
C++ Robotics with Arduino: A C to C++ Coding Adventure

Wookjin Chung

<https://fribot.com>

Fribot Co. Ltd.

Introduction

The reason for writing this new book is to provide a new learning opportunity for many users who are learning coding from scratch with 'neo-Arduino robot car'. At some point, I started to feel the limitations of coding education with just C language, and Python coding was gaining great popularity among the public. Furthermore, I believed there was a significant need to advance to C++ language, using the already in-use 'neo-Arduino robot car' to enhance coding skills up to an intermediate level of C++ as a means and motivation.

While preparing to launch a new product that combines Micro:bit and Arduino, which allows for Python language use, the intention was to provide a 'new weapon for coding education' that could elevate the level of education that had been lacking with just C language coding, for existing product users.

The expected readers are, of course, customers using the 'neo-Arduino robot car', and possibly those interested in the new products being launched by our company. However, even those who do not purchase the Arduino robot product can easily understand C++ coding language by reading this freely distributed book.

The origin and background of this book are based on the online distribution version of the coding robot from Parallax Inc. in the USA, which is the origin of 'neo-Arduino robot car'. The first and second parts of the 'Playing with Arduino Robot' online practical material distributed long ago by our company were translations based on the content from Parallax Inc. Additionally, various summaries, explanations, and example code development in this book were assisted by ChatGPT, and many illustrations used images from Dall-E.

This book aims not to duplicate but creatively maintain relevance with the content of previously distributed online practical materials. ChatGPT has been an undoubtedly excellent collaborator and a powerful producer. We hope that many customers who will benefit from reading this book will always communicate with ChatGPT and achieve great success.

We look forward to meeting you again with Fribot's new products in the near future and believe it won't be long before that happens. May the glory of God be with everyone.

by chung wookjin

***** For teachers teaching C and C++ languages with this book *****

This book is structured into 9 chapters. The content covered in this book may be insufficient for students who are completely new to the C language, as the basics of C language syntax and usage are intentionally omitted. For students who have no experience with C language, it is recommended to first learn the basics of handling 'neo-Arduino robot car' with C language and its basic grammar before using this book.

C and C++ languages, while similar, require a completely new concept and approach for C++ language, so it is recommended to first experience the basic grammar of C language before using this book. The core of the C++ content in this book is to learn the concept of classes through robot actions. Therefore, it intentionally does not cover C++ style I/O streams, namespace concepts, variable types and lifespans, especially modifiers and qualifiers, and the enhanced for loop in C++.

Here is a summary of the content covered in each chapter:

Chapter 1 - Understanding the SelfAbot Class: Introduces what C++ object-oriented programming is and uses Arduino examples to familiarize with the class concept. It introduces and explains the 'SelfAbot' class library code to be applied to the Arduino robot. This chapter focuses more on explaining the concept than on practice, so if students find it boring, it can be covered slowly at the end. The 'SelfAbot' class will continue to be explained in subsequent chapters.

Chapter 2 - Arduino Functions and Function Overloading: Describes and practices the use of class concepts in Arduino examples and the use of function overloading.

Chapter 3 - Robot Initialization and Basic Action Methods: Codes and practices the actions of the Arduino robot using the class concept. This is a practice of acquiring basic actions of the Arduino robot in the same way as mentioned in the first part of the 'Playing with Arduino Robot' online practical material distributed long ago.

Chapter 4 - Controlling the Movement of the Arduino Robot: Practices new driving actions of the Arduino robot from the previous chapters. Experience how to easily extend the robot's actions with the class concept. A practice of learning a more delicate and diverse moving Arduino robot.

Chapter 5 - Photo Transistor Sensor Light Signal Method: rcTime(): Introduces and practices the class method for handling light sensors. It's time to learn the expansion concept of handling sensor signals in various ways while using the same light sensor.

Chapter 6 - Autonomous Driving Robot with Infrared Sensor: You may have already tried driving practices with an infrared sensor. This chapter provides a practice experience of using infrared signals more diversely for different purposes.

Chapter 7 - Arduino Robot Equipped with Ultrasonic Sensor: Practices how ultrasonic sensors are integrated and used differently from infrared transceiver sensors on robots. It makes the robot's autonomous driving more intelligent with an ultrasonic sensor.

Chapter 8 - Creating Specialized Robots with Class Inheritance: A practice to convey the concept of inheritance in C++ class concepts to students. It creates a subclass with all the methods introduced in the previous chapters (practice examples) that are not included in the 'SelfAbot' basic class (superclass or parent class) and practices the class inheritance concept.

