eXecution PlaTfOrm for Sensors

XPTO Arduino

Diogo Miguel de Jesus Lopes Rodrigues
Instituto Superior Técnico
University of Lisbon
diogo.miguel.rodrigues@tecnico.ulisboa.pt

ABSTRACT

An Embedded System (ES) has a programmable processor, but is not targeted to general-purpose computing, which means is a computer system within a device/product to perform one or more specific tasks, often with computing constraints. The computing power of this systems is immense, ranging from high level processors to limited processing resources that performs actions for a given application. Example of the latter described system is the Arduino, because it's designed to run simple applications. Arduino UNO is an simple system with low resources making it a perfect candidate to test the goal of reusing applications by combining them in a new application. Also, Arduino is the most widely used platform and popular with any kind of user. This work aims to develop an execution platform, composed of a well-defined group of tools, to aggregate simple, independent and tested applications in one application. Basically, automate the process to allow a user reuse applications without needing to study them or create a new application by hand. Arduino platform tools and hidden features were analyzed and studied in order to be fully explored and incorporated.

Author Keywords

System; Microcontroller; Resources; Arduino; Features; Tools; Aggregator.

INTRODUCTION

The evolution of information technologies and their world are still associated with general computing devices such as computers, smartphones, tablets and so.

However, large part of them are ES intended to be incorporated in variable systems rather than general computing devices, such as household appliances, cars, life-support systems, among others.

Their computing power is immense, ranging from high level processors to limited processing resources to perform actions for a given task or application.

Therefore, ES can be implemented by two types of processors:

- Microprocessor: Integrated Circuit (IC) which integrates the functions of a Central Processing Unit (CPU);
- Microcontroller: IC which integrates all the necessary components, such as CPU, memory and programmable Input/Output (IO) peripherals [1];

Since microcontrollers are designed to be dedicated devices to specific applications, this makes them more used and common. Additionally, ES are smaller and low energy consumption.

Some ES (mostly with microcontrollers) don't use the traditional Operating System (OS) kernel, like Unix/Linux. This is mainly due to two reasons:

1. Most functionalities aren't needed (scheduling, tasks, preemption, among others);
2. Designed to have limited hardware resources (computing power, memory, instructions and so on);

Nowadays, a smartphone may contain more than 10 microcontrollers for controlling the touch screen, audio, multiple sensors, among others. And a single microprocessor for complex functions.

Prove of Concept

An prototyping platform was chosen for conception, implementation and validation of this work. One of the most relevant platforms is the Arduino. Because of it's affordability, ease of use and open-source platform Arduino is the number one choice for developing creative solutions by engineers, students, designers and so on.

Arduino [2] started as a research project by Massimo Banzi and David Cuartielles [3], in 2005. Furthermore, it's the world's leading open-source hardware and software project, offering a range of software and boards with vast documentation. Has it's own development environment and programming language (C/C++ dialect).

The Arduino UNO [4] board is an example of ES. It's a low cost system with a dedicated microcontroller that can interact with other devices. Table 1 presents the main hardware specifications.

Since UNO is a simple system, right for any user and available in Instituto Superior Técnico (IST) computer science laboratories, it's the right candidate to evaluate the goals of the work.

Table 1. Arduino UNO main hardware specifications.

Microcontroller	ATmega328p
Clock Speed	16 Mhz
Program Memory	32 KBytes
Random Access Memory (RAM)	2 KBytes
Digital IO	14
Analog IO	6
Communication	Serial I^2C SPI

Programming the Arduino only requires two functions. The `setup()`, a initialization routine executed on startup, and `loop()`, an infinite Round-Robin (RR) routine where tasks are executed in circular order.

Motivation and Goals

This work aims to create an execution platform, consisting of a well defined group of tools, to aggregate a set of simple, independent and tested applications in a single application.

This platform will allow an user to reuse already developed applications by combing them in a new application, avoiding the user to write or edit code.

An illustration of the problem is provided by the following example, where each application run on a different UNO.

A first application with a temperature sensor which periodically read it's value and sends it via wireless (Figure 1(a)).

A second application with a presence sensor and when detects a person on the room, acts on the respective light (Figure 1(b)).

The result will be a final application running on a single UNO without any visible difference to users (Figure 1(c)). The main problem is how to aggregate the applications, making it the fundamental objective of this work (Figure 1).

This process has numerous constraints that must solved or minimized, such as code generation or the applications analysis. Therefore, the following requirements where defined:

- Define a template for applications (define guidelines);
- Check for IO pins for collisions/overlap;
- Manage global variables, local variables, functions, and macro definitions (avoid name collisions in the same program space). Function parameters are excluded, the remaining cases are not addressed;
- Platform particular problems (*e.g.* extract the total delay time from the function call);
- Create a final application that contains unique `setup()` and `loop()` functions, and copy the rest of applications code;
- Facilitate user interaction and automate the execution of tools, because it will be a complex process;

Altogether, aggregate two or more applications has several advantages, such as: i) economic: monetary investment on a single system, ii) resource management: greater efficiency with a single system, lower energy consumption, among others.

