Network-Based Remote Game Deployment System on Arduino Using a Raspberry Pi

Sebastián David Pérez Dienstmaier
Dept. of Computer Science and Information Engineering
NTUST
Taipei, Taiwan
sebastianpd2004@gmail.com

Marco Antonio Verón Lezcano
Dept. of Computer Science and Information Engineering
NTUST
Taipei, Taiwan
marco.veron@uptp.edu.py

Sebastián Nicolás Núñez Pavetti
Dept. of Computer Science and Information Engineering
NTUST
Taipei, Taiwan
sebastian.nunez@uptp.edu.py

Carlos Arístides Rubén Gernhofer Olmedo
Dept. of Computer Science and Information Engineering
NTUST
Taipei, Taiwan
carlosa.gernhofer@uptp.edu.py

Project Repository: https://tinyurl.com/pi-game-console

***Abstract*—This paper presents the design and implementation of an Arduino-based game console supported by the Raspberry Pi's storage and networking capabilities. The console is built around an Arduino UNO connected to a 1.3-inch OLED display, along with additional modules for user-input and audiovisual feedback. A Raspberry Pi 4 is used to host a Flask-based web server that allows users to remotely select games from a catalog developed for this system. Once a game is selected, the server sends a signal to the Raspberry Pi and initiates an automated process that flashes the corresponding game onto the Arduino using `AVRDUDE` without requiring any manual interaction from the user. The user can then interact with the game using the input devices connected to the Arduino as well as receive feedback from the output modules. This architecture allows for quick and seamless switching between games and showcases the integration of network functionality to enhance embedded systems.**

***Index Terms*—Arduino, Raspberry Pi, embedded systems, IoT, interactive applications, Flask, web interfaces, local networking.**

I. Introduction

Embedded systems are increasingly integrating network connectivity as part of Internet of Things (IoT) applications. This project focuses on this interaction by creating a hybrid system that combines an Arduino UNO and a Raspberry Pi 4 to emulate a small-scale game console ecosystem. The console allows simple interactive games to be run directly on the Arduino and displayed on a 1.3-inch OLED display. Additionally, the Raspberry Pi works together with the Arduino by storing the game library and hosting a Flask-based web server [1] that provides the user with an interface to choose from those games.

The user may interact with the currently loaded game through a joystick and button module connected to the Arduino. Furthermore, the Arduino provides audiovisual feedback through the 1.3-inch OLED display, a buzzer, and a LED light.

Due to the Arduino UNO's limited memory, it is unfeasible to keep multiple games loaded simultaneously. However, this limitation can be addressed by the Raspberry Pi 4, which can contain and manage multiple game binaries.

A total of four games were developed for this project. These games are specifically designed to use the modules provided by the system, and are intentionally simple due to the hardware limitations of the system. The games are stored in a precompiled .hex format inside the Raspberry Pi. Once a game is selected, the Raspberry Pi flashes the corresponding binary onto the Arduino. This is done through the use of AVRDUDE [2], which is a command-line utility commonly used to program AVR-based microcontrollers.

The remainder of this paper is organized as follows: Section II provides an overview of the system architecture. Section III describes the hardware components used and the enclosure design. Section IV describes the software architecture, which includes the Flask server, the remote game deployment, and the game implementations. Lastly, Section V concludes the paper and mentions possible future improvements.

II. System Overview

The system consists of two primary components: the Arduino, which handles game execution and interactivity, and the Raspberry Pi, which manages storage, networking, and remote file deployment. The Raspberry Pi communicates with the Arduino over a USB serial connection. Additionally, in order for the Raspberry Pi to properly operate, it must remain

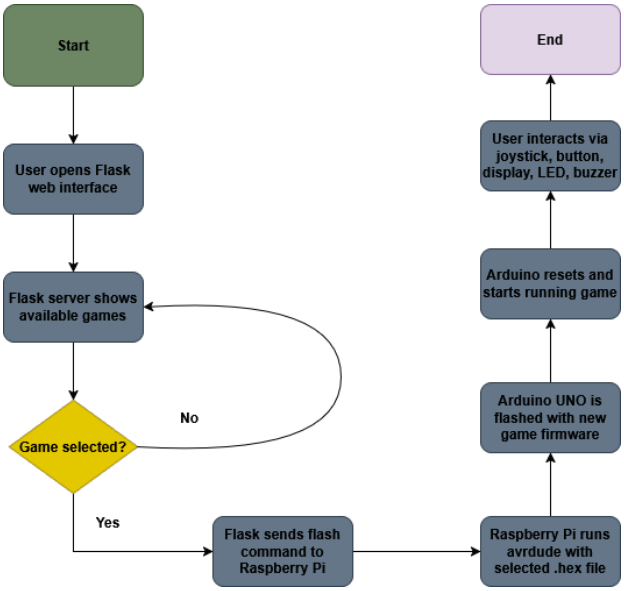


Fig. 1: System flowchart for remote game deployment and execution

powered and connected to the local Wi-Fi network. If the user is in the same network as the Raspberry Pi, it may then communicate with it through the web server it hosts.