Chapter 9 - Advanced Movement and Control of the Robot: Introduces cases of wireless data transmission using Bluetooth/WiFi/Cellular/IoT communications based on the codes practiced up to the previous chapters, and practices what advantages and benefits are possible. It also introduces how to reflect various situations that may occur while the robot is operating with exception handling code.

Finally, the 'SelfAbot' class concept and code introduced in this book are not the final version or final result for operating the Arduino robot. This code is merely one example for students to understand the C++ class concept and does not reflect many other code purposes or forms.

Considering this, we encourage students to freely create their own Arduino robot classes and test them.

***** For those who want to self-study C++ language with this book *****

This book can also be recommended to those who have no experience with the C language. The intent of this book is to make it as easy as possible for coding beginners to understand the C++ coding language. However, mentioning the basics of C language grammar in this book would increase the volume and complexity, so it was omitted.

If you are completely unfamiliar with the basics of C language grammar, it is recommended to first read the basic grammar section described in the "Playing with Arduino Robot" online practical material part 1 distributed long ago. If you wish to watch on YouTube, you can view the video titled 'Elementary students can learn the basics of C language grammar in 40 minutes' at <https://youtu.be/joUNuEGv3i0>.

Learning to code has always been seen as a process of mastering the basics of coding grammar. However, after the emergence of ChatGPT, we must revise all perspectives and viewpoints. Even if you do not know coding grammar immediately and cannot write a single line of code, if you can imagine the process of code development and the effects of code execution, you are in an era where you can write high-level code using ChatGPT. We have reached the era where 'computational thinking' is most required. To put it bluntly, it may be difficult for me to write code line by line based on code grammar, but if I can request code writing to ChatGPT regarding the purpose and method of the code I want, and I can read, judge, and modify the code when ChatGPT responds, that ability is more required.

Therefore, 'coding grammar', which was essential for code writing, is no longer a mandatory learning threshold. Understanding the concept of coding has become more essential than coding grammar. Coding grammar and coding concepts are no longer synonymous. That's why we recommend experiencing this book first for readers who are learning to code for the first time. This book provides an opportunity to easily understand the C++ coding language, a deeper concept than the C language.

This book does not cover detailed descriptions or usage of electronic components for robot movement, nor does it introduce the process of assembling the robot for readers who purchase the Arduino robot for the first time. If you are looking for more detailed materials, you can refer to the online practical material mentioned above. And descriptions of sensor components are intentionally omitted as much as possible, and it was written to be newer and closer to object-oriented programming than the previous materials.

Finally, if you are learning C++ language coding for the first time with this book, it might be a good idea to briefly read through the previous materials after finishing this book. The basics are always important.

About the Author

The author has been working to spread 'Arduino' microcontrollers in Korea and distribute books through Fribot.com since 2012, while also acting as the official distributor of products from Parallax Inc. in the USA. The background of launching Arduino robot products and distributing many educational materials to domestic users is largely due to cooperation with Parallax Inc.

To contact the representative of Fribot and the author of this book, you can use the address/contact information published on the Fribot website (fribot.com).

You can use various channels such as phone, email(mail@fribot.com), and blog for contact.

Thank you.

Table of Contents

Chapter 1 Understanding the SelfAbot Class

- 1.1 C++ Object-Oriented Programming
- 1.2 Class Coding: Blinking an Arduino LED
- 1.3 Object-Oriented Programming: Encapsulation
- 1.4 Introduction to the 'SelfAbot' Class
- 1.5 Classes: Constructors, Member Variables, Constants
- 1.6 Member Functions and Overloading
- Appendix: 'SelfAbot.zip' Library Header and Source Files

Chapter 2 Arduino Functions and Overloading

- 2.1 Adding the 'SelfAbot' Library to Arduino and Utilizing Examples
- 2.2 Executing Arduino Functions
- 2.3 Simplification through Function Overloading

Chapter 3 Robot Initialization and Basic Action Methods

- 3.1 Implementing the setup Method for Robot Initialization
- 3.2 Robot Movement Properties and Methods
- 3.3 Simplification through Function Overloading
- 3.4 Standard Servo Motor Basic Actions
- 3.5 Continuous Rotation Servo Motor Basic Actions

Chapter 4 Controlling the Movement of the Arduino Robot

- 4.1 Centering the Servo Motor
- 4.2 Straight-line Travel and Robot Correction
- 4.3 Left Turn / Right Turn and Continuous Actions
- 4.4 Gradual Acceleration and Deceleration
- 4.5 Additional Method: maneuver()

Chapter 5 Phototransistor Light Signal: rcTime()