RESEARCH AND FUNDAMENTALS

A research of the state of art was performed, however no systems, platforms or tools were found.

The only closer solution are schedulers, this developed as libraries. They allow users to define applications as tasks to be executed (and other tweaks). Still, the original problem remains, the user must study and join the application by hand.

That being said, an original and creative solution was developed. The chosen approach is described below.

Code Analyzer

The most popular design for a traditional static compiler is the three phase design, whose major components are the frontend, optimizer and backend [5] (Figure 2).

Figure 2. Three phase compiler design.

The frontend parses source code and builds an Abstract Syntax Tree (AST), among other actions, therefore in order to analyze the applications code an AST is needed.

Since the development of a frontend isn't part of the work, an existent solution is going to be used, the LLVM project.

LLVM Project

The LLVM Project [6] is a collection of modular and reusable compiler tools. It began as a research project at University of Illinois in 2000, to provide a modern compilation strategy of programming languages. Since then, LLVM has gained increasing popularity in a wide variety of areas, such as commercial, open-source projects and academic research [7].

Besides being open-source code, LLVM differs from the traditional compilers, like GNU Compiler Collection (GCC), because these are hard to change or reused in applications.

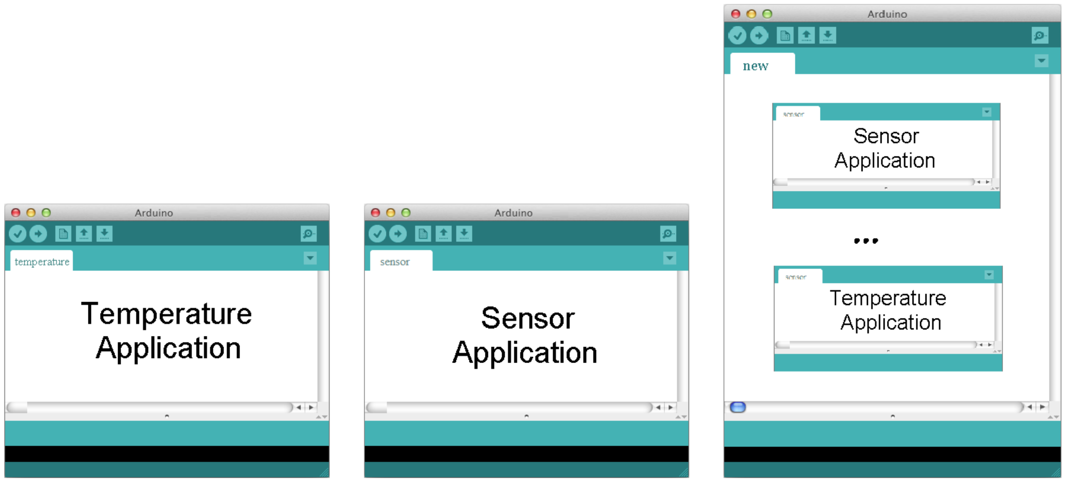
Clang

Clang [8] is a specifically developed LLVM frontend that supports many programming languages, including C and C++, with the following advantages:

- Easy to understand and manipulate AST;
- Simple API to use and access the libraries;
- Faster and low memory consumption than GCC;
- Extensive official documentation [9];
- Available tutorials and examples on internet;

Clang is developed in the newest versions of C++, thus the tools to be developed will exploit this and use `C++11` standard.

Two problems were encountered during the search: i) change variables types is difficult (only possible for predefined types as `int` or `float`), and ii) rename macros is basically impossible because they belong to the independent `Preprocessor` class (their value is evaluated and emitted).



(a) Temperature application (b) Sensor application (c) New Application

Figure 1. Join application 1(a) and 1(b) in the resulting application 1(c).

In sum, two tools are expected to be developed. The first to extract information requiring a basic AST, and the second more complex that produces an AST to be manipulated.

Arduino

yield

The `delay()` function is inefficient because the microcontroller stays on busy-waiting until the specific length of time is achieved. However, Arduino provides an hidden-feature in it's core, the `yield()` function. This is called on background while `delay()` is active.

The `yield()` function is defined as a weak symbol allowing it to be redefined as a user function to execute user actions.

That being said, the `yield()` function can be used whenever high delay time is used by the applications.

arduino-builder

An Arduino application despite being a C/ C++ dialect, isn't ready to be parsed, or even compiled, by any C/C++ compiler. By providing a easy programming language and easy-of-use platform most of the operations are executed in background.

Therefore, a tool for processing Arduino applications was developed, **arduino-builder** [10]. It's a command line tool which allows to parse an Arduino application and convert it into a valid C++ source code, and ready to be compiled.

An Arduino application differs from a standard C++ program in that it misses the `main()`, provided by the Arduino core, replaced by `setup()` and `loop()` functions.

To convert an application to C++ the predefined include `#include <Arduino.h>` is added at the beginning preceded by two line control directives, after generates all function prototypes just before the first function declaration. The original application can be obtained by an algorithm that removes the first three lines and functions prototypes.

