SAPIENZA
UNIVERSITÀ DI ROMA

# Custom Wireless Joystick Development with Arduino and Linux Integration


Facoltà di Ingegneria dell'Informazione, Informatica e Statistica
Dipartimento di Ingegneria Informatica, Automatica e Gestionale
Corso di Laurea in Ingegneria Informatica e Automatica

Candidate
Federico Gerardi
ID number 1982783

Thesis Advisor
Prof. Giorgio Grisetti


Academic Year 2023/2024

**Custom Wireless Joystick Development with Arduino and Linux Integration**
Bachelor's thesis. Sapienza – University of Rome



This thesis has been typeset by LaTeX and the Sapthesis class.

Author's email: federico.gerardi03@gmail.com

*A mio padre, Riccardo Gerardi*

# Contents

# Chapter 1

# Introduction

The goal of this thesis is to explain how to create a complex Arduino project that can communicate with other devices through serial communication and radio. I am going to explain how to:

- Read analogic and digital data from external modules
- Communicate via radio
- Serial communication
- Input and Output device manipulation in Linux
- Organize a project using a version control system

My contribution can be useful for people who want to start their journey in embedded programming and operating systems.

Approaching a difficult topic like embedded programming could be very complicated. This thesis introduces a step-by-step guide to approach this world.

The final project is a wireless joystick controller that can be used on Linux PCs to play video games, but this project can have many applications in other situations.

For example, joysticks for drones are expensive, reading this report, people can build one by themselves easily, saving money and learning something interesting.

It can also be used to control robots, internet of things devices, smart home devices, and many more...

This thesis gives a background also on Linux management of input and output devices I'm going to explain the creation of virtual devices, and how to control them.

If you want to create automatic tests for your applications, this part is very useful for you. It can also be interesting for people who work with AI to play video games automatically.

We are going to create a video game that uses the Linux Joystick Library. This part is very useful for game developers.

# Chapter 2

# Related works and bases

In this chapter, I'm going to explain a few concepts that are important to know before we start the project. I will explain basic concepts like:

- AVR microcontrollers and how to program them
- What is an HC-12 module
- What is make, and why it is so helpful
- What is the ioctl system call
- What is git

## 2.1 AVR

### 2.1.1 Introduction to AVR

AVR is a system that contains a CPU, RAM, peripherals and flash memory to store a program, in other words, a microcontroller. Microcontrollers are very useful for embedded applications and can be used in a lot of circumstances, like Internet of Things, drones, etc.

More precisely We are going to use the AVR-2560 microcontroller that has the following schema:

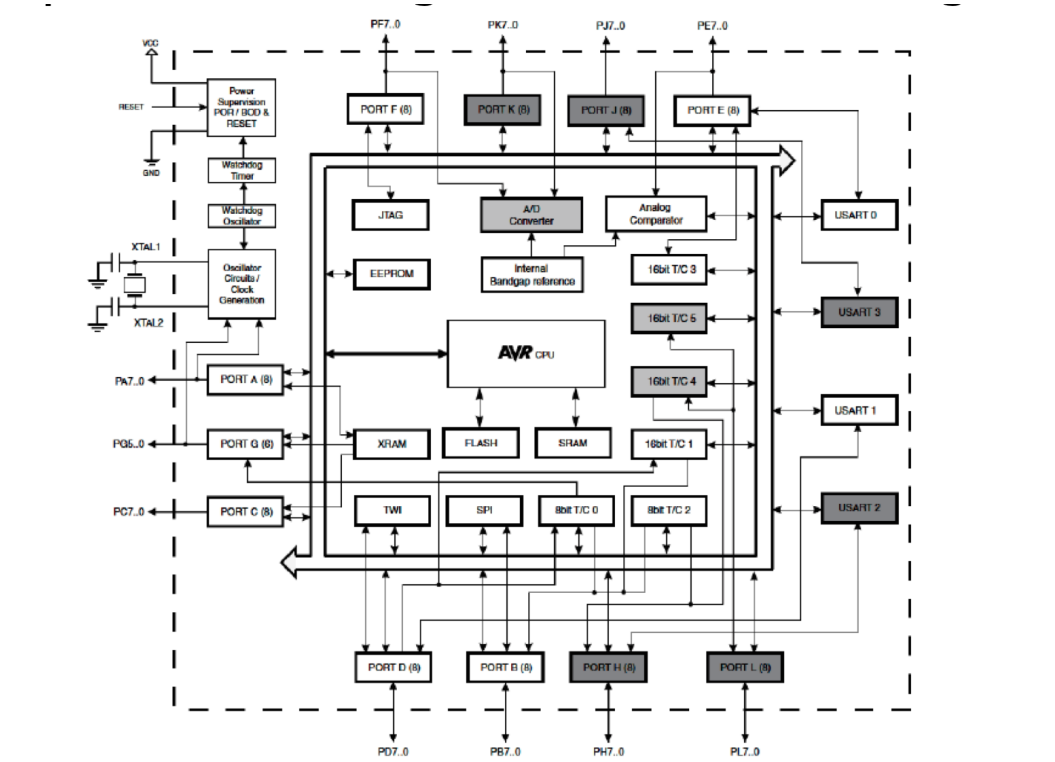


**Figure 2.1.** AVR-2560 Schema

To store a program in a microcontroller we need a programmer or a bootloader. To avoid the use of a programmer, We will use an Arduino Mega which is a board that contains an AVR-2560 and another microcontroller that works as a bootloader.

We will also use an Arduino Uno that is very similar to the Arduino Mega but contains an AVR-328p

Arduino is an open-source family of boards that contains everything we need to work with AVR microcontrollers like bootloaders and USB-to-serial converters.

### 2.1.2 UART

UART stands for Universal Asynchronous Receiver-Transmitter. It's one of the AVR peripherals for serial ports. The UART serial port is controlled by:

- UDR: the data register that contains the data being sent and received
- UCSR: the status register
- UBRR: the register that tells how fast the transmission should be

### 2.1.3 avrdude

Users can download and upload data on the on-chip memories of AVR microcontrollers using the Avrdude program. It can be used to program the Flash, EEPROM and where supported by the programmer, lock bits, fuses that hold the microcontroller's configuration and other memories that the part might have.