The general workflow of this system is shown in Fig. 1. The user starts by opening the Flask web interface. When a game is selected, the Flask server issues a command that triggers the Raspberry Pi to execute `AVRDUDE` with the corresponding .hex file. This causes the Arduino to reset and run the newly flashed firmware. When the game finishes loading, the user can use the modules provided by the system to interact with it. Note that while the game is running, the Raspberry Pi continues to serve its web server functionality, and the user may load another game at any point during its service.

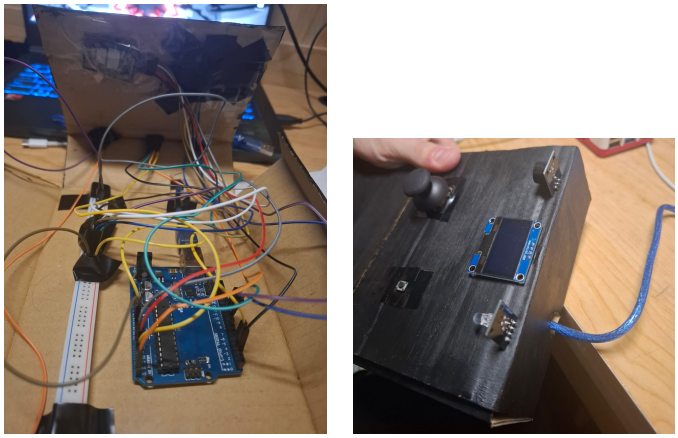
This architecture creates a clear division between application deployment and execution, enabling remote switching of applications through a browser interface without requiring direct physical access to the Arduino.

III. Hardware Design

A. Electronic Components

The system makes use of multiple hardware modules in order to provide the available functionalities. The main components are:

- Arduino UNO microcontroller
- Raspberry Pi 4 Model B
- 1.3-inch I^2C SH1106 OLED display
- Joystick module
- Push-button module
- Passive buzzer
- RGB LED
- Breadboard and jumper wires
- USB cable connecting the Raspberry Pi to the Arduino
- External USB-C power supply for the Raspberry Pi



(a) Internal Wiring (b) Cable hole

Fig. 2: Wiring and cable design

B. Electrical Connections and Assembly

All of the wiring between the electronic modules, Arduino, and breadboard is mounted inside a cardboard enclosure. The OLED display is connected to the Arduino via its I^2C interface. The joystick is connected to two analog input channels corresponding to the X-axis and Y-axis directions, as well as a digital input pin for its switch functionality. The button is connected to a single digital input pin. The passive buzzer is connected to a PWM-capable digital output pin. Lastly, the RGB LED is connected to three PWM-capable digital output pins, one for each color channel. This wiring is summarized in Fig. 9 at the end of the document.

The Arduino UNO and breadboard were also placed inside the enclosure. A small opening was cut in the cardboard in order to allow the USB cable to pass through to the Raspberry Pi, as can be seen in Fig. 2.

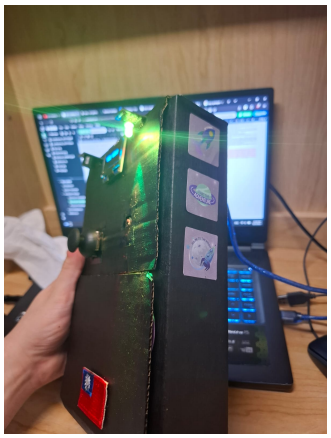
TABLE I: Electrical Connections Between Arduino and Hardware Modules

Module	Arduino Pin	Signal Type
Joystick (X-axis)	A0	Analog input
Joystick (Y-axis)	A1	Analog input
Joystick Button	D2	Digital input
Push Button	D4	Digital input
OLED SDA	A4	I^2C data
OLED SCL	A5	I^2C clock
Passive Buzzer	D3	PWM output
RGB LED (Red)	D9	PWM output
RGB LED (Green)	D10	PWM output
RGB LED (Blue)	D11	PWM output

C. Physical Enclosure

The console enclosure was constructed from layered cardboard panels. Cardboard was chosen due to its availability, low cost, ease of modification, and low weight. Two openings were cut into the top panel to make space for the joystick and button modules, which allows the user to directly and comfortably interact with them. The output modules (RGB LED, OLED display, buzzer) were mounted by inserting their pins through the cardboard and securing them from behind.

(a) Exterior view of the console enclosure



(b) Console Decorations

Fig. 3: Physical enclosure design of the Arduino-based console

All internal components and wiring were reinforced with tape in order to secure them in place. More permanent mounting techniques were purposely avoided since most of the components are borrowed and have to remain in their original condition.