As a result, converting Arduino to C++ or vice-versa is doable.

SOLUTION DESIGN

Code Guidelines

In order to have uniform applications development the following set of rules must be ensured.

Additionally, these rules must be clear and easy to apply on already developed applications.

- Use the code template (Listing 1)

Listing 1. Application code template

```
// macros, includes, global variables...

void setup() {
 ...
}

void loop() {
 ...
}

// needed functions
```

- `loop()` only has functions call, no arguments (Listing 2)

Listing 2. Example `loop()` function

```
void loop() {
 read();
 react();
}

// read() e react() functions ...
```

- Use **SoftwareSerial** library for serial communications

  This library allows serial communication on almost every digital pin of the Arduino.

- `.txt` file for listing used IO pins

  This allows a user to have official and reliable documentation about used pins, either in the code or from libraries, without having to analyze or study application, libraries, etc.

This file should contain in each line, either cases:

- The pin number;
- Name (may have spaces) followed by a space (␣) and the pin number;

I^2C and SPI modes of communication have predefined pins but these can be share through a bus and a master-slave strategy.

This isn't the case of the serial communication. Moreover, sharing the native serial instance would result in unwanted and random behavior by the applications (*e.g.* different baud rate). Thus the use of **SoftwareSerial** library is preferable.

Folder Organization

To ease operations that tools will perform an organizational structure of folders has been defined, inasmuch as, working with static paths is much easier than dynamic paths (Figure 3).

At the root, **XPTO** folder containing the following folders:

bin: Location for the tools binaries (like Unix/Linux binaries);

final: Where final application will be generated (`final.ino`);

input: To adding the applications folders to aggregate (App1 and App2 in Figure 3 are examples);

Then, on each application (in **input**), the following sub-folders will be created for:

files: Extracted data and temporary files;

sketch: **arduino-builder** generated files;

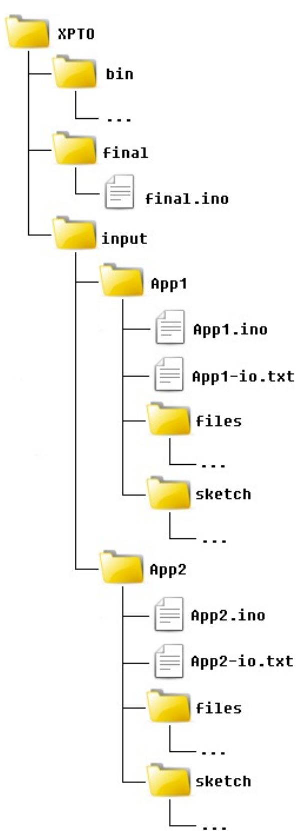


Figure 3. XPTO folder organization.

Platform Particular Problems

Wire Library

Since aggregate applications returns in unwanted behavior when each application should communicate from different systems, an workaround must be developed.

As a particular example of the Arduino platform an alternative library of **Wire** was developed, **Wire2**. The purpose of **Wire2** is to simulate I^2C communication on the same system by multiple applications. In **Wire2** all functions are simulated as normal operations of **Wire**.

SOLUTION

The developed solution consist of five tools, each with a specific task, and a script. These six tools make the execution platform.

Colider

COLIDER is a tool to check IO pins conflict in the set of applications to aggregate. An **input** folder location is the only required argument.

From the **input** folder it navigates through the set of applications, reads each `.txt` file, containing the pins information, and compares if the pin appears more than once.

COLIDER is the only tool that has two types of return, `EXIT_SUCESS` or `1`, while all the others return `EXIT_SUCESS`.

The type of return is associated with the existence of collisions. If it were collisions prints the warning message, which IO pins and returns `1`. Otherwise, simply returns `EXIT_SUCESS` (0).

Extractor

EXTRACTOR is a tool to extract all the relevant applications information. It was developed with the Clang frontend to generate an AST to be analyzed.

EXTRACTOR requires one argument, an application processed by **arduino-builder**. As previously stated, an Arduino application is not C/C++ code nor can be compiled as one, hence the mandatory conversion with **arduino-builder**.

In short, EXTRACTOR extracts:

- Macros names;
- Variables names (global and local)
- Functions names;
- Functions calls names in `loop()`;
- Struct names (structure data type);
- Total delay;
- **Wire** library usage;

Since no full control over the AST is required an simple AST can be created.

First an `CXIndex` must be created. This data structure contains one or more `CXTranslationUnit` (the representation of a source file (*e.g.* as AST)). Then, the visitor pattern is applied to create user callback functions that are called whenever an AST node is reached.

Analyze the Data

While walking the AST the target data can be easily detected, since Clang has a dedicated data structure (`CXCursorKind`) of type enumeration, that identifies any kind of information or entry. `CXCursorKind` identifies namespaces, function calls, variable names, conditional instructions and so on.