Avrdude is a command line tool that can be used as follows:

```
avrdude -p partno options ...
```

Command line options are used to control Avrdude's behaviour. Here you can find a list of the most useful options:

- -p partno
  This option tells avrdude what part (MCU) is connected to the programmer. For example, if you are using Arduino Mega the partno parameter will be m2560. Otherwise, if you are using Arduino Uno the partno parameter will be m328p
- -P port
  This option tells avrdude the connection through which the programmer is attached. This can be a parallel, serial, spi or linuxgpio connection.
- -b baud rate
  This option overrides the RS-232 connection baud rate specified in the respective programmer's baud rate entry of the configuration file or defined by the default_baudrate entry in your configuration file.
- -c programmer-id
  This option specifies the programmer to be used. For example, if you are using Arduino Mega the programmer will be wiring. Otherwise, if you are using Arduino Uno the programmer will be arduino.
- -C config-file
  This option uses the specified config file for configuration data. This file contains all programmer and part definitions that avrdude knows about.
- -U memory:op:filename[:format]
  This option performs a memory operation when it is its turn. The memory field specifies the memory type to operate on. The memory field can also be a comma-separated list of memories, eg, flash, eeprom.
- -D
  This option disables auto-erase for flash. When the -U option for writing to any flash memory is specified, avrdude will perform a chip erase before starting

any of the programming operations, since it generally is a mistake to program the flash without performing an erase first. This option disables that. Setting -D implies -A.

- -A
  This option disables the automatic removal of trailing 0xFF sequences in file input that is programmed to flash and in AVR reads from flash memory. This option should be used when the programmer hardware, or bootloader software for that matter, does not carry out chip erase and instead handles the memory erase on a page level. Arduino bootloader exhibits this behaviour for this reason -A is engaged by default.

### 2.1.4 avr-objcopy

avr-objcopy is a program that copies the contents of an object file to another. It uses the GNU BFD Library to read and write the object files. It can write the destination object file in a format different from that of the source object file.

avr-objcopy is a command line tool that can be used as follows:

```
avr-objcopy infile [outfile] options ...
```

Command line options are used to control Avrdude's behaviour. Here you can find a list of the most useful options:

- infile
  outfile
  The input and output files, respectively. If you do not specify outfile, objcopy creates a temporary file, named the same as the infile.

- -O bfdname
  Write the outfile using the object format bfdname

- -R sectionname
  This option is useful to remove sections which contain information that is not needed by the binary file

## 2.2 HC-12

### 2.2.1 Introduction to HC-12

HC-12 wireless serial port communication module supports long-distance wireless transmission, up to 1 Km in open space and 5000 bps baud rate in the air. Its working frequency range is between 433.4 MHz and 473.0 MHz with up to 100 communication channels. With a maximum of 100 mW (20 dBm) transmitting power and three working modes (to adapt it to different situations), It's one of the best new-generation modules for wireless communication.

The MCU inside the module is what makes this module the easiest to use, the user doesn't need to program the module separately.

The following schema compares the difference between a serial physical connection and an HC-12 module connection. As you can see is very easy to use.

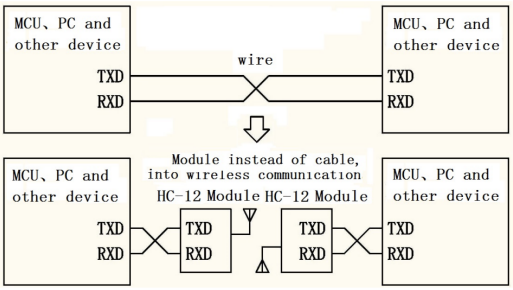


**Figure 2.2.** Comparation between physical connection and HC-12 module

### 2.2.2 Definition of pins

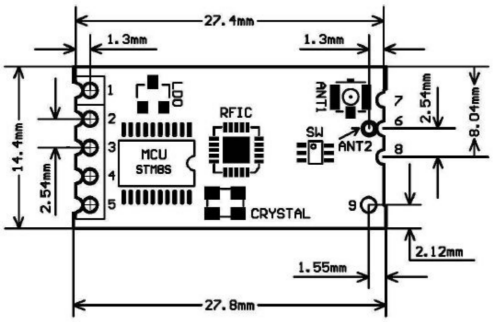


**Figure 2.3.** HC-12 module

1. VCC, between 3.2V and 5.5V not less than 200 mA
2. GND
3. RXD, UART input port
4. TXD, UART output port

## 2.3 Make

### 2.3.1 Introduction to make and makefiles

The make utility automatically determines which pieces of a large program need to be recompiled, and issue commands to recompile them.

Suppose you have a project with hundreds of c files and thousands of lines of code for each. Every time you modify a file and want to compile the project, you have to compile every file. If you have modified only one file, why should you re-compile all the files? This is a waste of energy and time.

Make solves this problem, by compiling only what you have modified.

Make needs a special file called makefile to know what to do. Usually, the makefile tells make how to compile and link a program.

Here you can see an example of a Makefile:

```
# Variables
CC = gcc
CFLAGS = -Wall -Wextra -O2
SRC = test.c
TARGET = test

# Rule: Dependencies
all: $(TARGET)

$(TARGET): $(SRC)
        $(CC) $(CFLAGS) -o $(TARGET) $(SRC)

# Rule that deletes all the compiled files
clean:
        rm -f $(TARGET)

# Rule that clean and build
rebuild: clean all

.PHONY: all clean rebuild
```

In the first section, there are the variables. Then you can see the first rule called "all" that compiles all the modified files. I have also defined a "clean" rule to delete all the compiled files and a "rebuild" one that cleans and compiles all the files.