The exterior surface of the enclosure was painted in black to provide a console-like appearance. Some aesthetic elements were also added to improve the external appearance of the enclosure.

The Raspberry Pi was kept outside the enclosure in order to maintain easier access to its power and USB ports and to keep the cardboard console lightweight.

IV. Software Architecture

A. Flask Web Server

The Flask web server application, game binaries, and game thumbnails are organized in the Raspberry Pi in the following structure:

```
/home/sebas/gameconsole
|-- app.py
|-- games/
|    |-- Astral Attack 2.ino.hex
|    |-- Bop-It.ino.hex
|    |-- Snake.ino.hex
|    \-- Survive.ino.hex
\-- static/thumbnails/
     |-- Astral Attack 2.png
     |-- Bop-It.png
     |-- Snake.png
     \-- Survive.png
```

The server is implemented in the `app.py` file and provides the user a front-end interface through which a game may be selected. The server automatically enumerates all files in the `games/` directory with the `.hex` extension. For each game, the server constructs a display name, a thumbnail image if provided, and fills in the HTML interface with each component.

The resulting web interface shows each game as a card containing its respective thumbnail and a button with its name, as can be seen in Fig. 4. Whenever the user presses the button to select a game, it triggers a POST request signal which starts the remote flashing pipeline.

B. Remote Flashing Pipeline

Once a game is selected, the web interface sends a POST request to the `/flash` endpoint of the Flask server, sending within it the filename of the chosen game. The Flask server then executes an `AVRDUDE` command that flashes the Arduino UNO over its USB serial connection with the `.hex` file.

The `AVRDUDE` command performed for flashing is the following:

```
avrdude -v -patmega328p -carduino
    -P/dev/ttyACM0 -b115200 -D
    -U flash:w:<game>.hex:i
```

This command places the Arduino into bootloader mode, erases the currently loaded firmware, and writes the new game binary into the Arduino. Once the flashing is complete, the Arduino resets and starts to execute the selected game.

The server also returns the terminal output from `AVRDUDE`, which is saved and displayed to the user in the web interface. This is useful to indicate when the flashing is completed, if there were any issues during the execution, and for additional information with regards to the flashing process (e.g., time taken to flash).

C. Arduino Runtime and Game Implementation

All of the games were first developed as standard Arduino sketches in the Arduino IDE, where files are in `.ino` format. The SH110X and GFX libraries were used to interact with the OLED display [3], [4]. Since the Raspberry Pi cannot compile Arduino sketches directly, the games had to be exported as precompiled `.hex` binaries. The Arduino IDE provides this export functionality within the Sketch menu. These binaries contain the machine code that the Arduino UNO requires for file execution, and are in a format accepted by `AVRDUDE`.

After a game is flashed onto the Arduino, all game-related functionalities are handled locally on the Arduino. The Raspberry Pi remains idle until the user decides to choose another game to play.

The four games implemented for this system are the following:

Astral Attack 2 is a shooting-style game in which the player controls a small triangular spaceship in space. The player can move vertically across three horizontal lines using the joystick, and bullets can be fired from the spaceship by pressing the button module, which also causes the buzzer to make a sound effect. Obstacles in the form of squares appear on the right side of the screen and start to move towards the left, where the player is located. Note that the obstacles are also limited to the three lines the player has access to. Lastly, pixels are

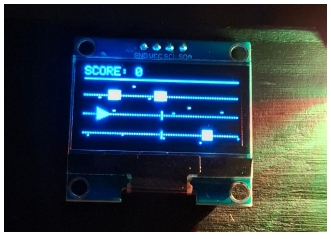
Fig. 4: Flask-based web interface for selecting and flashing games

drawn in the background and slowly drift to the left of the screen as to simulate movement and stars in space.

The player loses health if they are hit by an obstacle or they let an obstacle pass through to the left side of the screen. The health of the player is indicated by the RGB LED module, with green indicating full health, yellow indicating medium health, and red indicating low health. The player may destroy the obstacles and gain score by striking them with bullets.

The game gets progressively harder based on the current score of the player. As score increases, both the enemy spawn rate and enemy speed increase as well. The objective of the game is to achieve the highest score possible, and the player loses when they have no more health remaining.

(a) Astral Attack 2 title screen



(b) Astral Attack 2 gameplay

Fig. 5: Screenshots of *Astral Attack 2* running on the console

Snake is a recreation of the classic snake game on the 1.3-inch OLED display. The 128x64 pixel screen of the OLED is divided into a 16x8 grid of 8-pixel cells. The snake's head is represented as a filled square, while its body is represented as unfilled squares. The player can control the snake's direction with the joystick, and food items are placed on random unoccupied grid cells.