Therefore, an auxiliary function is called in the visitor method to save the data in one of the following data structures:

infoDecls: Map to hold delay and flag to state if **Wire** is used;

loopFuncs: Vector of `loop()` functions calls;

macroDecls: Vector of macros;

structDecls: Vector of struct names;

varDecls: Vector of variables names (global and local);

functionDecls: Vector of functions names;

Save the Data

Last, save the data in the data structures, into files in the **files** folder, with the following disposal (`x` is an application name):

x.txt: Functions and variables names;

x.info.txt: Delay and flag of **Wire**;

x.loop.txt: `loop()` function calls;

x.macro.txt: Macros and structs names;

The diagram of EXTRACTOR processing the Blink application (Listing 3) is represented in Figure 4.

Listing 3. Example Blink application

```
#define LED 13

void setup() {
 pinMode(LED, OUTPUT);
}

void loop() {
 piscar();
}

void piscar() {
 digitalWrite(LED, HIGH);
 delay(1000);
 digitalWrite(LED, LOW);
 delay(1000);
}
```

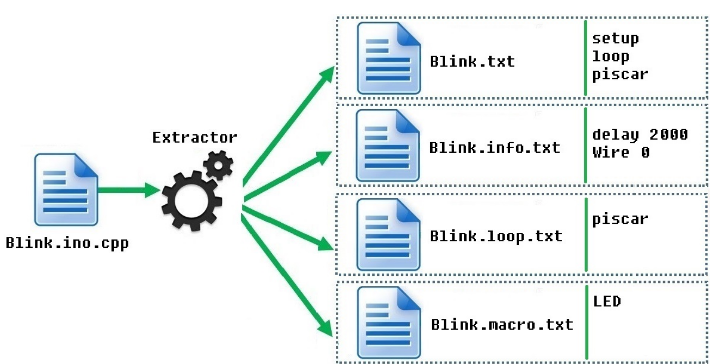


Figure 4. EXTRACTOR diagram on Blink application.

Rewriter

REWRITER is a tool to rename variables, functions and other names accordingly to an algorithm. It was developed with Clang tools to generate an complex AST to be edited.

REWRITER requires the following arguments: an C++ code (already processed by EXTRACTOR), separator "`--`", number that represents index (used by algorithm) and a flag (0/1) to instruct if **Wire** should be change to **Wire2**.

The result is a new file named `x-final.ino.cpp` (`x` is an application name).

In short, REWRITER renames:

- Macros names;
- Variables names (global and local)
- Functions names;
- Struct names (as a type);
- **Wire** inclusion into **Wire2** inclusion;
- **Wire** calls into **Wire2** calls;

The previously mentioned number that represents an index must be a positive integer and unique in the set of applications. This index is used by the algorithm as: `ledState` variable name becomes `ledState_1`, or the macro `MAX_BUFFER` becomes `MAX_BUFFER_1`.

By the application name the extracted data is loaded into two vectors: **changeDecls** and **changeMacros**. The first has variables and functions names, while the second has macros and structs names.

Although the index number can be provided as argument, it may also be read from the **index.txt** file in the **files** folder, by providing "`-`" as argument.

As stated, Clang doesn't offer an easy procedure either to rename macros or change variables types (for rename structs names), consequently an additive algorithm was developed and integrated to rename these specific cases.

Altogether, REWRITER has two cycles, the first using Clang and the second using the complementary rename.

Clang

Since REWRITER must have full control over the AST, an standalone tool must be developed with the help of LibTooling tools. These are a collection of specific developer tools built on top of the LibTooling infrastructure and as part of the Clang [11].

The separator "`--`" is used by `CommonOptionsParser` class that has the responsibility to parse command line arguments related to compilation database and inputs, so that all tools share the same information (*e.g.* source files paths).

REWRITER, instead of EXTRACTOR, uses `FrontendAction` which is an abstract class to execute user specific actions. In this case, to run code over the AST, `ASTFrontendAction` must be used to take care of executing those actions.

Then an `ASTConsumer` must be implemented, to read all the entries in the AST, which is created by `CreateASTConsumer`.

Each `ASTConsumer` correspond to a source file, thereby the correspondent AST, that is recursively visited by the `RecursiveASTVisitor` class. The benefits of `RecursiveASTVisitor` is the entry points or visitor methods for most AST nodes, this means only the relevant node types will have implemented methods.

Last, to edit code there's a tool component that performs source-to-source transformations, the `Rewriter` class. `Rewriter` it's a sophisticated buffer manager that can insert, remove or edit code very accurately from the source locations of an AST node.

The `Rewriter` is created in the `ASTFrontendAction` user class and the `ASTContext` is linked to have the same source code.

When an target node is identified to rename (in `RecursiveASTVisitor`), the `ReplaceText()` method from `Rewriter` is called with the current source location, current name length and the new name.

Complementary Rename

The complementary rename aims to change macros and variables type accordingly to the rename algorithm. Also, if the flag to instruct **Wire** should be change to **Wire2** is true replaces the inclusion directive, since rename function calls is performed by Clang.

Detecting the inclusion directive is easier, by simply comparing the whole line read for `#include <Wire.h>` and once detected replace with `#include <Wire2.h>`.