### 2.3.2 Arduino and makefile

Make can call every bash command without any problem. So we can create a special makefile that uses avr-gcc instead of gcc and call avr-objcopy and avrdude to convert the elf file and upload it to Arduino.

```
INCLUDE_DIRS=-I. -Iavr_common
CXX=avr-g++
CC=avr-gcc
AS=avr-gcc
AVRDUDE=avrdude


CC_OPTS_GLOBAL=\
-O3\
-funsigned-char\
```

```
-funsigned-bitfields\
-fshort-enums\
-Wall\
$(INCLUDE_DIRS)\
-DF_CPU=16000000UL\

TARGET=uno
AVRDUDE_PORT=/dev/ttyACM0

ifeq ($(TARGET), mega)
        CC_OPTS_GLOBAL += -mmcu=atmega2560 -D__AVR_3_BYTE_PC__
        AVRDUDE_FLAGS  += -p m2560
        AVRDUDE_BAUDRATE = 115200
        AVRDUDE_BOOTLOADER = wiring
endif

ifeq ($(TARGET), uno)
        CC_OPTS_GLOBAL += -mmcu=atmega328p
        AVRDUDE_FLAGS  += -p m328p
        AVRDUDE_BAUDRATE = 115200
        AVRDUDE_BOOTLOADER = arduino
endif

CC_OPTS=$(CC_OPTS_GLOBAL) --std=gnu99
CXX_OPTS=$(CC_OPTS_GLOBAL) --std=c++17
AS_OPTS=-x assembler-with-cpp $(CC_OPTS)

AVRDUDE_WRITE_FLASH = -U flash:w:$(TARGET):i
AVRDUDE_FLAGS += -P $(AVRDUDE_PORT) -b $(AVRDUDE_BAUDRATE)
AVRDUDE_FLAGS += -D -q -V -C /usr/share/arduino/hardware/tools/avr
    /../avrdude.conf
AVRDUDE_FLAGS += -c $(AVRDUDE_BOOTLOADER)

.phony: clean all

all:    $(BINS)

%.o:    %.c
        $(CC) $(CC_OPTS) -c  -o $@ $<

%.o:    %.s
        $(AS) $(AS_OPTS) -c  -o $@ $<

%.elf:  %.o $(OBJS)
        $(CC) $(CC_OPTS) -o $@ $< $(OBJS) $(LIBS)


%.hex:  %.elf
        avr-objcopy -O ihex -R .eeprom $< $@
        $(AVRDUDE) $(AVRDUDE_FLAGS) -U flash:w:$@:i

clean:
        rm -rf $(OBJS) $(BINS) *.hex *~ *.o

.SECONDARY:     $(OBJS)
```

As you can see we set variables in different ways based on which Arduino board we are using. Also, we define a new rule called %.hex. Given an elf file, this rule will:

1. Call avr-objcopy to convert the elf file to a hex file
2. Call avrdude to upload the hex file to the Arduino through the serial port

## 2.4 ioctl

The ioctl system call manipulates the underlying device parameters of special files.

ioctl call takes as parameters:

1. an open file descriptor
2. a request code number
3. an untyped pointer to data

The kernel sends an ioctl call straight to the device driver, which reads the request number and data in the right way.

ioctl can also be used to create virtual devices. You can use the open function to open the /dev/uinput device and call the ioctl UI_DEV_CREATE operation. These operations will respectively create a file descriptor for a virtual device and the new virtual device.

The virtual device needs a few ioctl calls (especially UI_SETUP_*) to allow Linux to know the type of device created. For example, if you want to create a virtual joystick you need to set:

- UI_SETUP_EVBIT to EV_ABS, so Linux will know that the new device will have events that are absolute and not relative
- UI_SETUP_ABSBIT to ABS_X, so Linux will know that the new device will have events for the x-axis absolute
- UI_SETUP_ABSBIT to ABS_Y, so Linux will know that the new device will have events for the y-axis absolute
- UI_SETUP_EVBIT to EV_KEY, so Linux will know that the new device will have binary events
- UI_SETUP_KEYBIT to BTN_JOYSTICK, so Linux will know that the new device will have a joystick button

With these operations, Linux will automatically understand that the device is a joystick controller.

## 2.5 Git

Git is the most used software for version control systems in the world. Developed by the Linux developers community to carry on the Linux kernel open source project.

Git Goals are:

- Speed
- Data integrity
- Support for distributed, non-linear workflows running on different computers
- Capable of supporting very large projects

It's compatible with existing systems and protocols like Hypertext Transfer Protocol Secure (HTTPS), Hypertext Transfer Protocol (HTTP), File Transfer Protocol (FTP) and Secure Shell (SSH).

In Git the user clones the repository from a server. The clone operation is basically a download of the repository. After that operation, the user will have a local copy of the repository.

The user can do a few operations with the local copy like:

- Update his local copy with the new versions from the server
- Edit some files and commit the changes

It's important to know that the local copy keeps the history of every commit. This allows the user to roll back to an older commit if something is not working.

When you are ready you can push the changes to the server and synchronize the repositories.

Git uses branches to allow people to work together without any issues that can occur when a few or two people are editing the same file.

When a user needs to start a new task he can create a branch from the main one. The new branch will be identical to the main one, but only one user will work on it. When the user ends his job with that task, he can merge his branch to the main one. If the files the user has edited, meanwhile have not changed in the main branch, everything is fine, the user can merge without any problem. But if the files the user has edited, meanwhile have changed in the main branch, there is a conflict and git helps the user to solve the conflict and merge the branch.

There are a lot of different git providers, the most famous are:

- Github
- Gitlab
- Bitbucket

# Chapter 3

# My Contribution

In this chapter, we are going to start the project. I am going to explain how to:

- Connect the components physically
- Create the firmware for both the transmitter and receiver
- Create the driver (virtual joystick)
- Create a videogame that works with the Linux controller library
- Work with repositories

## 3.1 Hardware

For this project, I will use:

- Arduino Mega
- Arduino Uno
- 2x HC-12 modules
- Joystick module
- Jumpwires

I have connected them as you can see in the following schema:

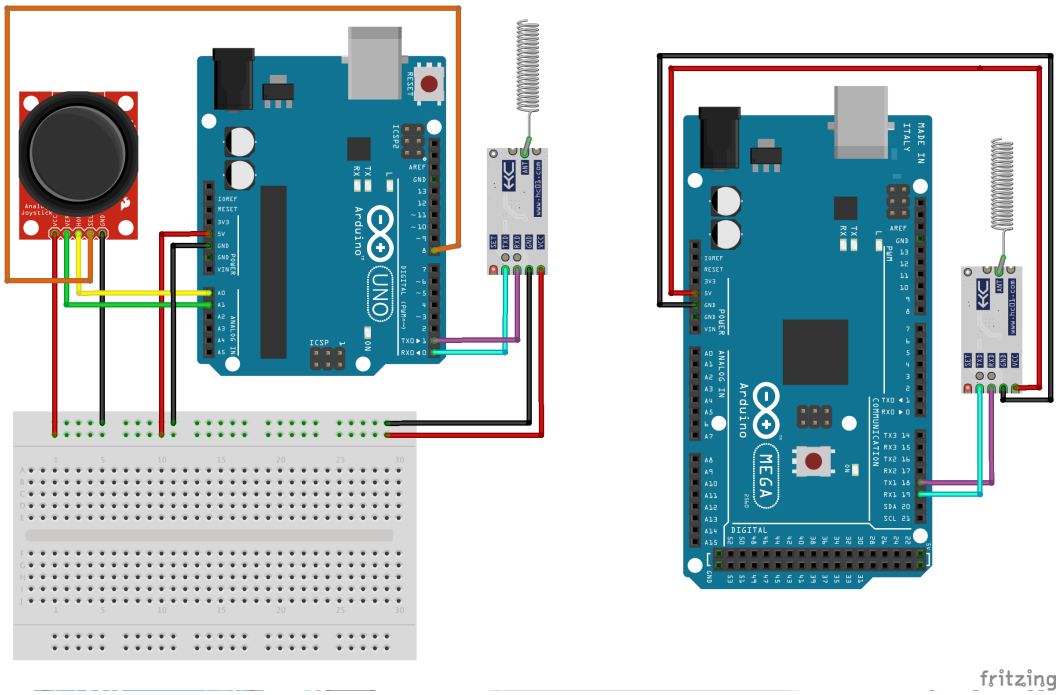


**Figure 3.1.** Physical Schema

I have connected to the Arduino Uno the joystick and the HC-12 module. So Arduino Uno will be the transmitter. I have taken this decision because Arduino Uno has only one Serial UART, while Arduino Mega has 3 Serial UARTs. Using Arduino Uno to receive data and retransmit on the same UART could cause conflict problems.

So I have connected the HC-12 module to the UART0 (the same as the USB port) to Arduino Uno (from now we will call him transmitter) and I have connected the other HC-12 module to the UART1 to Arduino Mega (from now on we will call him receiver).

The joystick is connected to the transmitter's 5 V and GND. The X and Y axis values are connected to A0 and A1 respectively (2 generic analogic pins are ok). The joystick button is connected to the digital pin 8 (1 generic digital pin is ok).

## 3.2 Transmitter

The transmitter's role is:

1. Read the data from the joystick
2. Send the joystick data to the UART0 (The HC-12 module will transmit the data)

### 3.2.1 Read the data from the joystick

To read analogic data we are going to use an Analog-Digital converter, included on AVR microcontrollers. This is because AVR microcontrollers are digital devices so they can understand only binary and work with discrete voltage levels. We need to do a few operations to use the ADC so we define the adc_init function to:

1. Set the ADC prescaler to divide the system clock by 128. Important because the ADC requires a clock frequency slower than the system clock to get accurate readings.
2. Then we set the voltage reference to AVcc (5 V). This guarantees that the ADC readings are scaled to the AVcc voltage.
3. Finally we can activate the ADC setting the ADEN bit in the ADC Control and Status Register A.

Here is the equivalent C code:

```
void adc_init() {
  // Setting ADC Prescaler to 128
  ADCSRA |= (1 << ADPS2) | (1 << ADPS1) | (1 << ADPS0);

  // Set voltage to AVcc with an extern capacitor on AREF pin
  ADMUX |= (1 << REFS0);

  // Activate ADC
  ADCSRA |= (1 << ADEN);
}
```

Reading digital data is way easier because we don't need to do any conversion. We only have to set the pin as input and activate the pull-up resistor.

Here is the equivalent C code for the PIN 8:

```
void digital_init() {
  // Setting port as input
  DDRB &= ~(1 << 0);

  // Activating pull up resistor
  PORTB |= (1 << 0);
}
```

Now that everything has been set up we can read data.

I have created a function that allows reading analog data from a generic channel, using the ADC. The function chooses the channel, starts the conversion, waits until the conversion is completed and then returns the value.

```
uint16_t adc_read(uint8_t ch) {
  // Select ADC channel
  ch &= 0b00000111;
  ADMUX = (ADMUX & 0xF8) | ch;

  // Start conversion
  ADCSRA |= (1 << ADSC);

  // Wait for the conversion to complete
  while (ADCSRA & (1 << ADSC));

  return (ADC);
}
```

I have also created a function to read digital data, but it is basically an if that checks if the bit of the pin 8 is 1 or 0.

```
uint8_t digital_read() {
  return (PINB&(1 << 0))==0;
}
```

### 3.2.2 Send the joystick data to the UART0

To send data through the UART0, we need to set it up.

```
void UART_init(uint16_t baudrate) {
  uint16_t ubrr_value = (F_CPU / 16 / baudrate) - 1;

  UBRR0H = (uint8_t)(ubrr_value >> 8);
  UBRR0L = (uint8_t)ubrr_value;

  UCSR0B = (1 << RXEN0) | (1 << TXEN0);
  UCSR0C = (1 << UCSZ01) | (1 << UCSZ00);

  // Activating UART interrupts
  UCSR0B |= (1 << UDRIE0);
}
```

This function:

1. Calculates the UBRR value (UART Baud Rate Register) using the F_CPU value (CPU Frequency) and the baud rate parameter.
2. Insert the UBRR value in the two separated registers UBRR0H (High) and UBRR0L (Low).
3. Activate the receiver and transmitter.
4. Set the data frame to 8 bits and 1 stop bit.
5. Enable the UART Data Register Empty Interrupt.

The UART0 will be initialized with 9600 baud as baud rate. This is the recommended baud rate by the HC-12 module's datasheet.

We need to decide the format to send data through the UART0. I have designed this structure that fits perfectly for this job.

```
typedef struct __attribute__((packed)) {
  uint16_t x_axis;
  uint16_t y_axis;
  uint8_t button;
} joystick_event;
```

The attribute packed ensures that the structure is packed without any padding between its members.