- 5.1 Practice for Processing Analog Input Signals
- 5.2 Digital Input Signal Method: rcTime() Method
- 5.3 Recognizing Light Intensity with Two Phototransistors
- 5.4 Arduino Robot Light Following Exercise

Chapter 6 Autonomous Driving Robot with Infrared Sensors

- 6.1 Infrared Sensor Signal Processing for Obstacle Detection
- 6.2 Integrating Infrared Sensor Signals into the Robot
- 6.3 Using the irDistance Method with Infrared Sensors
- 6.4 Scanning Surrounding Objects with Infrared Sensors
- 6.5 Finding Nearby Objects by Scanning with Infrared Sensors

Chapter 7 Arduino Robot Equipped with Ultrasonic Sensor

- 7.1 Processing Ultrasonic Sensor Signals
- 7.2 Displaying the Radar Screen for Objects Ahead
- 7.3 Driving with an Ultrasonic Sensor Robot

Chapter 8 Specialized Robots with Class Inheritance

- 8.1 Creating a Subclass with 'SelfAbot' as the Superclass
- 8.2 How to Write an .ino File Using an Inheritance Class
- Appendix: 'EnhancedSelfAbot.zip' Library Header and Source Files

Chapter 9 Advanced Movement and Control of the Robot

9.1 Implementing Advanced Functions of Wireless Robots

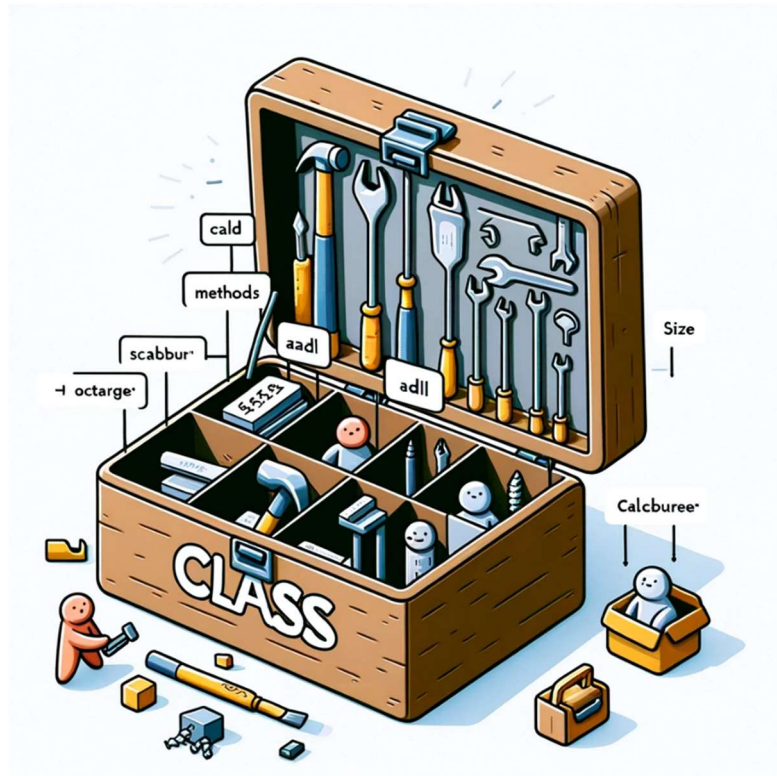
9.2 Managing Unanticipated Situations During Robot Operation

Appendix: Processing Source Code for Ultrasonic Radar Imaging

Chapter 1 Understanding the SelfAbot Class

- 1.1 C++ Object-Oriented Programming
- 1.2 Class Coding: Blinking an Arduino LED
- 1.3 Object-Oriented Programming: Encapsulation
- 1.4 Introduction to the 'SelfAbot' Class
- 1.5 Classes: Constructors, Member Variables, Constants
- 1.6 Member Functions and Overloading

Appendix: 'SelfAbot.zip' Library Header and Source Files



1.1 C++ Object-Oriented Programming

In C++, a class can be thought of as an 'extended form of a struct' from C. In the C language, a struct provides a way to bundle data into a single unit. Similarly, C++ classes bundle data, but with the additional capability to include functions (also called methods).

The basic concepts are as follows:

- (1) Data Encapsulation: A class bundles data (variables) and the functions (methods) to process this data into a single unit. This closely links the data with its processing logic, making the code structure clearer and easier to manage.
- (2) Access Control: Classes can control the accessibility of data and functions. Access specifiers like public, private, and protected can be used to differentiate between members that can be accessed from outside the class and those that can only be accessed internally.
- (3) Reusability and Extension: Using classes enhances code reusability. Once a class is defined, multiple objects can be created based on it. Furthermore, inheritance allows for the extension or

modification of existing class functionalities.

