0301-346: APFE Project 3

Final Project Documentation Guidelines

1. Write a description of your main program step-by-step. For example, if you have menus, explain all the menu items. Try to call all operation in the main to show your program features. It should be able to run by reading your step-by-step definition of your main function. You also should explain the followings in your write up:

   - your classes,
   - inheritance between them,
   - why you need overloading and where you used it
   - important functions of the classes
   - other things you want to explain about your program.

2. Fill the following table according to your program. In the "yes/no" column, write yes if you used the related subject. In the "File" column, write the file in which the related subject is used.

	Subject	Yes/No	File Name
Required Subjects	Pointers		
	Classes **(at least 4 classes)**		
	Overloading		
	Inheritance		
	Virtual Functions		
Extra Credit	Standard Template Library		
	File Processing		

3. Zip your ".exe", source files (.cpp and .h files), and project **write-up** (.doc), name it as **"name_surname_project3.zip"** and upload it to my courses dropbox.
4. Include ***this page*** to your project as the first page and submit your **write-ups** and your code

Grading

- Write-up **20%**
- Classes **35%**
- Other Required matters **35%**
    - Overloading,
    - Inheritance,
    - Virtual Functions

- **Code Style** **10%**
    - Name on all files
    - Comments
    - All prototyping done in header files
    - All implementation done is cpp files
    - Separate main file

- Extra Credit **20%** (for the ones who need it)

- **Potential Deductions**
    - Does not compile -30%
    - Submission Guideline Failure -10%

Some Ideas for Final Project

- Calendar
- Hospital Management system
- Student Registration System
- Bank Account
- Geometric Shapes
- Car Registration System
- Museum entrance and acceptance system
- Library Management System
- A Sudoku puzzle solver
- Battleship game
- Chess
- Pokemon
- A program which helps to identify new species
- An appointment tracker
- Simple RPG games

Shane Snover

EEEE 346 Advanced Programming

Dr. Sahin

4 May 2018

Final Project: Arduino Simulator

Introduction

The Arduino platform is an open source hardware and software platform popular among amateur hardware hackers and engineering students due to its simplicity and low cost. This popularity has snowballed through the proliferation of tutorials involving the system. One common problem with the Arduino UNO's toolchain is that it can be difficult to debug application firmware as the hardware has no on-board debugger and the Arduino IDE does not support step-by-step debugging. This project attempts to provide a solution to this problem by implementing a barebones framework wherein an Arduino firmware can be written and run inside of a simulator on one's own computer without the need for expensive hardware like logic analyzers, oscilloscopes, or other tools out of reach of the target audience for the Arduino framework. This project is written in standard modern C++14 which allows it to run cross-platform on multiple operating systems.

Using the Framework

In order to use the framework, a sample file named "*arduino-sketch.ino.cpp*" is provided where the user will find the familiar functions *setup()* and *loop()* which are required to be implemented in every Arduino sketch. The API available to the user is linked through a header file which defines many of the constants and function declarations familiar to users of Arduino.

Current functionality implemented for this project include configuring and toggling general purpose digital input/output pins and communication with the Serial port. The simulator wraps the Arduino API and runs its own implementation to log events and simulate the hardware of the Arduino UNO.

Once the sketch has been written, the user may compile the code within the rest of the framework and execute their binary file from a terminal. The terminal will display any strings printed to the Serial port and the user is able to enter text via the terminal which will be received by the Arduino sketch. This is handled by a text file in the local directory which is written to by a thread scanning the terminal for user input and which is read by another thread inserting data into an internal buffer. When the user is done running the simulated sketch, they may exit by typing "\exit" into the terminal or sending the process a signal SIGINT (on most platforms, this can be done by typing Ctrl+C in the terminal).

After the program has exited, the user can review the log file, by default placed locally in "*log.txt"* which contains data on pin configuration changes and Serial messages transmitted and received. This data may be useful by the user for analysis on the order of events occuring in their sketch. A sample log file is shown in a screenshot in Figure 1 below.

```
INFO: Simulator Starting
SERIAL: Port opened with baud: 9600
DIO: Pin D13 set to Direction[OUTPUT]
DIO: Pin D13 set to State[HIGH]
DIO: Pin D13 set to State[LOW]
SERIAL_TX: Toggling my LED for dayssss
DIO: Pin D13 set to State[HIGH]
DIO: Pin D13 set to State[LOW]
SERIAL_TX: Toggling my LED for dayssss
DIO: Pin D13 set to State[HIGH]
DIO: Pin D13 set to State[LOW]
SERIAL_TX: Toggling my LED for dayssss
DIO: Pin D13 set to State[HIGH]
DIO: Pin D13 set to State[LOW]
SERIAL_TX: Toggling my LED for dayssss
DIO: Pin D13 set to State[HIGH]
DIO: Pin D13 set to State[LOW]
SERIAL_TX: Toggling my LED for dayssss
```

Figure 1. Sample Log File Output

Application Structure

The application consists of three threads which run throughout the lifetime of the program. On start up, the Main Thread of the process spawns two threads with the construction of the *ARDUINO_SIMULATOR* object: the Simulator Context Thread and the RX ISR Context Thread. Following spawning the other threads, the Main Thread begins reading in input from the console and writing it to the filestream at *".terminal"*.