So we have initialized the UART, we have decided on the data format, and now we can finally send the data.

```
void UART_transmit_struct(joystick_event* ev) {
  uint8_t* byte = (uint8_t*) ev;
  uint8_t size = sizeof(joystick_event);

  for (uint8_t i = 0; i < size; i++) {
    uint8_t next = (tx_index + 1) % TX_BUFFER_SIZE;
    while (next == tx_read_index);  // Spinlock buffer full
    tx_buffer[tx_index] = byte[i];
    tx_index = next;
  }

  // Activating Transmission Interrupt
  UCSR0B |= (1 << UDRIE0);
}


ISR(USART_UDRE_vect) {
  if (tx_read_index != tx_index) {
    UDR0 = tx_buffer[tx_read_index];
    tx_read_index = (tx_read_index + 1) % TX_BUFFER_SIZE;
  } else {
    UDR0 = '\n';
    UCSR0B &= ~(1 << UDRIE0);  // Send \n and stop transmission if
        the buffer is empty
  }
}
```

The first function sends the joystick_event struct through the UART. We send byte-by-byte the whole struct, we use a TX buffer to avoid problems due to serial saturation, and we trigger the transmission interrupt.

When we activate the transmission interrupt, the ISR (Interrupt Service Routine) starts. When the ISR starts, if there is data in the buffer, data is written in UDR0 (serial port 0). Otherwise, we write the new line character as a reminder for the end of the struct.

## 3.3 Receiver

The receiver role is:

1. Read the data from the UART1
2. Send the data through the UART0

### 3.3.1 Read the data from the UART1

To read data from UART1 we need to configure it first. The initialization function is the same as the transmitter, but we don't initialize only the UART0, but also the UART1. (It's the same function but we will point to the registers of the UART1, so UBRR1 instead of UBRR0).

Now we can read the data with the following function:

```
unsigned char uart1_receive(void) {
    while (!(UCSR1A & (1 << RXC1)));
    return UDR1;
}
```

That waits for a byte in UART1 and returns it.

### 3.3.2 Send the data through the UART0

The UART0 is already initialized in 3.3.1, so we can already write with the following function:

```
void uart0_transmit(unsigned char data) {
    while (!(UCSR0A & (1 << UDRE0)));
    UDR0 = data;
}
```

We wait until the transmission buffer is empty and we send data to UART0.

### 3.3.3 Receiver Main

```
int main(void) {
    uart1_init();
    uart0_init();

    while (1) {
        unsigned char data = uart1_receive();
        uart0_transmit(data);
    }

    return 0;
}
```

So we get the data from UART1 and forward it to the UART0.

## 3.4 Driver

The driver's role is:

1. Create the virtual joystick
2. Read the data from the Serial
3. Send the data to the virtual joystick

### 3.4.1 Create the virtual joystick

To create a virtual joystick we need to open a new file descriptor that points to /dev/uinput.

/dev/uinput is a special device file that allows us to emulate input devices if we write on it.

Now that we have the file descriptor, we can define a new function to initialize the new virtual joystick.

```
void joystick_init(int fd, struct uinput_setup* usetup) {
    int ret;

    // Event ABS
    ret = ioctl(fd, UI_SET_EVBIT, EV_ABS);
    if (ret == -1) handle_error("Error ioctl EV_ABS");

    ret = ioctl(fd, UI_SET_ABSBIT, ABS_X);
    if (ret == -1) handle_error("Error ioctl ABS_X");

    ret = ioctl(fd, UI_SET_ABSBIT, ABS_Y);
    if (ret == -1) handle_error("Error ioctl ABS_Y");

    // Event button
    ret = ioctl(fd, UI_SET_EVBIT, EV_KEY);
    if (ret == -1) handle_error("Error ioctl EV_KEY");

    ret = ioctl(fd, UI_SET_KEYBIT, BTN_JOYSTICK);
    if (ret == -1) handle_error("Error ioctl BTN_JOYSTICK");

    // Configuration
    memset(usetup, 0, sizeof(struct uinput_setup));
    usetup->id.bustype = BUS_USB;
    usetup->id.vendor = 0x0198;
    usetup->id.product = 0x2783;
    strcpy(usetup->name, "Gieristick");

    ret = ioctl(fd, UI_DEV_SETUP, usetup);
    if (ret == -1) handle_error("Error ioctl DEV_SETUP");


    // Virtual joystick creation
    ret = ioctl(fd, UI_DEV_CREATE);
    if (ret == -1) handle_error("Error ioctl DEV_CREATE");
}
```

In this function, we define the events (Absolute and Button) in the same way as I explained in 2.6. Then we define the device info, like vendor, product, bus type and name; and finally, we create the device.

### 3.4.2 Read the data from the Serial

To read the data from the Serial, we need to open and configure it first.

We can open the serial using the open function. The open function will return the file descriptor of that serial device. Then we can use the termios library and set attributes like baudrate with the tcsetattr function to a specific file descriptor.

Now that we have opened the serial, and it's configured correctly, we can start to read data.

```
while(1) {
    uint8_t buf[BUFFER_SIZE];

    int bytes_read = 0;
    do {
        ret = read(serial_fd, buf+bytes_read, 1);
        if (ret == -1) {
            if (errno == EINTR) continue;
            else handle_error("Error reading serial");
        }
    } while(buf[bytes_read++] != '\n');

    joystick_event_t ev = *((joystick_event_t*) buf);
    printf("x: %d\ty: %d\tbut: %d\n", ev.x_axis, ev.y_axis, ev.
        button);
    joystick_event(joystick_fd, ev.x_axis, ev.y_axis, ev.button);
}
```

As you can see, we create a buffer, we read byte-by-byte everything from the serial until we get the newline character.

When we get the newline character, it means that the joystick has sent the whole struct, so we can cast the buffer to a joystick_event_t struct.

We print everything for debugging and then send data to the virtual joystick through the joystick_event function.

### 3.4.3 Send the data to the virtual joystick

Now we analyse the joystick_event function.

```
void joystick_event(int fd, int x, int y, int btn) {
    struct input_event event;
    int ret;
    memset(&event, 0, sizeof(struct input_event));
    event.type = EV_ABS;

    event.code = ABS_X;
    event.value = x;
    ret = write(fd, &event, sizeof(struct input_event));
    if (ret == -1) handle_error("Error write X");

    event.code = ABS_Y;
    event.value = y;
    ret = write(fd, &event, sizeof(struct input_event));
    if (ret == -1) handle_error("Error write Y");

    event.type = EV_KEY;
    event.code = BTN_JOYSTICK;
    event.value = btn;
    ret = write(fd, &event, sizeof(struct input_event));
    if (ret == -1) handle_error("Error write button");
}
```

We define an input_event struct (from stdlib.h), we allocate the memory for that struct, and then we write an event for the x-axis, an event for the y axis and an event for the joystick on the virtual joystick's file descriptor.