As a simple example, consider a Car class. This class might have data such as fuel amount (fuel) and speed (speed), and include functions (methods) like accelerate() and brake(). Using classes, the state and behavior of the car can be defined and managed in one place, which is convenient.

```
class Car {
private:
int fuel;
int speed;

public:
void accelerate() {
// Use fuel to increase speed
}

void brake() {
// Decrease speed
}
};
```

Thus, classes bundle data and functions into a single logical unit, facilitating code reusability and maintenance. Users of the C language, already familiar with grouping data using structs, can think of classes as '**structs with functionality**'.

1.2 Class Coding: Blinking an Arduino LED

Using the concept of classes, here's an example .ino for blinking an LED. See how Arduino code can be written using classes. Apply the following code to your Arduino, connecting the LED pin to pin 5. The method of connecting the LED and resistor to the Arduino is omitted.

Ex1.1 Cplus_LED_on_off.ino

```
#include "Arduino.h"
class SelfArduino {
public:
SelfArduino() {}
void setup() {}
void digitalWrite(byte pin, byte value) {
pinMode(pin, OUTPUT);
::digitalWrite(pin, value);
}

private:
};

SelfArduino adu;
int pin = 5;
void setup() {
adu.setup();
}

void loop() {
adu.digitalWrite(pin, HIGH);
delay(1000);
adu.digitalWrite(pin, LOW);
delay(1000);
}
```

This example can be used by adding the 'SelfAbot.zip' library to the Arduino and opening the example menu, where it is available in the SelfAbot folder as explained in Section 1 of Chapter 2.

The above code is an example of using classes without the 'SelfAbot' library, but for convenience, it has been included in the 'SelfAbot' library. You can open the Arduino IDE program, create a new sketch, and copy-paste the code above into it.

The code is explained as follows:

It demonstrates the use of the 'SelfArduino' class for interacting with Arduino's digital pins. This code follows the basic principles of C++ classes and Arduino programming and is straightforward. Here's a brief explanation:

(1) Including the Arduino library:

```
#include "Arduino.h"
```

This line includes the standard Arduino library, providing access to Arduino's common functions and types.

(2) Class Definition - SelfArduino:

```
class SelfArduino {
public:
  SelfArduino() { }
  void setup() { }
  void digitalWrite(byte pin, byte value) {
    pinMode(pin, OUTPUT);
    ::digitalWrite(pin, value);
  }
private:
  // Private members would go here
};
```

'SelfArduino' is a user-defined class. You create the name yourself. There is a constructor, SelfArduino(), that can be used to initialize the class.

The setup() method is empty but can be used for initial setup tasks later.

The digitalWrite() method is used to set the mode of a digital pin to OUTPUT and then write a value (HIGH or LOW) to that pin. The :: before digitalWrite indicates that it is calling a global function from the Arduino library, not a method belonging to the 'SelfArduino' class. If you name the digitalWrite(byte pin, byte value) method in the 'SelfArduino' class differently, you do not need to use the :: namespace prefix before ::digitalWrite(pin, value).

(3) Creating an Object of the 'SelfArduino' Class:

```
SelfArduino adu;
int pin = 5;
```

An object (or instance) adu of the 'SelfArduino' class is created. The pin is set to 5, indicating that digital pin 5 will be used.

(4) Arduino setup() Function:

```
void setup() {
  adu.setup();
}
```

This function runs once when the Arduino starts. The setup() method of adu can be used for initial setup, but since no specific code is written here, it does not perform any action yet.

(5) Arduino loop() Function:

```
void loop() {
  adu.digitalWrite(pin, HIGH);
}
```

```

    delay(1000);
    digitalWrite(pin, LOW);
    delay(1000);
}

```

This function runs repeatedly after setup(). It sets the digital pin 5 to HIGH (on) for 1000 milliseconds (1 second), waits, then sets it to LOW (off) and waits again for 1000 milliseconds. This creates a blinking effect on an LED or device connected to pin 5. It executes the digitalWrite method of the adu instance.

1.3 Object-Oriented Programming: Encapsulation

C++ object-oriented programming has very different characteristics from C language programming. While it satisfies many of the coding concepts of C language, it encompasses a more varied and comprehensive set of concepts.

C language programming is based on a procedural programming approach. Thus, the C language programming method typically involves writing code in the order that the program progresses, listing first the code for operations, then the code for input and output methods, and the types of data.

