arduino development ecosystem.

Although the efficiency of using Arduino for teaching microcontroller programming itself is debatable, the use of it in connected fields like instrumentation, measurement, sensors, robotics, etc. is of great success [2]–[5]. Students rapidly get familiar with the interface and can easily focus on the actual problems and not the microcontroller development itself.

The same idea propelled the need for this framework to be adapted for the embedded GUI course. Although it is an embedded development course, the main focus is on the UI and UX concepts and low level implementation stuff can prove a challenge, distracting the students' attention from the high level issues. Using a familiar framework, like the one used in Arduino, can lead to fast development of the low-level microcontroller interfacing and leaves more time to focus on the course objectives.

The proposed framework is adapted to the needs of this course, and a thorough analysis was performed on the usage of Arduino functions in public open source projects to determine what functions need to be implemented. Results of this analysis are also presented in the paper.

Finally, a survey among the students involved in the GUI module are also presented, students were very appreciative of the new module and although the learning curve was high also the satisfaction levels were higher than average.

II. Embedded GUI Course - Overview

Three possible solutions were considered for embedded GUI programming (using the STM32F746-Discovery board):

- **TouchGFX** ST's own graphic library and designer integrated with STM32Cube[1]
- **STEmWin** an adaptation of the SEGGER EmWin graphic stack for STM32[2]
- **Embedded Wizard** a platform-independent high performance GUI design solution from Tara Systems[3]

TouchGFX is a C++ framework with a UI designer which generates C++ code and any additional user code is added directly in the usual IDE in C++. This is the most direct approach and is best optimized for STM32 microcontrollers. Testing and debugging the application must be done directly on the target development board.

STEmWin uses a two-step approach in which UI is designed in the dedicated IDE, with the possibility to add user code directly there. The whole code can be compiled directly from the IDE and run on the target board, or the whole GUI application can be simulated on the host computer and debugged with Visual Studio.

Embedded Wizard also employs a similar approach, however the high level UI programming is done in its own programming language called Chora. This is a high level object-oriented programming language specifically designed for GUI operations. The UI and the user code is then compiled in Embedded Wizard into ANSI C code, which can then be compiled with for the target with the ST build environment. Besides compiling and running on the target, Embedded Wizard also includes a complete interpreter for Chora, allowing

[1]https://www.st.com/en/development-tools/touchgfxdesigner.html
[2]https://www.st.com/en/embedded-software/stemwin.html
[3]https://www.embedded-wizard.de