For the rest of cases, let's see the following example when we want to detect the word "is" in the phrase "This is a phrase". The previously type of detection used for the inclusion, string match, cannot be used because it will have two matches: "Th<u>is</u> <u>is</u> a phrase".

Therefore, regex must be used to match word boundaries ("\b"). In this way "whole word only" search is performed avoiding a wrong detection and rename. Then, regex with the word boundary "\bis\b" applied, only one, and correctly, match happens: "This <u>is</u> a phrase".

This technique can be applied to macros and variable types because they have fixed and unique names on the code.

The diagram of REWRITER processing the Blink application (Listing 3) is represented in Figure 5.

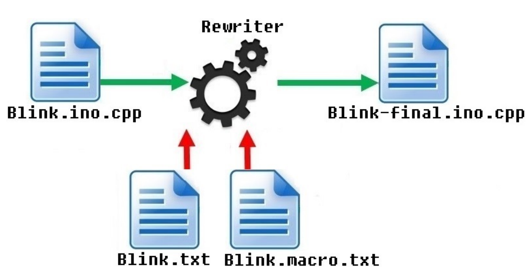


Figure 5. REWRITER diagram on Blink application.

Preparator

PREPARATOR is a tool to convert C++ code (**arduino-builder** processed) into Arduino code. An C++ code is the only required argument.

The result is a new file named `x-final.ino` (`x` is an application name), ready to be merged.

The first step is read and ignore the first three lines. Then, regex is used to detect and remove all functions prototypes, plus commenting the `loop()` function. For instance, detecting loop with regex becomes: "`void loop_([0-9]+)\(\) \{`". The `loop()` is commented because it's no longer use in the final application.

The diagram of PREPARATOR processing the Blink application (Listing 3) is represented in Figure 6.

Figure 6. PREPARATOR diagram on Blink application.

Merger

MERGER is a tool to aggregate Arduino code into one final Arduino application. The **input** folder and a flag to instruct if yield function should be generated are the only required arguments.

The result is a `final.ino` application inside the **final** folder.

From the **input** folder navigates through the set of applications to read the extracted information (`loop()` function calls) and copy the code from `x-final.ino` (`x` is an application name).

While copying the application code from `x-final.ino` filters the inclusion directives to avoid repeating on the final code.

Additionally, if the flag for generating `yield()` is true then this function will be generated for the user. The `yield()` function was previously mentioned.

In short, MERGER does:

- Generate `setup()` and `loop()` functions;
- Generate `loop()` function calls from the applications;
- Generate `yield()`, if true;
- Filter inclusion files (avoid repeating);
- Copy Arduino code;

This new application as only one `setup()` and `loop()` functions, `yield()` function if instructed and also all the code from each application to aggregate.

Script

The developed script allows automation of the process with minimal interaction to tools, since it's a complex process with many settings and arguments.

It's mandatory the Arduino software on user's computer to execute the script and tools.

Before executing the script an user must set two variables in it, `ARDUINO` and `LIBRARIES_DIR`, the first is Arduino software location and the second library folder location. From the `ARDUINO` all necessary folders and files can be obtained.

The script run each tool in the correct order with the right arguments, inquiries user about the **Wire** replacement by **Wire2**, delay and yield, also informs if it were IO collisions by the COLIDER.

Initialization

The initialization phase of the script performs:

- From `ARDUINO` variable get full paths of **arduino-builder**, Arduino core include files, AVR include files (incorporated in the platform) and so on;
- Get all libraries full paths from `ARDUINO` and `LIBRARIES_DIR` variables. These can be the root folder or the **src** folder inside root folder;

Execution

The execution phase of the script executes the tools in the correct order, as following:

1. COLIDER is executed and if no IO collisions exists, the next tools is executed. Otherwise, the user is asked if want to abort or continue (note that collisions may occur like I^2C shared IO pins);
2. For each application **arduino-builder** is executed, then the C++ code file is copied form **sketch** to **files** folder and an index is assigned to each application (starting from 1 and increasing);
3. For each application EXTRACTOR is executed;
4. For each application index is retrieved from file `index.txt`, then if **Wire** is used the user is asked to switch over to **Wire2** and then REWRITER is executed;
5. For each application PREPARATOR is executed;
6. Afterwards, the delay of each application is printed on screen along side with the total delay, the user is asked if `yield()` should be generated, and the answer is saved;
7. Last, MERGER is executed with the yield answer;

Finalization

The finalization phase deletes temporary files and extracted information:

- Deletes all **arduino-builder** temporary files (**sketch** folder), since these files are no longer needed;
- Ask user if temporary files should be deleted (*e.g.* `index.txt`, C++ code files);
- Ask user if extracted information files should be deleted (*e.g.* file with `loop()` functions calls, file with the variables and functions names);

Figure 7 shows the script execution flow (some interactions were omitted).