## 3.5 Test Game

I have also created a test game that works with the official Linux Joystick Library.

The game is really simple, there is a matrix that represents the field of the game. We can move a star with the joystick inside the field of the game.

We define a struct called player position that saves the position of the star.

```
typedef struct {
    int x;
    int y;
} player_position;
```

We also have to define a matrix of chars for the field. This matrix needs to be initialized

```
int matrix_init() {
    matrix = malloc(MATRIX_SIZE*sizeof(char*));

    pp = malloc(sizeof(player_position));

    for (int i=0; i<MATRIX_SIZE; i++) {
        matrix[i] = malloc(MATRIX_SIZE*sizeof(char));
    }

    for (int i=0; i<MATRIX_SIZE; i++) {
        for (int j=0; j<MATRIX_SIZE; j++) {
            if (i == 0 && j == 0) matrix[0][0] = PLAYER_CHAR;
            else matrix[i][j] = EMPTY_CHAR;
        }
    }

    pp->x = 0;
    pp->y = 0;

    return 0;
}
```

The matrix needs to be printed and updated when the user moves the joystick.

```
void matrix_print() {
    for (int i=0; i<MATRIX_SIZE; i++) {
        for (int j=0; j<MATRIX_SIZE; j++) {
            printf("%c", matrix[i][j]);
        }
        printf("\n");
    }

    printf("X: %d\tY: %d\n", pp->x, pp->y);
}

void matrix_update() {
    for (int i=0; i<MATRIX_SIZE; i++) {
        for (int j=0; j<MATRIX_SIZE; j++) {
            if (i == pp->x && j == pp->y) matrix[i][j] =
                PLAYER_CHAR;
            else matrix[i][j] = EMPTY_CHAR;
        }
    }
}
```

Using the js_event struct, and reading from the virtual device we have created with the driver we can update the matrix with the new position.

```
int fd;
int ret;
struct js_event js;

fd = open("/dev/input/js1", O_RDONLY);
if (fd == -1) {
    perror("Error connecting joystick");
    return -1;
}

matrix_init();
system("clear");
matrix_print();

while (1) {
    ret = read(fd, &js, sizeof(struct js_event));
    if (ret != sizeof(struct js_event)) {
        perror("Error reading joystick");
        return -1;
    }

    if (js.type == JS_EVENT_AXIS) {
        // x axis
        if (js.number == 1) {
            if (js.value < -1000 && pp->x > 0) pp->x--; // left
            if (js.value > 1000 && pp->x < MATRIX_SIZE - 1) pp->x
                ++; // right
        }

        // y axis
        if (js.number == 0) {
            if (js.value < -1000 && pp->y > 0) pp->y--; // up
            if (js.value > 1000 && pp->y < MATRIX_SIZE - 1) pp->y
                ++; // down
        }

        matrix_update();
        system("clear");
        matrix_print();
    }


}
```

## 3.6 Repository

For this project, I have created a repository on GitHub. GitHub is the most famous and most used git hosting website.

I have chosen to create a repository GitHub for these reasons:

- Synchronization: Git allows me to synchronize my work on all my devices with only one command: git pull
- Debugging: if something that was working, after some changes doesn't work anymore, I can go back as much as I want and see the differences.
- Open Source: my project is open source, so anyone who wants can use my code, modify it for his purpose, or contribute to it with a pull request.
- Issues: if a person uses my code and finds a bug. He can report it to me through the Issues section.

Here you can find a link to the repository: Github

# Chapter 4

# Use cases and experiments

Now that the project is complete, we are going to test the joystick in different scenarios. We are going to test with:

- The debugging console
- jstest-gtk
- The test game that we have created in Chapter 3

## 4.1 Initial phase

### 4.1.1 Clone the repository

Without code, we can't do anything, so the first thing to do is to clone the repository on our PC.

Before doing that we need to ensure that git is correctly installed on our machine. We can test it by doing:

```
git -v
```

If your response is something like:

```
git version 2.39.3
```

Everything is ok, otherwise, you need to install Git.
Now that we ensured that git is installed, we can clone the repository:

```
git clone https://github.com/iGieri/so-project.git
```

### 4.1.2 Transmitter and Receiver

Transmitter and receiver codes need to be compiled, converted to hex and uploaded to the respective Arduino.

For the transmitter, we need to move on his directory, and let make to compile everything:

```
cd arduino/
make
make arduino.hex
```

The Arduino directory is the transmitter's directory. So we move to that directory. Make command will call avr-gcc and compile the code, generating elf files. Make arduino.hex command is saying to make to get the arduino elf file, convert it to a hex file with avr-objcopy, and upload it to Arduino Uno with avrdude.

For the receiver, we have an identical process, but this time the directory will be "receiver".

Ensure that both Arduino are connected, or avrdude won't work.

### 4.1.3 Driver and Test

Driver and test are programs that will be executed on the PC. So we don't need to use AVR programs or upload them somewhere. We need only to compile them with gcc.

So let's move to the driver's directory and compile it.

```
cd driver/
make
```

Now we have generated elf files that can be executed by Linux.

We can do the same calling make but in the test directory.

## 4.2 Debugging Console

Now everything is set up, we can start testing.

The first test we can do is to see the debugging console. The driver prints on stdout all the information about the joystick: x-axis, y-axis and if the button is pressed.

To test it we need to:

1. Connect the transmitter to a power source and the receiver to the PC.
2. Execute the following command:

```
sudo ./driver
```

(superuser permission are required for serial communication)

Now the console will show the status of the controller, and it will update automatically and instantly.