On the other hand, C++ object-oriented programming typically involves writing code according to the following principles, separate from the conceptual approach of data '**encapsulation/access control/reusability and extension**':

(1) First, declare an object. The object's composition can include public, private, and protected areas. Public area: Members can be accessed from anywhere outside the class where the object was created.

Private area: Members can only be accessed within the class itself, and not directly from outside the class.

Protected area: Members can be accessed within the class and derived classes (subclasses) but not from outside these classes.

(2) Code the design concept of how to handle data. For this, design the details of the object. Declare the types of data and write methods as means to process the data. This includes declaring constructors.

(3) Declare specific 'instances' using the class, and write code for their actual use.

(Note) -----

In object-oriented programming, "instance" specifically means a concrete occurrence of any object or structure, while "object" is a more general term for an object. Creating an object of a class essentially creates an "instance" of that class. Each instance has a unique set of properties and methods defined by the class, but the specific values of these properties and their operational states can vary between instances.

Encapsulation means making some members of an object private and providing public methods to access and interact with these private members. It involves bundling code that manipulates data with the data itself and restricting direct access to some components of an object to protect the integrity of the object. It's a core concept in class design to prevent unauthorized access and modification, promoting modularity and preventing external interference and misuse of the internal state of the object.

The Arduino example code written below adds encapsulation functionality, presenting an improved Arduino class file compared to before. To write encapsulated code, we will set some members as private and provide public methods to interact with them.

Ex1.2_CplusCapsule_LED_on_off.ino

```
#include "Arduino.h"

class SelfArduino {
public:
  SelfArduino() { }

  // Public method to setup pins
  void setup(byte pin) {
    _pin = pin;
    pinMode(_pin, OUTPUT);
  }

  // Public method to write value to the pin
  void digitalWrite(byte value) {
    ::digitalWrite(_pin, value);
  }

private:
  byte _pin; // Private member variable to store the pin number
  // You can add more private methods and variables here
};

SelfArduino adu;
int pin = 5;

void setup() {
  adu.setup(pin); // Setup the pin for abot
}

void loop() {
  adu.digitalWrite(HIGH); // Turn on the pin
  delay(1000);
  adu.digitalWrite(LOW); // Turn off the pin
  delay(1000);
}
```

Explained is the encapsulated version example for turning the LED light on and off.

(1) `_pin` is a private member variable of the 'SelfAbot' class used to store the pin number. Since this variable cannot be accessed directly from outside the class, it is encapsulated within the class.

(2) The `setup` method is public and used to initialize the pin. It sets the private `_pin` variable and initializes the pin mode.

(3) The `digitalWrite` method only requires a value ('HIGH' or 'LOW') and writes it to the pin specified by the private `_pin` variable. This method provides controlled access to the private `_pin` member.

The encapsulation example demonstrates how to hide the internal state of an object and expose only what is necessary for interaction with the outside world. Using this approach allows for better modular design, making the code easier to maintain and extend.

(Note) -----

`_pin` variable: The reason for using an underscore ('_') symbol before private variable names

(1) Scope clarity: The underscore prefix helps distinguish private members (variables or methods) from public or protected members.

(2) Avoiding name clashes: The underscore prefix for class members helps avoid name clashes and clarifies when class members are used instead of local variables. Especially in some coding practices using Getter/Setter methods, getter and setter methods have the same name as the variable. Using an underscore for member variables helps distinguish between methods and variables (e.g., 'age' for getter, '_age' for the private member).

(3) Rules and readability: The underscore prefix for private members can become part of these standards to ensure consistency across the codebase. It visually segregates the code, making it easier to read and understand at a glance what variables are within a class.

(4) Historical and language-specific conventions: Python uses double underscores (__) for name mangling, which is a stronger rule indicating the variable is private, but a single underscore is commonly used as a "soft" private indicator. It's more of a convention telling other developers it should be treated as a private member. The single underscore prefix is also used in languages like C++, but it is more of a convention than a language requirement.

The following notes explain why class files are separated into header files and source files, or how they are divided and created.

(Note) -----

How to separate class files into header (.h) and source (.cpp) files ?

In C++, dividing class definitions into header (.h) files and implementation (.cpp) files is common practice for several reasons.

(1) Separation of interface and implementation: This separation allows the interface (defined in the .h file) to be separated from the implementation details (found in the .cpp file). Header files declare what the class (or function) does, and implementation files define how the class (or function) does it. This abstraction is useful for encapsulation, a core principle of object-oriented programming.