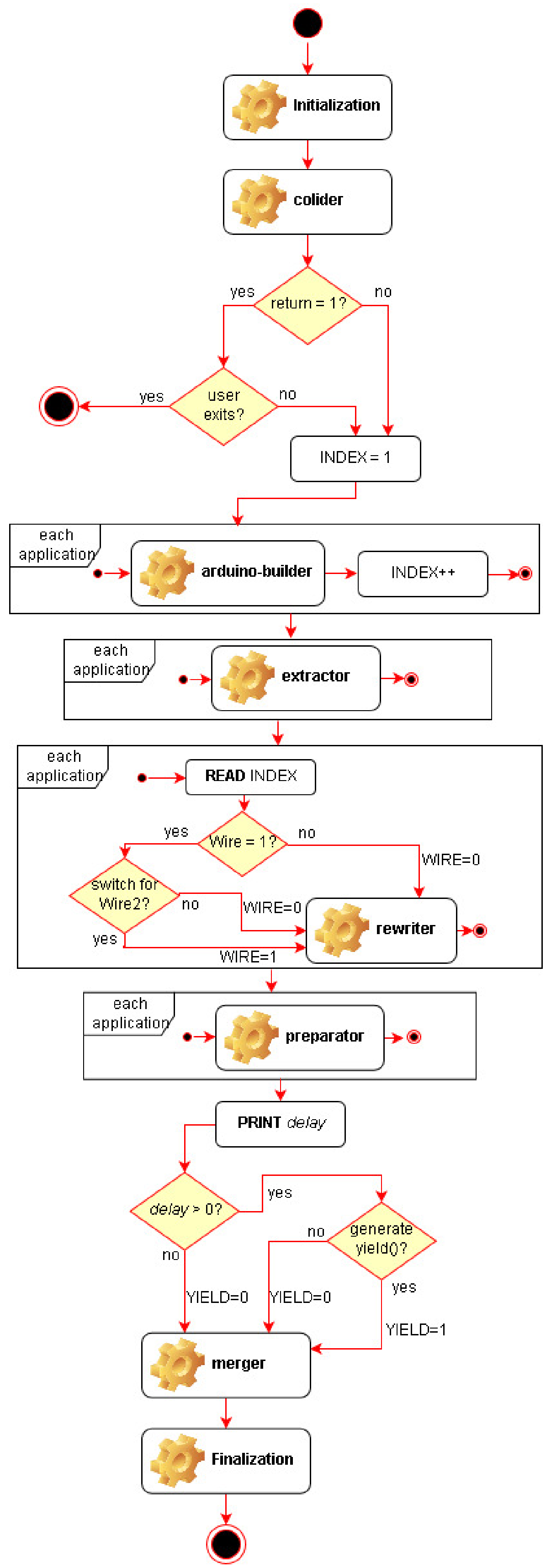


Figure 7. Script diagram flow.

RESULTS ANALYSIS

Several test were performed to evaluate **XPTO**. However, in lack of space, only the most relevant results are demonstrate.

Methodology

For this evaluation, six applications were developed or adapted (Table 2):

- Application 1: Light Emitting Diode (LED) blinks every second;
- Application 2: Gets temperature from DS18B20 [12], calculates last 5 reads mean and sends it via serial;
- Application 3: Simulates an network of traffic lights in I^2C, uses serial for debug and blinks an LED;
- Application 4: Simulates an radio control vehicle by receiving serial commands and acts upon motors;
- Application 5: Periodically gets and sends via nRF24L01 [13] the temperature from DS18B20 [12], blinks an LED every second;
- Application 6: Periodically gets and sends via nRF24L01 [13] the concentrations in the air from MQ-7 [14], blinks an LED every second;

The Communication (Comms.) identify types of protocols used. The delay, in milliseconds (ms), is the `delay()` function calls total time. The Memory is the Program Memory application uses. The execution Time, in microseconds (μs), is the average `loop()` function time within one second of execution (Figure 8).

No input data were received in applications (no data or commands sent to them), but output data is transmitted, either from prints or writes. Communication with sensors were always executed and the rest were not performed (`e.g.` interruptions, button pushes).

Table 2. Applications information.

No.	Comms.	delay (ms)	Memory (bytes)	Execution Time (μs)
1	-	0	862	5,37
2	serial	100	5344	146441
3	I^2C, serial	110	8656	128,32
4	serial	0	4342	25,26
5	SPI	10	3914	8,07
6	SPI	10	2544	8,07

Note, in applications 3, 5 and 6 delay time is bigger than Execution Time, this means delay has a particular condition to be used (`e.g.` interruption not activated) or the timing of delay is bigger than one second.

From these applications a set of three tests were created (Table 3).

Table 3. Test set information.

No.	Application No.	Switch **Wire** for **Wire2**	Use `yield()`
1	1 + 2	No	No
2	2 + 3	No	No
3	4 + 5 + 6	No	No

The direct alternative to **XPTO** is a programmer that must study the applications code in order to aggregate them. However this approach has a benefit, the programmer can apply optimizations in the aggregated application to improve memory consumption and execution time.

Therefore, the aggregation performed by **XPTO** must be evaluated and compared to the existent method, an aggregation by a programmer. Subsequently the resource usage (Program Memory and Execution Time) is going to be compared along side with the productive development.