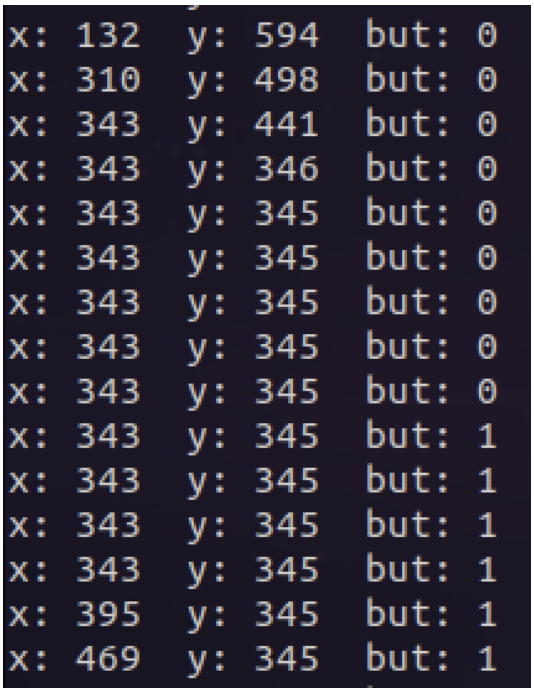


**Figure 4.1.** Console showing values

## 4.3 jstest-gtk

jstest-gtk is a gui for the jstest program. It's a program used to test joysticks that are recognized by Linux.

jstest-gtk has a lot of functionalities, some of which are:

- Have a graphical view of how Linux receive the joystick's information
- Calibrate the joystick to use it to play a videogame

To use jstest-gtk we need to install it, in Ubuntu-based operating systems we can run the following command:

```
sudo apt install jstest-gtk
```

So we can run jstest-gtk with the following command:

```
jstest-gtk
```

### 4.3.1 Graphical View

When we open jstest-gtk we need to choose which joystick we can use, we will probably have a lot of joysticks because a lot of mouse devices and touchpads are recognized by Linux also as joysticks.

We can recognize our joystick by the model name "Gieristick"

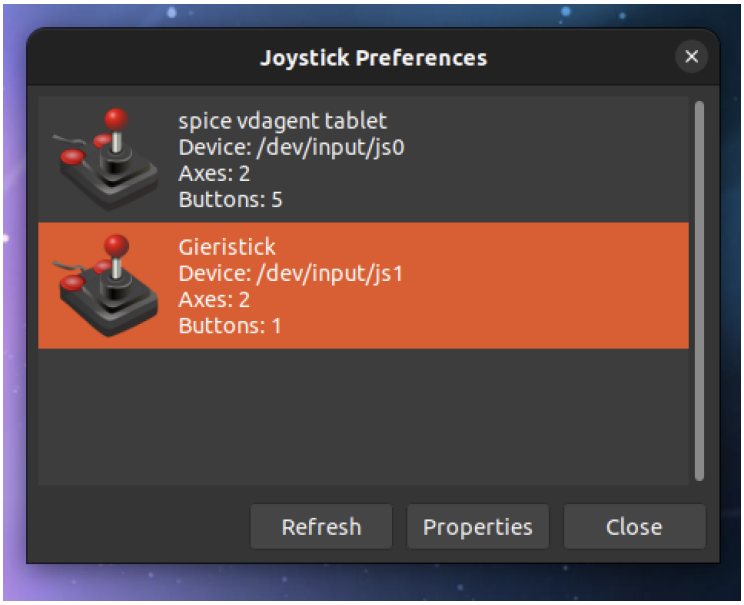


**Figure 4.2.** Joystick choosing screen

Now we have a new GUI in front of us showing the x-axis, y-axis and the button pressed.

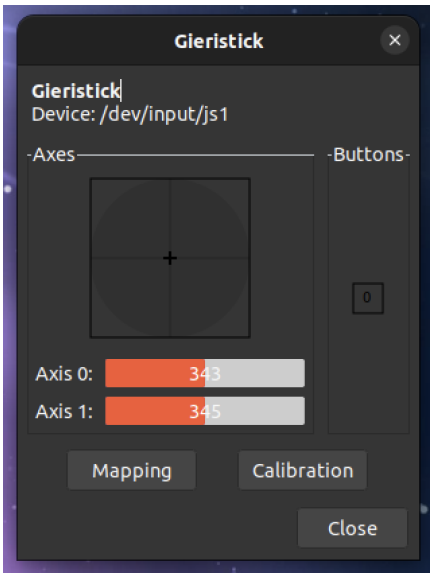


**Figure 4.3.** Joystick statistics

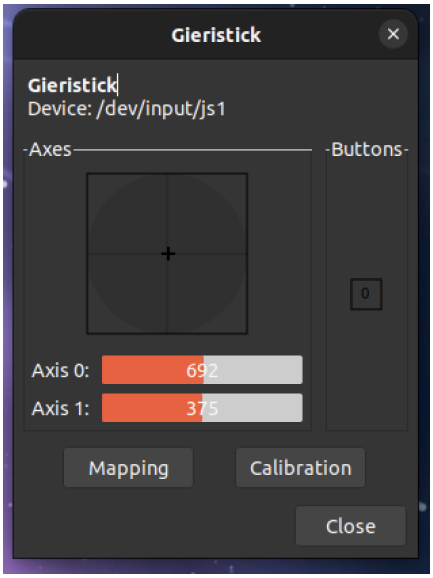


**Figure 4.4.** Axis 0 and Axis 1 changed

If we move the joystick also the gui will move live. We can notice that the joystick in the GUI doesn't move very much. We have to calibrate it.

### 4.3.2 Joystick calibration

To start the calibration, we need to press the button "Calibration"

This pop-up will appear:

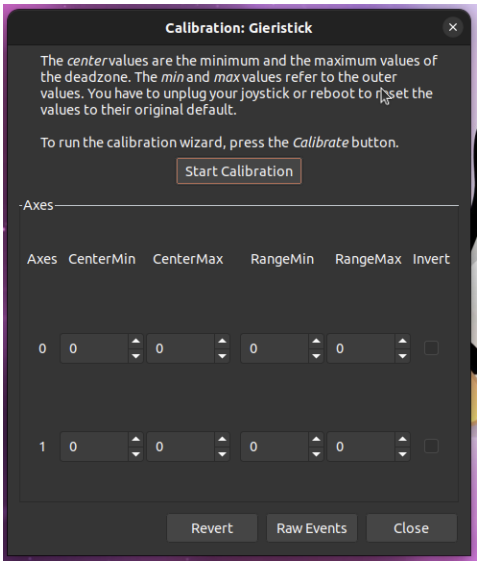


**Figure 4.5.** Calibration popup

Now we have to click "Start the calibration"

This phase is crucial, we have to move the joystick 360 degrees and then reposition it back to the centre and click on "OK".