(2) Reduced compile time: If the implementation in a .cpp file is changed, only that file needs to be recompiled, not all files that include the .h file. This can significantly reduce compile time in large projects.

(3) Avoiding multiple definitions: In C++, the One Definition Rule (ODR) states that an entity (e.g., class, function, or variable) must have exactly one definition in the code. Placing definitions in .cpp files and declarations in .h files prevents multiple definitions of the same entity that could occur when the same header is included in multiple source files.

(4) Improved readability and organization: Keeping the interface separate from the implementation makes the code easier to read and navigate. Developers can quickly understand the interface of a class by looking at the header files without getting bogged down in implementation details.

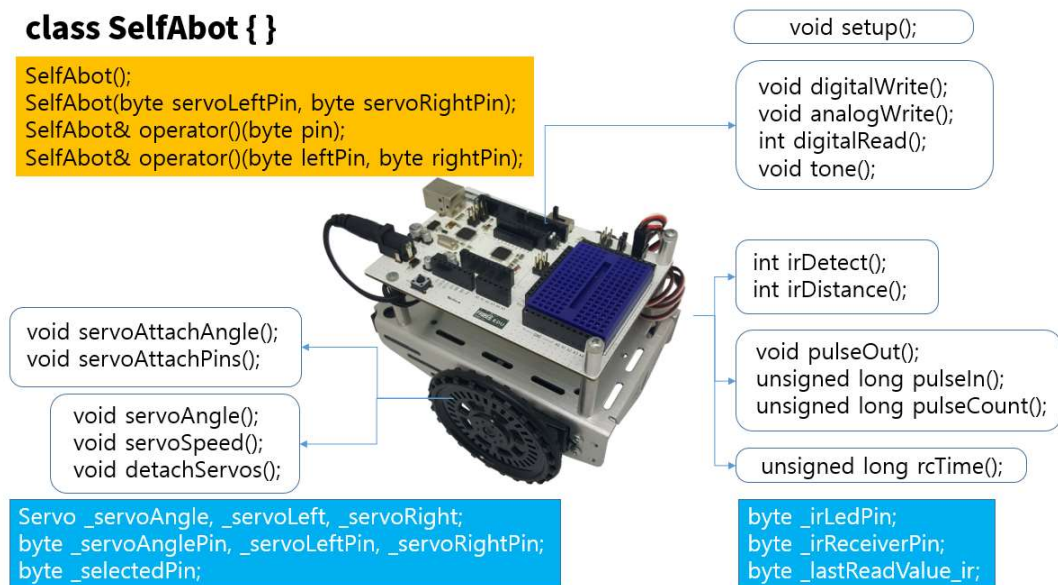
(5) Better for team development: In a team environment, different team members can work on different parts of the code. A clear separation allows for parallel development without many conflicts. One team member can work on implementation while another works on the interface or uses the interface in other parts of the project.

(6) Encouraging reusability and modularity: By separating the implementation, the same header file can be shared across multiple projects. This modularity makes the code easier to reuse.

(7) Forward declaration and reduced dependencies: Using forward declarations in header files reduces the need to include other headers, decreasing coupling and dependencies between different parts of the codebase.

In summary, separating .h and .cpp files in C++ is not just a stylistic choice but a fundamental practice that supports the principles of the language and especially improves the development process in larger, more complex projects.

1.4 Introduction to the 'SelfAbot' Class



Introducing the 'SelfAbot' class for operating the ['neo-Arduino robot'](#).

Below, we expand on the previously discussed class concept to introduce a more generalized concept that allows for understanding beyond the specific 'SelfAbot' class. Therefore, please note that the following description introduces a broader concept of the actual 'SelfAbot' class.

(1) 'SelfAbot' Class:

This is the blueprint for the 'neo-Arduino robot' class. It defines the robot's operational functions (methods) and characteristics (attributes/variables) and does nothing beyond that.

(2) 'SelfAbot' Object Design:

Object design refers to the process of designing the entire class code, consisting of .h and .cpp files. Below, we have listed detailed elements for your review. Consider the various aspects described below when designing your class.

'SelfAbot' Object Conceptualization:

When thinking about what the 'SelfAbot' class can do or have (e.g., move forward, move backward, or servo status), describing the 'SelfAbot' object in design terms involves specifically defining these capabilities.

Methods and Attributes Definition:

For example, method code could include actions such as moving forward, stopping, changing direction, or interacting with sensors. Attribute variables describing the characteristics or state of the 'SelfAbot' robot could include variables for pins connected to servos, current speed, sensor values, etc.