In addition, three tests were developed as the programmer applications. These with all the possible optimizations [15]:

- Test 1: No optimizations possible;
- Test 2: The **SoftwareSerial** instance is shared by both apps;
- Test 3: The radio instance and buffers are shared by both apps, also only one LED function is used;

Resource Usage

Program Memory

The results of program memory evaluation are in Table 4 and the Close-Max-Min chart in Figure 9. The Close is **XPTO** result, Min is Programmer metric and Max the Basic metric.

The percentage listed is the gain relative to Basic metric.

	Program Memory (bytes)		
No.	Basic	Programmer	**XPTO**
1	862 + 5344 = 6206	5484 (12%)	5484 (12%)
2	5344 + 8656 = 14000	10632 (24%)	10994 (21%)
3	4342 + 3914 + 2544 = 10800	7012 (35%)	7542 (30%)

Table 4. Program Memory results.

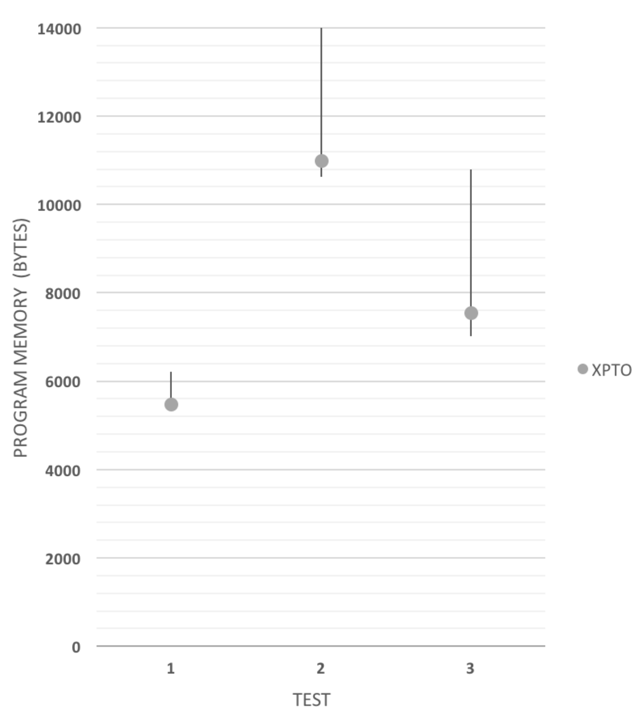


Figure 9. Close-Max-Min chart of Program Memory.

```
void loop() {
 unsigned long contar = millis(); // get the time to count a second

 while(1) {
  unsigned long t = micros(); // get the time for this cycle

  // ... perform loop function calls here

  Serial.println(micros() - t); // print this cycle time
  if (millis() - contar >= 1000)
   while(1);  // one second reached
 }
}
```

Figure 8. `loop()` function to calculate execution time.

The difference between Basic and other two metrics exist because the compiler performs an considerable number of optimizations while generating low-level code, because ES has this resource limited.

Test 1 proves that if two applications are indeed independent then an aggregation can be performed by either methods.

Test 2 had one small optimization that overall didn't make a big impact, because only 362 bytes were improved, although it can be considered relevant.

Test 3, the most complex, is the worst result because applications 5 and 6 share numerous things that can be optimize. This result was expected since merging functions and variables lead to smaller code, thus improving the final application result. It improved 530 bytes in the program memory, considered relevant.

Overall, regarding the program memory, the **XPTO** results are closer to Programmer metrics than Basic and the gain of Programmer was not that critical, therefore these results are positive and acceptable.

Execution Time

The loop used to calculate execution times introduces an overhead on the system, however since all applications and tests were conducted with it, this overhead is ignored to compare the results (Figure 8).

The results of execution time evaluation are in Table 5. The percentage is the gain relative to the Basic metric.

No Close-Max-Min chart were created because of scale fluctuations among results and no additionally information would be inducted.

	Execution Time (μs)		
No.	Basic	Programmer	**XPTO**
1	5,37 + 146441 = 146446,37	146443,05 (0%)	146444,15 (0%)
2	128,32 + 146441 = 146569,32	146451,12 (0%)	146453,33 (0%)
3	25,26 + 8,07 + 8,07 = 41,40	27,28 (34%)	32,21 (22%)

Table 5. Execution Time results.

Once again, the existence difference between Basic and the other two metrics is remarkable. The Arduino core, libraries and so on, all executes background functions. When the applications are merged only one execution of those functions occurs, therefore lowering the execution time.

Test 1 shows one microsecond difference but it can be discarded (since it's so insignificant), thus proving if two applications are indeed independent then an aggregation can be performed by either methods.

Test 2 shows a two microseconds difference, which can be discarded because doesn't affect the overall performance of the final application. This result also evidence a good implementation of **SoftwareSerial**, thus it was the right solution to solve serial communication conflicts.

Test 3, the most complex, is again the worst result. Even so, it's an expected outcome, because the optimizations performed result in smaller code with fewer instructions to be executed, thus lower execution time.

Altogether, in execution time, since **XPTO** results are closer to Programmer metrics than Basic, these results are acceptable.