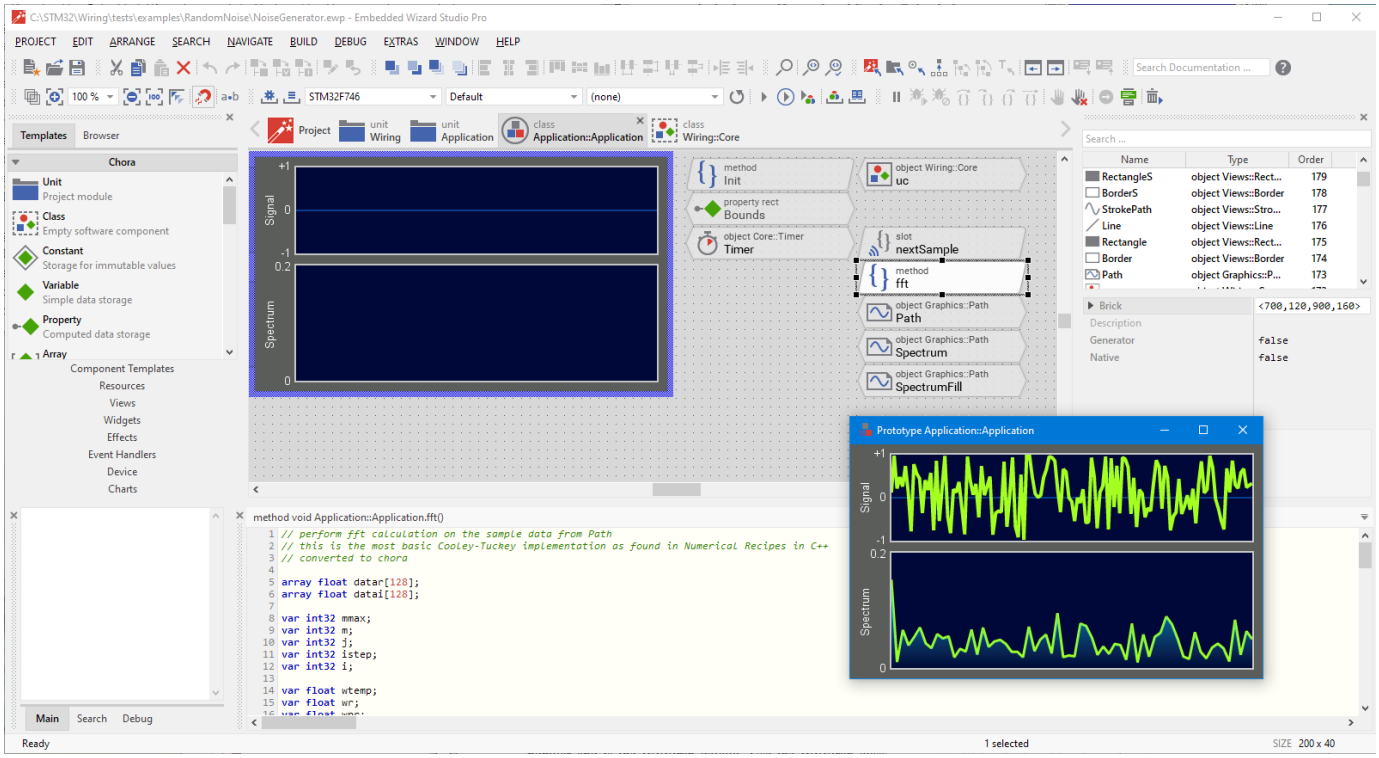


Fig. 1. Embedded Wizard with an example Wiring project and active prototyper window

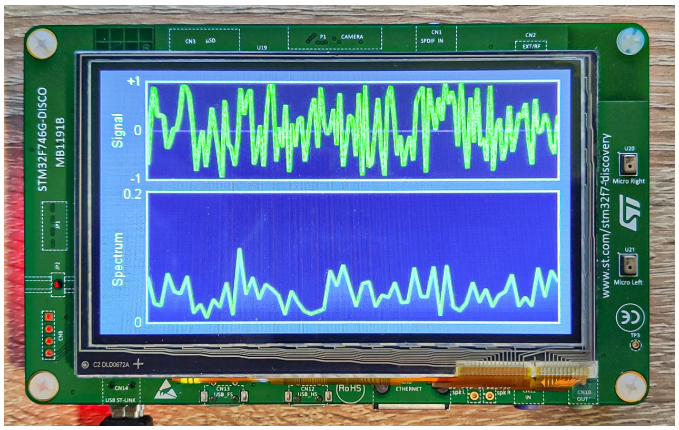


Fig. 2. STM32F746NG-Discovery board running the example application

live testing of the UI. This means that the entire screen instantly reacts to any code change inside the editor. Also, the full UI application with every animation and executable code can be run inside an integrated prototyper without the need for compiling or connecting to the target.

Due to the COVID-19 pandemic, the ability to do some or all of the laboratories online has become an important factor in designing new course materials. Because of this, the choice for GUI development environment went with Embedded Wizard. This allowed for all the development to be done (if needed) online on the students' own computers, prototyping and debugging fully on the host, and only in the end running the projects on the target.

An example Embedded Wizard project (displaying a random signal in time and frequency) running in the built-in prototyper is presented in Fig. 1. The very same application compiled and run on the target (without any other intervention necessary) is shown on Fig. 2. Applications run and offer the exact same UI look and feel both on host and target. The color cast in the photo is due to the reflection in the camera.

III. Wiring Framework Usage

The Wiring framework was initially developed for AVR microcontrollers, and the functions defined in the framework are very much oriented towards the capabilities of this microcontroller [6]. When I started developing a similar framework for Embedded Wizard using an ARM processor, immediately it became obvious that not all functionality is needed for the GUI programming course. Reasons for this include:

- Implementation differences between AVR and ARM;
- Language differences between Chora and C;
- Functionalities already included in the Mosaic framework;
- Functionalities for which better Chora-oriented alternative can be implemented.

And of course, not all functions are really needed and used by the students in their projects. This in particular has driven a need for doing a statistical overview of the usage of Wiring functions in general.