The Main Thread algorithm is pictured in Figure 2 below:

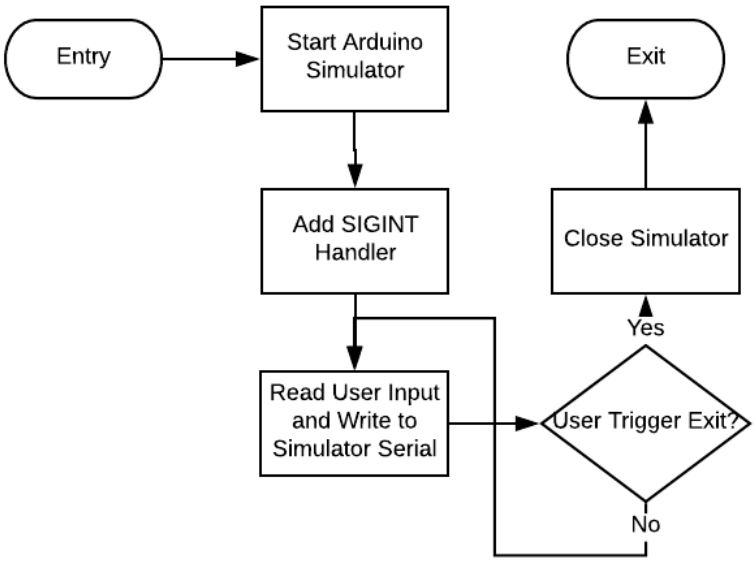

Figure 2. Algorithm Flow Chart for Main Thread

Next, the Simulator Context Thread was simply in charge of running the user's firmware. The *setup* function is called once at start and then the *loop* function is called once every millisecond. If a flag is disabled, it signals to the simulator to close and join it's thread back to the Main Thread.

Finally the RX ISR Context Thread, or Receive Interrupt Service Routine Context Thread, emulates the behavior of the Arduino UNO microcontroller and the interrupt vector hardware which allows the hardware to insert program instructions from a pre-programmed address onto the current stack frame and execute a little bit of simple code. In the Arduino framework, this is used to quickly read in received data from the UART connection and store it in an internal ring buffer. For the purpose of the simulator, this behavior was emulated with a thread which polled the filestream for new data before storing it into a buffer available to the user's sketch with the Serial instance. This thread also runs once per millisecond and exits when a flag is cleared.

Class Explanations

Many classes are defined and implemented in the Arduino Simulator program. Each one will be explained in brief with a description of the class's role and how it relates to other classes through inheritance.

ARDUINO_SIMULATOR : This class is ultimately responsible for running the user's firmware and accepting the user's API calls normally handled by the Arduino hardware in some fashion. It is a Singleton object which can only be accessed through a static method which constructs the only instance on its first call. It is in charge of two of the program's thread.

SERIAL_TERMINAL : This class takes input from the user and communicates it along to a file. It is in charge of a single main program loop and is a means through which the user can interface with and eventually exit the program.

ARDUINO_PIN_INTF : This is an abstract class intended to define an interface which can be inherited by others as a kind of contract. The ARDUINO_SIMULATOR instance keeps an array of pointers to this class such that it does not need two separate arrays and sets of functions for the ARDUINO_ANALOG_PIN and ARDUINO_DIGITAL_PIN classes.

ARDUINO_ANALOG_PIN : This class inherits and implements the interface defined by ARDUINO_PIN_INTF to allow the user to use the pins marked A0 through A6 on the Arduino UNO as digital GPIO pins or analog inputs.

ARDUINO_DIGITAL_PIN : This class inherits and implements the interface defined by ARDUINO_PIN_INTF to allow the user to use the pins marked D0 through D13 on the Arduino UNO as digital GPIO pins.

USART : This class is the user sketch's interface to the ARDUINO_SIMULATOR's emulated USART peripheral functionality. It implements the same interface as Arduino's Serial class, with the small exception of using template functions to account for various inputs to the

print and *println* functions rather than overloading the functions for numerous types.

LOGGER : This class is a global singleton available as through a *getInstance* call. It allows anywhere in the application code to send a formatted string to the log file in an interface identical to the C Standard Library *printf* function.

Assignment C++ Features

A number of features made available by the C++ programming language are utilized in the program in order to satisfy both good software design practices and the requirements of the project. The features are tabulated below alongside the filename and line where they can be found. This table is not exhaustive. All filenames and line numbers are based on the most recent commit as of this writing in the source's repository on GitHub, with commit hash 55c502a6afbf1178f5b1ddca44e2c5efc2c81f02. The repository can be found here: https://github.com/ssnover95/arduino-simulator.

Language Feature	Filename	Line Number	Class and Function
Pointers	arduino-simulator.hpp	53	ARDUINO_SIMULATOR : getInstance
Classes	arduino-simulator.hpp, logger.hpp, serial-terminal.hpp, USART.hpp	--	--
Overloading	arduino-analog-pin.hpp	56	ARDUINO_ANALOG_PIN : operator>>
Inheritance	arduino-analog-pin.hpp	14	ARDUINO_ANALOG_PIN : --
Virtual Functions	arduino-pin-intf.hpp	37, 44, 52	ARDUINO_PIN_INTF : pinMode, digitalWrite, digitalRead
Standard Template Library	arduino-simulator.hpp	60, 63	std::array

File Processing	logger.cpp	41, 43	LOGGER : printMessage
Lambdas	main.cpp	22-25	– : main
Variable Argument Number	logger.cpp	33	LOGGER : printMessage
Function Templates	USART.hpp	50, 60	USART : print, println

Table 1. Language Features Utilized in Program

Conclusion

The Arduino simulator is in very early stages as of its current state, with many parts of the Arduino API unimplemented and more work to be done in logging events. The granularity granted from running the source code directly however is not enough to correctly simulate all of the edge cases possible in a sketch. There are also frequently instances where the user will hand write firmware which bypasses the Arduino API and makes calls to the hardware directly which currently is unsupported in any form. To achieve a truly flexible simulator, a program would need to be written which is able to parse the machine code emitted by the toolchain's linker itself. This program would have significantly more insight into the user's firmware and could be designed to avoid the issues present in this approach.