Abstraction and Encapsulation:

Abstraction: When designing the 'SelfAbot' object, you abstract the complexity. You are not interested in how servos work at the electrical level or the details of the microcontroller's I/O pins. Instead, you focus on higher-level functionalities.

Encapsulation: This design process includes determining which internal details should be hidden from the outside (encapsulation) and which details should be exposed as a public interface (public methods and properties).

Interaction and Relationships:

Interactions with other objects: Part of the design is understanding how 'SelfAbot' will interact with other

objects or systems. This can include receiving commands, processing sensor data, and communicating with external devices.

Relationships and Hierarchies: If there are different types of robots or components, you might design 'SelfAbot' with inheritance or composition in mind. For example, 'SelfAbot' could be the base class for more specialized robot types or assemble functionality from smaller, more focused classes.

Use Cases and Scenarios:

Imagining Use Cases: Imagining various scenarios where 'SelfAbot' will be used helps in designing the object. Examples include navigating a maze, following a line, or reacting to sensor inputs. These use cases require the development of relevant methods and properties.

Planning for Flexibility and Scalability:

Adaptability: In the design, consider how easily the 'SelfAbot' object can be adapted or extended. For example, adding new functionalities or adapting to different hardware configurations.

In summary, the designed 'SelfAbot' object is a conceptual model of what the robotic system should be able to do and what characteristics it should have. It combines defining functionalities, properties, interactions, and potential use cases with principles like abstraction, encapsulation, and adaptability in mind.

(3) 'SelfAbot' Instances in Code:

In the actual application code, writing something like `SelfAbot myRobot;` creates `myRobot` as an instance of the 'SelfAbot' class. It is a specific robot object that can interact within the program. Many instances (e.g., `myRobot`, `yourRobot`, etc.) can be created and distinguished, and each instance becomes a real entity with its own unique state and behavior as defined by the 'SelfAbot' class.

The 'SelfAbot' class is an abstract blueprint, object design is a generalized code concept, and an instance is the implementation of that class concept in a concrete form within the code. The header file (`SelfAbot.h`) and source file (`SelfAbot.cpp`) of the 'SelfAbot' class are introduced at the end of Chapter 1.

1.5 Classes: Constructors, Member Variables, Constants

(1) Constructors

Default Constructor:

The default constructor is a special type of constructor method that is called when an object is created without any arguments. For 'SelfAbot', the default constructor could set initial values for member variables such as servo pin numbers or default states.

Example: `SelfAbot::SelfAbot() { _selectedPin = INVALID_PIN; }`

Parameterized Constructor:

This type of constructor allows passing parameters when creating an object to initialize it with specific values. For 'SelfAbot', a parameterized constructor could use the pin numbers for the left and right servos as arguments, allowing the user to specify the pins to be used for these components.

Example: `SelfAbot::SelfAbot(byte ServoLeftPin, byte ServoRightPin) { ... }`

(2) Member Variables and Constants

Member Variables:

Variables defined inside a class. They maintain the state of an object. In 'SelfAbot', member variables could include servo pin numbers, current speed, or status flags.

Example: `byte _servoLeftPin;`

Constants:

Constants are values that, once set, do not change. In C++, they are often defined using `const` or `#define`. For 'SelfAbot', constants like `INVALID_PIN` could be used to represent invalid or undefined pin numbers.

Example: `const byte INVALID_PIN = 255;`

1.6 Member Functions and Overloading

(1) Member Functions (also called 'Methods')

Member functions are code written within a class that defines the actions (or operations) an object created from that class can perform. In the context of arduino or C++ environments, dealing with classes is about creating the blueprint for a robot. For example, in an Arduino class, `digitalWrite()` is an action that creates a digital output of 5V or 0V on a pin specified by the code. The `SelfAbot` class also defines various member functions, which will be explained in more detail later.

(2) Function Overloading

Function overloading is a feature of object-oriented programming where two or more functions (or methods) can have the same name but different parameters. It's a way to create multiple methods with the same name that perform similar tasks but may require different inputs. Thus, function overloading allows for multiple methods to have the same name but different parameters.

This provides flexibility in how methods are used. The example code below briefly explains function overloading, but there will be an opportunity to see how function overloading is used in the actual 'SelfAbot' class code for the arduino robot and what effects it has.

Two examples of class methods usage:

```
void SelfAbot::digitalWrite(byte value) and  
void SelfAbot::digitalWrite(byte pin, byte value) are both valid and used for different purposes.
```

For example, in the `digitalWrite()` function, it's possible to pass both the pin and value parameters together, or use function overloading to pass the pin value as an instance value and only the value parameter. When representing abot instance code, both of the following forms are possible.