For usage statistics, a very good source to start with are the open source projects hosted on GitHub. The base repositories,

excluding any forks, were added into a public dataset hosted on Google BigQuery, for large-scale data mining operation on them using a cloud-based distributed engine. This BigQuery dataset was successfully used for various code quality analysis [7], code snippet provenience [8], finding the top programming languages [9] or even the most used words in various programming languages [10].

The GitHub contents dataset on BigQuery is a very large one (well above 1 TB for the contents table), so a filtered subset should be analyzed in detail. Standard SQL queries can be used to filter the results as needed. Filtering can be done on file type, size, and other metadata before even starting the code analysis. All the above-mentioned studies employ some sort of such filtering. For example, searching for the most common words in a programming language is done by first selecting only the files with the corresponding extension. Wiring functions are implemented and used in C/C++ code, so it would be a good start for searching for them in C/C++ files only. However, as they are mostly used in Arduino projects, the source files will have an '.ino' extension instead of the more traditional '.c' or '.cpp'. Because of this, existing studies ignored Arduino projects completely.

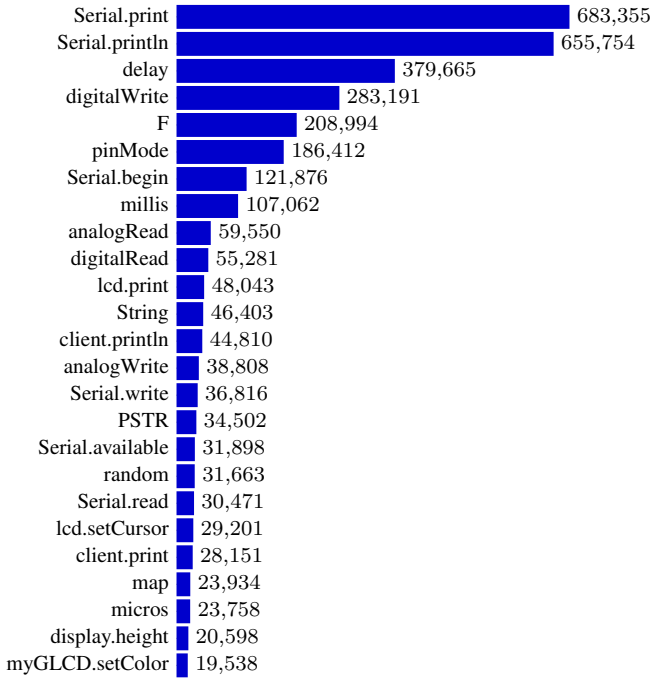
To search for the most used Wiring functions, first, a table with all '.ino' files is generated with the following SQL query:

```
SELECT repo_name as repo, ref, path
FROM `bigquery-public-data.github_repos.files`
WHERE RIGHT(path, 4) = '.ino'
```

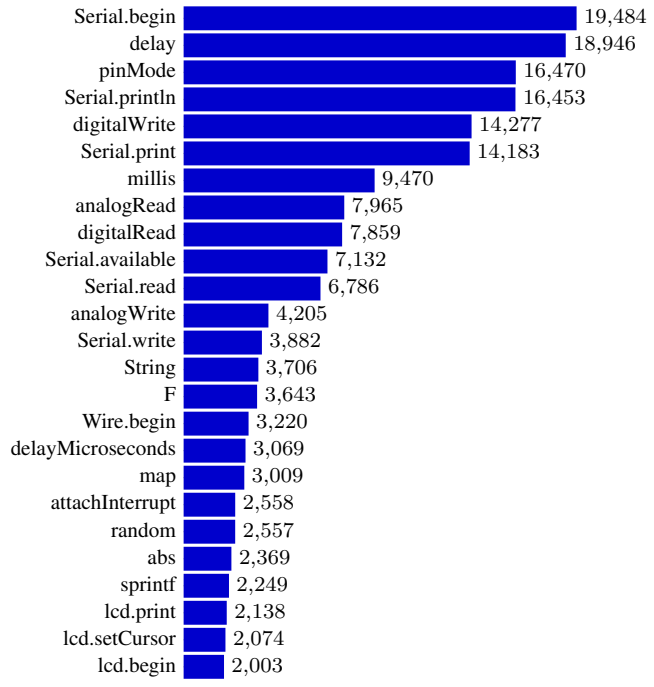
Analyzing the content of the files themselves is a bit more cumbersome, as C/C++ code is notoriously hard to parse without relying on an external preprocessor and compiler [11]. So instead of parsing completely these files, a compromise solution is to use a small Python script to search for any identifiers that would look like a function call. To avoid listing also function declarations, only identifiers that follow an operator or delimiter are to be considered (identifiers directly after a type name are ignored). This of course also includes function-like macros, but these also can have some significance for the usage statistics.