Productive Development

The optimizations introduced in final applications by a programmer can be relevant. Therefore let's examine if that advantage is profitable or if **XPTO** should be used.

Test 2 had some improvements by sharing one **SoftwareSerial** instance. For this optimization a programmer must read both codes, review the shareable objects and create new application.

A detailed analysis of this test was carried out (Table 6).

	Application 2	Application 3	Total
Lines of code	88	753	841
Objects for review	10	78	88
Macros	3	21	24
Global Variables	4	28	32
Functions	3	29	32
Accounted gain			3 (3%)

Table 6. Information about test 2.

The accounted gain are two macros (receiver and transmitter IO pins) and the **SoftwareSerial** instance.

The programmer had to read a total of 841 lines of code, fairly considerable value for some programmers, proving already to be a waste of time in contrast to the obtained gain.

Leaving aside the numerous lines of code, there were a total of 88 objects for review. Around one third of those are functions, some of them can be quite hard to interpret by programmers in order to assess if they are similar or can be merged in only one function.

The same method can be applied to variables, for instance two variables with the same name on different applications may not perform the same purpose.

Altogether, the time invested on this process did not compensate the overall gain either in program memory and execution time, hence **XPTO** should be used.

FUTURE WORK

XPTO it's a command line platform with a disposition of folders. To use it users only need to edit the script for assign two variables (Arduino dependency), afterwards copy applications to aggregate into **input** folder and last run the script.

In contrast, Arduino is a easy of use graphic interface platform, hence a graphic interface could be developed for **XPTO** which would make it much easier to use and improve user experience.

Static paths associated with the script and tools would no longer exist, since users would drag and drop applications to aggregate, and these processed in real-time. Also, the final application could be generated in any location of the computer.

A detailed analysis of the used communication modes could be performed in order to understand how the IO pins are employed and better inform the user about collisions. This would allow a faster detection of communications IO pins collisions instead or regular IO pins.

CONCLUSION

Building on the conclusions of tests, aggregate simple, independent and tested applications is possible and viable. The obtained results of **XPTO** were as good or equal to that of a programmer. It's an useful tool for reuse already developed applications by creating new ones. **XPTO** aggregates applications promptly and easily.

Moving away from the initial assumptions (independent applications), **XPTO** results aren't that beneficial but nonetheless workable, it's at the choice of user. It can be used for aggregating applications with no concern for optimizations, or when this are possible but not exploited by users.

ACKNOWLEDGMENTS

I would like to thank my supervisor Prof. Alberto Manuel Ramos da Cunha for the opportunity to accomplish this thesis. I would also like to acknowledge Mr. Massimo Banzi and Mr. Cristian Maglie for providing valuable insights regarding the Arduino Platform. Any opinions, findings, conclusions, or recommendations expressed here are those of the author and only the author.

REFERENCES

1. M. Verle, "What are microcontrollers and what are they used for?" in *Architecture and programming of 8051 MCUs*. MikroElektronika, Jan. 2007.
2. Arduino. [Online]. Available: https://www.arduino.cc (retrieved 12 July 2017).
3. D. Mellis, M. Banzi, D. Cuartielles, and T. Igoe, "Arduino: An open electronic prototyping platform," in *Proc. Chi*, 2007.
4. Arduino UNO. [Online]. Available: https://store.arduino.cc/arduino-uno-rev3 (retrieved 12 July 2017).
5. C. Lattner, "LLVM," The Architecture of Open Source Applications. [Online]. Available: http://www.aosabook.org/en/llvm.html (retrieved 13 July 2017).
6. LLVM Project. [Online]. Available: http://llvm.org (retrieved 13 July 2017).
7. C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Mar. 2004.
8. clang: C language family frontend for LLVM. [Online]. Available: https://clang.llvm.org (retrieved 13 July 2017).
9. Clang Documentation. [Online]. Available: https://clang.llvm.org/docs/index.html (retrieved 16 July 2017).
10. Arduino Builder. [Online]. Available: https://github.com/arduino/arduino-builder (retrieved 17 July 2017).
11. LibTooling. [Online]. Available: https://clang.llvm.org/docs/LibTooling.html (retrieved 27 July 2017).
12. Temperature Sensor - DS18B20. [Online]. Available: https://datasheets.maximintegrated.com/en/ds/DS18B20.pdf (retrieved 24 August 2017).
13. Wireless Solution Module - nRF24L01. [Online]. Available: https://arduino-info.wikispaces.com/Nrf24L01-2.4GHz-HowTo (retrieved 24 August 2017).
14. Carbon Monoxide Sensor - MQ7. [Online]. Available: https://cdn.sparkfun.com/datasheets/Sensors/Biometric/MQ-7%20Ver1.3%20-%20Manual.pdf (retrieved 24 August 2017).
15. *Atmel AVR4027: Tips and Tricks to Optimize Your C Code for 8-bit AVR Microcontrollers*, Atmel Corporation. [Online]. Available: http://www.atmel.com/images/doc8453.pdf (retrieved 24 August 2017).