Check that the values have changed in the pop-up

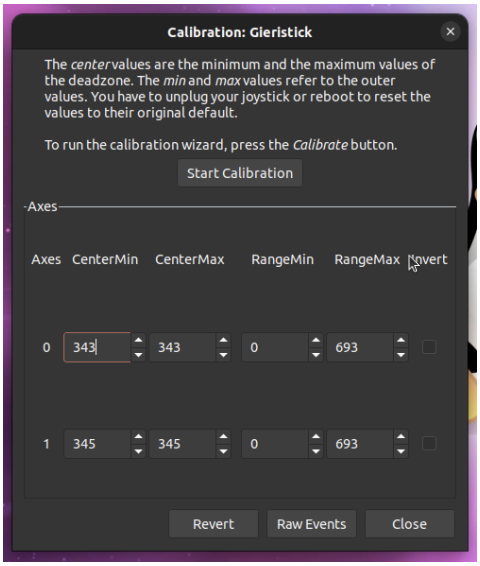


**Figure 4.6.** Values changed

Now the joystick is perfectly calibrated. As you can see in the picture now the joystick on the screen moves in the same way as the real joystick.

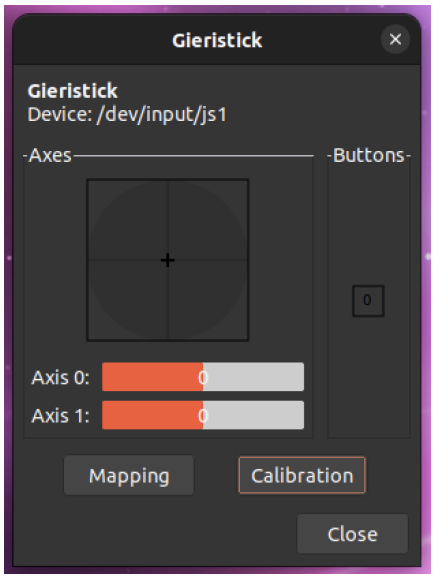


**Figure 4.7.** Joystick statistics

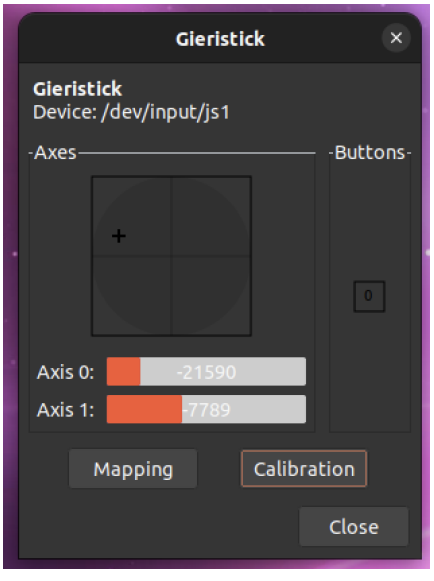


**Figure 4.8.** Axis 0 and Axis 1 changed

## 4.4 Test Game

Let's try our joystick on the game we have made in 3.5. We have already compiled it on 4.1.

We can start the game with the following command:

```
./test
```

It's important to run it with the driver running in the background, otherwise the game won't work.


**Figure 4.9.** Game

If we move the joystick, the star will move in the same direction. So we are playing the game!

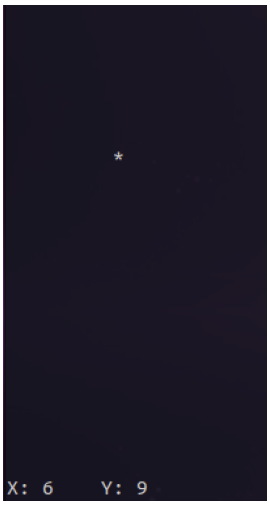


**Figure 4.10.** Star moved

# Chapter 5

# Conclusions

In this thesis, we have created a wireless joystick with Arduino, recognized by Linux thanks to its driver.

We have tested the joystick in different ways, with the console, with the jstest-gtk tool, and with a whole game.

The debugging console works perfectly, sometimes there is a value not correct caused by interferences. The input lag is very very low, slight.

The jstest-gtk tool and the test game have a small input lag, but evident. This is caused by the fact that HC-12 requires 9600 baud as baud rate and I am working on a virtual machine that does not have good specifics.

# Bibliography

[1] Hans Eirik Bull, Brian S. Dean, Stefan Ruger and Jorg Wunsch "*AVRDUDE, A program for downloading/uploading AVR microcontroller flash, EEPROM and more*", Version 8.0, 24 August 2024

[2] HC-12 Wireless Serial Port Communication Module User Manual

[3] Richard M. Stallman, Roland McGrath, Paul D. Smith "*GNU Make A Program for Directing Recompilation*", GNU make Version 4.4.1, February 2023

[4] Scott Chacon, Ben Straub "*Pro Git*", Version 2.1.434, 2024-09-04

# Acknowledgements

Desidero ringraziare il mio relatore, il Prof. Giorgio Grisetti, per il supporto, per avermi sempre sostenuto nelle decisioni progettuali e soprattutto per avermi fatto appassionare alla sua materia.

Ringrazio i miei genitori, Riccardo e Gioia, per il loro sostegno incondizionato e per avermi incoraggiato a inseguire i miei sogni con determinazione. Li ringrazio per il loro supporto morale e l'amore che mi hanno sempre donato, che mi hanno guidato fino a qui.

Voglio ringraziare la mia fidanzata Arianna, per avermi sostenuto nei momenti più difficili, per esserci sempre stata nei momenti più belli di questo percorso e per avermi sopportato nelle ore in cui mi ascoltava ripetere.

Un grande grazie a Gabriele Onorato, Pietro Costanzi Fantini e Soykat Amin. Amici da una vita, che mi hanno sempre incoraggiato e sostenuto, fino a raggiungere questo traguardo.

Infine ci tengo a ringraziare Cristian Di Iorio, Cristian Apostol, Leonardo Miralli, Edoardo Costariol e Alberto Guida per le ore passate insieme nelle aule studio, le pause caffè, le chiaccherate sul calcio e i pranzi a mensa.