The Python scripts directly access the BigQuery database through the `google.cloud.bigquery.Client` API for getting the names and path of '.ino' files. Afterwards, the file content is read directly from GitHub to reduce data processing ('.ino' files are just a very small portion of the available GitHub content). At the writing of the paper, there were 198057 files available from 24988 repositories. Given that BigQuery GitHub dataset is updated weekly, the actual counts can change quite fast, but one can safely assume to have only a small impact on the usage ranking.

The scripts and the results are available in the https://github.com/kcs/ino-analysis repository, due to size constraints I am reproducing here only the top 25 ranked functions (see Fig. 3). The usage count of the functions decreases rapidly, so even this graphic is quite conclusive as to which functions should be first and foremost added to the GUI Wiring framework. Functions are ranked both by absolute number of function calls and by the number of repositories that make at least one call to the function. There are some differences in the two lists, as there are some Wiring functions (initialization for example) that are required only once to be called, but without them other functions would not work. Some other functions are also intensively used only in a handful of repos, while the



(a)



(b)

Fig. 3. Top 25 functions used in Arduino projects by total number of calls (a) and by number of repositories they are used in (b)

vast majority of repos do not use them at all. For example, the myGLCD.setColor function ranked 25th in overall use, but only 58 repos ever use it.

Besides the function usage, it is also interesting to see the top most used libraries in Arduino projects. Due to the nature of the Arduino IDE regarding the linking of libraries, a good clue to measure library usage is to parse the '.ino' files for `#include` directives. The top 10 libraries resulted from the parsing of the same files as before are presented on Fig. 4. The most important libraries used are not surprisingly for the SPI and I²C (Wire in Arduino terms) peripherals.

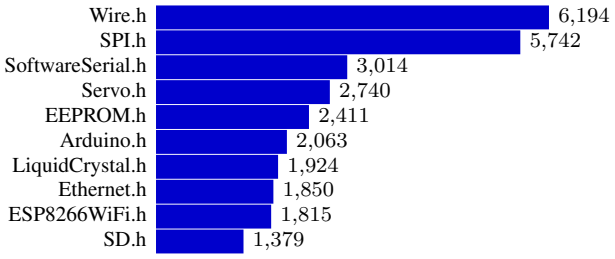


Fig. 4. Top 10 libraries used in Arduino projects

IV. Implementing the Wiring functions

When starting the design process of the Wiring framework adaptation for Embedded Wizard, a couple of design choices had to made regarding what to implement and what differences to the original scheme should be allowed. First, the Chora programming language's paradigm differences had to be addressed. Chora is fully object-oriented, in contrast to C++ which is both procedural and object-oriented. As such, there is no concept for global functions in Chora. It has some built-in functions which act on a global scope, but only object methods can be implemented. This means that the Wiring core functions could not be implemented as global functions, so a dedicated Wiring::Core class was devised for holding these functions. Simple calling of functions like `digitalWrite` is not possible, instead a call to a `digitalWrite` method of an object must be issued. This object, of course, can be a global autoobject. One catch is that the user must ensure to keep a reference to this object throughout the life of the GUI root object, otherwise it can be garbage collected. The objects defined in the Wiring framework, of course, can be implemented as Chora objects.

The second design choice was about which functions to be implemented. This started from the analysis presented before, from which the most popular functions were determined, and only those with a high probability to be used by the students were implemented. However, this list was further filtered, as some of the more popular functions were deemed unnecessary for the GUI integration.

For example, the LiquidCrystal library is unnecessary as we already have a full color graphic display for building GUI applications. Also, the Arduino library and the ESP8266WiFi is of no use on the STM32 target. After careful thoughts, only the SPI, I²C and Serial libraries were considered to be necessary to implement. For SoftwareSerial was not the case, as it is specially thought for AVR microcontrollers having only one hardware UART which is used for debugging purposes. The STM32F746 microcontroller has multiple USART interfaces even on the Arduino headers which can be used for any application needing it. Also, the `Serial.print` and `Serial.println` functions, which are the mostly used functions in Arduino projects, are actually used for debugging purposes as they are connected through the USB interface directly into Arduino IDE. The SoftwareSerial library (which gives access to other serial connections on the Arduino header) is much less used anyway. For debugging purposes, however, Chora also has the built-in `trace` command which is by default uses the USART which is connected through ST-Link to the host computer, so it is no need for another serial debugging feature. The built-in Chora debugging features are more powerful anyway.