Each time the player comes into contact with a food item, a single unfilled square is added onto the snake's tail and the buzzer provides a brief audio feedback. A score is also kept, with each food item collected increasing it by one. This is also reflected by the RGB LED, which first starts as a dim green light and slowly increases in brightness as the score increases.

The goal of the game is to get the highest score possible. The game gets progressively harder due to the increasing tail length, and ends whenever the player hits the border of the screen or their own tail.

(a) Snake title screen

(b) Snake gameplay

Fig. 6: Screenshots of *Snake* running on the console

Bop-It is a video game adaptation of the real-life toy known as Bop It. Like the toy version, it gives the player simple and time-sensitive commands that they must quickly react to in order to keep playing.

The joystick and button are used the player input tools for the commands. There are six possible instructions that the

player may receive: Joystick up, joystick down, joystick right, joystick left, joystick switch, and button push. The buzzer provides audio feedback each time an action is successfully performed.

Two gameplay modes were implemented for this game: single-player and multi-player.

In single-player, only one user is given commands to perform. There are no pauses between instructions, and the time given to react is slowly decreased after each successful action. The RGB LED reflects the difficulty by transitioning from green to red as it increases. The game ends when the player is unable to perform the given command under the provided time, and the goal is to last as long as possible.

In multi-player, up to four users can be given commands to perform. The game rotates between players and gives each player a sequence of instructions to perform. A slight pause is given between turns to give each user time to prepare themselves. In this case, the RGB LED is used to assign a color to each player and indicate the current player's turn. The game difficulty also increases in multi-player, but at a slower rate than in single-player. The game ends when only one player is remaining, and that player is declared the winner.

(a) Bop-It title screen

(b) Bop-It player selection screen

(c) Bop-It gameplay.

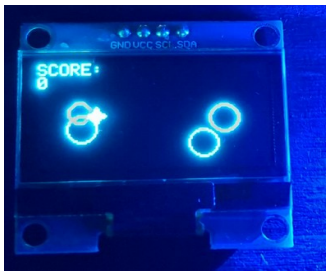
Fig. 7: Screenshots of *Bop It* running on the console

Survive is an avoidance game in which the player controls a star-shaped figure and has to dodge incoming obstacles by moving out of their way. The star can be moved in all directions using the joystick module, and the button module can be pressed to activate an ability that temporarily gives the player invincibility for a brief time. However, this ability can only be used once every two seconds.

The RGB LED is used to indicate whether the ability is ready to be used, with green indicating ready and blue indicating unavailable. The buzzer module also makes a sound effect whenever the player pushes the button and successfully activates the ability.

Score increases for each second the player remains alive. The game ends the moment the player is hit by an obstacle, and the goal is to get the highest score possible.

(a) Survive title screen



(b) Survive gameplay

Fig. 8: Screenshots of *Survive* running on the console

V. Conclusion

This project has demonstrated the integration of embedded hardware and networking software to create a working game console ecosystem. The system works by combining the Arduino UNO and Raspberry Pi 4, and assigning different roles to them. The Raspberry Pi handles the storage and networking, while the Arduino executes the game logic and is connected to the I/O interface modules. A total of four games were developed to showcase the capabilities of the console.

Future improvements could be adding more games, improving the enclosure, and expanding the architecture to support even more sensors.

References

[1] Flask Documentation, Pallets Projects. [Online]. Available: https://flask.palletsprojects.com/.

[2] AVRDUDE: AVR Downloader/UploaDEr, GitHub repository. [Online]. Available: https://github.com/AVRDUDEs/AVRDUDE.

[3] Adafruit SH110X Arduino Library, Adafruit Industries. [Online]. Available: https://github.com/adafruit/Adafruit_SH110X.

[4] Adafruit GFX Graphics Library, Adafruit Industries. [Online]. Available: https://github.com/adafruit/Adafruit-GFX-Library.

[5] Fritzing: Open-source Electronics Design Tool. [Online]. Available: https://fritzing.org/.

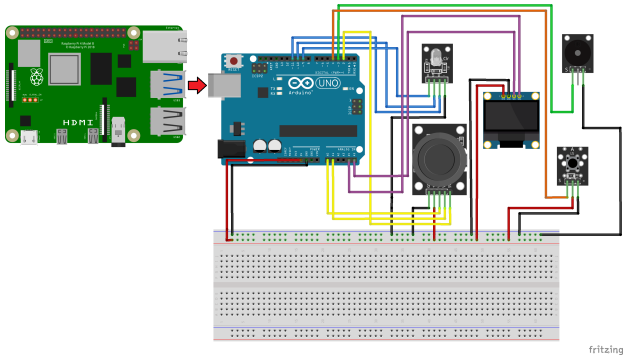


Fig. 9: Wiring diagram made with Fritzing [5]