Of the other popular functions the `F` and `PSTR` macros are AVR specific, String constructors are not needed, because of the built-in string handling of Chora, and LCD functions are not needed either, the GUI display is completely driven by Embedded Wizard.

Some other functions, even though are very used in Arduino environments, were deliberately left out to force students using the Chora alternatives for them. The timing functions (`delay`, `millis`, etc.) do not have direct counterparts in Chora, but one of the most important features for students to learn in GUI development was the signal-slot mechanism and effect handler, which covers more than enough any timing functionalities needed.

Another design choice in the development of the wiring functions was to implement the whole functionality entirely in a single Chora unit. According to the Embedded Wizard manual [12], the normal approach to integrate the GUI application written in Chora with the low level microcontroller drivers is to use the Device Interface template. This template offers 3 mechanisms to for interaction: commands, event handlers, and properties. Out of the 3, the easiest to use are commands which use a native function call directly to low level functions implemented in C. Event handlers will channel an external event or an interrupt into the Mosaic DispatchEvent-HandleEvent queue, but of course they need proper multi-threaded synchronization in the main loop. Device interface properties are the most complex interfaces, allowing bidirectional communication between Chora code and C code, and offering the best integration into Chora. They also need complete knowledge of the C code generation behavior of Embedded Wizard.

Using these Device Interface templates, however, increases the application development complexity, needing code tweaking both in Embedded Wizard and in an external editor (like STM32CubeIDE) and synchronizing both of them. This extra complexity was deemed unnecessary for educational purposes, the main focus of the course being on the GUI development concepts. Because of this, every Wiring function was imple-

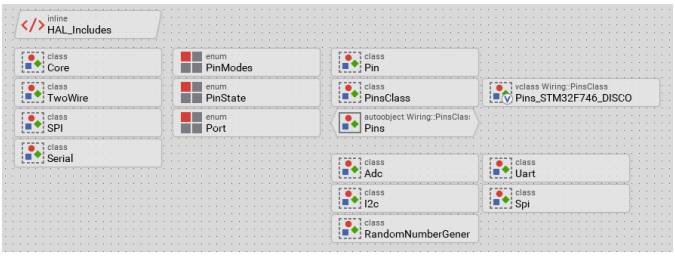


Fig. 5. Wiring unit structure

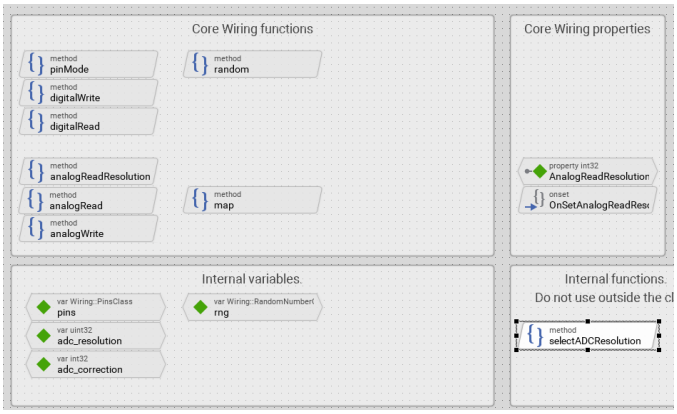


Fig. 6. Core class structure

mented in Chora, and low-level interfacing was included in there using `native` statements. This allows mixing pure C code and connecting to low-level drivers written in C inside Chora code.

The implemented Wiring unit (Fig. 5) includes the Core class for the core wiring functions, the SPI, TwoWire and Serial units for the corresponding peripheral access. There are also some internal units containing low level functions. The most important of these is the Pins unit, which contains the pin mapping for the on-board Arduino headers. This is board dependent, and it was implemented to make full use of Chora features. They can be used through normal property-based interfacing, and what makes is truly portable is the variant dedicated to the Discovery board used in the laboratory. A variant is a Chora feature which transparently overloads a class based on the build profile used. This allows to correctly map pins and pin functions (normal and alternate) to those used on the given board, but also to seamlessly work in the Windows based prototyper where actual pin functionality does not make sense and has just a simulated behavior.

The Core class (Fig. 6) contains the implemented core functions. Not all wiring functions were deemed necessary to be implemented, so it contains only a subset of all the functions. These are:

- `pinMode` configures a pin (numbered according to Arduino header board) to either input, output, analog or alternate function - the latter is not a usual Wiring feature, but it is the STM32 GPIO behavior, and is normally only used by the peripheral units;
- `digitalWrite` sets or resets an output pin;
- `digitalRead` reads a boolean value from a digital input pin;
- `analogRead` reads an analog value from an ADC input pin;
- `analogWrite` configures a PWM output pin which with proper filtering can give a desired analog voltage;
- `random` is a special case, as normally the random functions generates a pseudo-random number, but we already have a Chora built-in function for pseudo-random numbers (`math_rand`), so the function from the Wiring::Core will generate a true random number, using the STM32 hardware Random Number Generator;
- `map` will map a number from one range to another range (usually used to convert ADC range into corresponding voltage value).

All the implemented functions were tested, and an example Embedded Wizard project was created for each of them, covering every available laboratory module. The examples are available alongside the Wiring unit source code and are configured to fully use the build environment that comes along with the Embedded Wizard platform package for the STM32F746 Discovery board. Only a few minor modifications have to be made to compile and run the examples on the target.

V. Students' feedback

Embedded GUI development was introduced as a module of the Software for Embedded Electronics course, at the Faculty of Electrical Engineering and Computer Science of Transilvania University, Brasov. This course essentially is about advanced development techniques for microcontrollers. Students already had previous courses of Computer Architecture and of Microcontrollers, so they were proficient in basic development techniques using AVR and PIC microcontrollers and especially the Arduino platform. The other modules of the course, before the GUI development module, involved low level development for ATxmega and STM32 microcontrollers using direct register access and low-level drivers. ATxmega microcontrollers were programmed using Atmel Studio with direct register programming and a few examples from the Advanced Software Framework[4]. The STM32 microcontrollers were programmed with STM32CubeIde[5] using the HAL drivers included in STM32Cube.

At the end of the course, a survey was conducted about their experiences with both low level development and GUI development. In total, 21 students responded to the survey, grading their satisfaction levels from 1 to 5 for the three aspects of the laboratory work: programming an ATxmega128B1 with direct register access, programming an STM32F746 using HAL drivers from STM32Cube and finally GUI programming with Embedded Wizard, using the Wiring library for peripheral access. The results are shown on Fig. 7. For the low level

[4]https://asf.microchip.com/docs/latest/
[5]https://www.st.com/en/development-tools/stm32cubeide.html

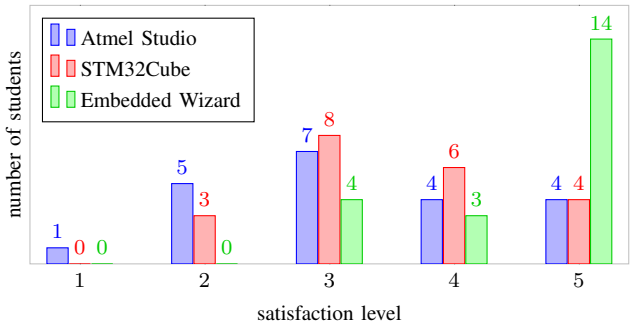


Fig. 7. Distribution of the students by their satisfaction levels on the various development environments

programming modules we can observe a normal distribution of the satisfaction levels, however the GUI programming had a pretty obvious takeoff among the students with two thirds of the students affirming very satisfied with the outcomes of this module.

This survey result is of course skewed by the impact of COVID-19 restrictions. Most of the laboratories were carried out online, students working at home, using simulators instead of running their projects on the hardware target and hardware tests were carried out only occasionally. Because of this, the low level programming modules were seriously impacted by the rather cumbersome simulation environments.

Using the Wiring unit in Embedded Wizard, students were able to access the microcontroller peripherals through a proven and familiar interface, and could focus their development process entirely to the very reliable and easy to use prototyping mechanism of Embedded Wizard. This resulted in the higher satisfaction levels for the GUI programming module compared to the low level programming modules.

Students' opinion was also expressed in the open-text feedback next to the self-grading of their satisfaction with the course. Here are a few remarks about using Embedded Wizard:

- *"It is a very interesting environment, but GUI programming is more arduous than programming an Arduino."*
- *"Personally, I like GUI programming more compared to Arduino."*
- *"It is rather hard adapting to Embedded Wizard, but after a short while the process becomes easier."*
- *"It is not an easy-to-use tool, it is completely different from other tools we learned so far, but it is very interesting."*
- *"It is very intuitive, I liked working in this environment; everything I intended to build, I did easily and fast."*

This points out the very diverse opinions about the difficulty of GUI programming, but by the end of the semester the satisfaction levels of the students show that they adapted quite well to the new environment. In this adapting process the Wiring unit was really helpful for them, because they could concentrate only on the GUI programming tasks and leave the microcontroller interfacing to the already familiar functions.

VI. Conclusions

In this paper I have presented a framework that mimics the Wiring framework used in Arduino programming which was developed for a GUI programming course, using Embedded Wizard as the main GUI building tool. This tool proved very advantageous for fast GUI development with full in-host prototyping capability, making it a good choice especially for mixed (online and offline) laboratories.

While the GUI building capabilities of Embedded Wizard are excellent, the low-level microcontroller interfacing is rather cumbersome, and the proposed Wiring-like framework makes this interfacing much easier in a simple educational environment. The implemented library and examples were well-received by the students, allowing them to focus only on the graphic design and GUI programming aspects of the course.

The source code of the Wiring unit and the example Embedded Wizard projects are all made available in the https://github.com/kcs/emwi-wiring repository. Examples and configurations are only available for the STM32F746-Discovery board, but can be easily reconfigured for any other STM32 boards. The Wiring unit itself consists only of a single source file and can be easily copied into any new Embedded Wizard project.

References

[1] D. Van Merode, G. Tabunshchyk, P. Arras, and K. Henke, "New teaching approaches in embedded system courses," in *International symposium on Ambient intelligence and embedded systems*, 2015, pp. 24–26.

[2] J. C. Martínez-Santos, O. Acevedo-Patino, and S. H. Contreras-Ortiz, "Influence of arduino on the development of advanced microcontrollers courses," *IEEE Revista Iberoamericana de Tecnologias del Aprendizaje*, vol. 12, no. 4, pp. 208–217, 2017.

[3] P. Plaza, E. Sancristobal, G. Carro, M. Blazquez, F. García-Loro, S. Martin, C. Perez, and M. Castro, "Arduino as an educational tool to introduce robotics," in *2018 IEEE International Conference on Teaching, Assessment, and Learning for Engineering (TALE)*. IEEE, 2018, pp. 1–8.

[4] L. M. Herger and M. Bodarky, "Engaging students with open source technologies and arduino," in *2015 IEEE Integrated STEM Education Conference*. IEEE, 2015, pp. 27–32.

[5] M. A. Rubio, C. M. Hierro, and A. Pablo, "Using arduino to enhance computer programming courses in science and engineering," in *Proceedings of EDULEARN13 conference*. IATED Barcelona, Spain, 2013, pp. 1–3.

[6] C. Reas, B. Fry, and J. Maeda, *Processing: A Programming Handbook for Visual Designers and Artists*. The MIT Press, 2007.

[7] B. Ray, D. Posnett, V. Filkov, and P. Devanbu, "A large scale study of programming languages and code quality in github," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 155–165. [Online]. Available: https://doi.org/10.1145/2635868.2635922

[8] S. Baltes and S. Diehl, "Usage and attribution of stack overflow code snippets in github projects," *Empirical Software Engineering*, vol. 24, no. 3, pp. 1259–1295, 2019.

[9] S. Kumar. (2019) Finding top programming language with bigquery. [Online]. Available: https://towardsdatascience.com/finding-top-programming-language-with-bigquery-dbe96d463d99

[10] A. Kashcha. (2016) Common words. [Online]. Available: https://anvaka.github.io/common-words/#?lang=cpp

[11] Y. Padioleau, "Parsing c/c++ code without pre-processing," in *International Conference on Compiler Construction*. Springer, 2009, pp. 109–125.

[12] P. Banach and M. Schweyer. (2020) Embedded Wizard manual. [Online]. Available: https://doc.embedded-wizard.de/