Here is an example of not using function overloading versus using it in Arduino .ino code. The two different code usages below actually show the same code execution result, just represented differently.

```
SelfAbot abot; // Creating an abot instance  
  
// Example code not using function overloading  
abot.digitalWrite(pin, value);  
  
// Example code using function overloading  
abot(pin).digitalWrite(value);
```

Appendix: 'SelfAbot.zip' library header and source files (SelfAbot.h & SelfAbot.cpp)

```
//arduino robot class library SelfAbot.h' @0.1.10
#ifndef SelfAbot_h
#define SelfAbot_h

#include "Arduino.h"
#include <Servo.h>

class SelfAbot {
public:
    SelfAbot();
    SelfAbot(byte servoLeftPin, byte servoRightPin);
    SelfAbot& operator()(byte pin);
    SelfAbot& operator()(byte leftPin, byte rightPin);

    void setup();

    void digitalWrite(byte pin, byte value);
    void digitalWrite(byte value);
    void analogWrite(byte pin, int value);
    void analogWrite(int value);
    int digitalRead(byte pin);
    int digitalRead();
    void tone(byte pin, unsigned int frequency, unsigned long duration);
    void tone(unsigned int frequency, unsigned long duration);

    void servoAttachAngle(byte servoAnglePin); //
    void servoAttachPins(byte servoLeftPin, byte servoRightPin); //

    void servoAngle(int angle);
    void servoSpeed(int speed);
    void servoSpeed(int leftSpeed, int rightSpeed);
    void detachServos();

    int irDetect(byte irLedPin, byte irReceiverPin, int frequency);
    int irDistance(byte irLedPin, byte irReceivePin);

    void pulseOut(byte pin, unsigned long duration);
    unsigned long pulseIn(byte pin, int state);
    unsigned long pulseCount(byte pin, unsigned long duration);

    unsigned long rcTime(byte pin);

private:
    byte _servoAnglePin, _servoLeftPin, _servoRightPin;
    Servo _servoAngle, _servoLeft, _servoRight;
    byte _selectedPin;

    byte _irLedPin;
    byte _irReceiverPin;
    byte _lastReadValue_ir;
};

#endif
```

```

//arduino robot class library SelfAbot.h' @0.1.10
#include "SelfAbot.h"
#define INVALID_PIN 255

int SelfAbot::_lastReadValue = 0;

SelfAbot::SelfAbot() {
    _selectedPin = INVALID_PIN;
}

SelfAbot::SelfAbot(byte servoLeftPin, byte servoRightPin)
: _servoLeftPin(servoLeftPin), _servoRightPin(servoRightPin) {
    _selectedPin = INVALID_PIN;
}

SelfAbot& SelfAbot::operator()(byte pin) {
    _selectedPin = pin;
    return *this;
}

SelfAbot& SelfAbot::operator()(byte leftPin, byte rightPin) {
    _servoLeftPin = leftPin;
    _servoRightPin = rightPin;
    return *this;
}

void SelfAbot::setup() {
    if (_servoLeftPin != INVALID_PIN && _servoRightPin != INVALID_PIN) {
        _servoLeft.attach(_servoLeftPin);
        _servoRight.attach(_servoRightPin);
    }
}

// Original method
void SelfAbot::digitalWrite(byte pin, byte value) {
    pinMode(pin, OUTPUT);
    ::digitalWrite(pin, value);
}

// Overloaded method
void SelfAbot::digitalWrite(byte value) {
    if (_selectedPin != INVALID_PIN) {
        pinMode(_selectedPin, OUTPUT);
        ::digitalWrite(_selectedPin, value);
    }
}

// Original method
void SelfAbot::analogWrite(byte pin, int value) {
    ::analogWrite(pin, value);
}

// Overloaded method
void SelfAbot::analogWrite(int value) {
    if (_selectedPin != INVALID_PIN) {
        ::analogWrite(_selectedPin, value);
    }
}

// Original method
int SelfAbot::digitalRead(byte pin) {

```

```

        pinMode(pin, INPUT);
        return ::digitalRead(pin);
    }
    // Overloaded method
    int SelfAbot::digitalRead() {
        if (_selectedPin != INVALID_PIN) {
            pinMode(_selectedPin, INPUT);
            return ::digitalRead(_selectedPin);
        }
    }

    // Original method
    void SelfAbot::tone(byte pin, unsigned int frequency, unsigned long duration) {
        ::tone(pin, frequency, duration);
    }
    // Overloaded method
    void SelfAbot::tone(unsigned int frequency, unsigned long duration) {
        if (_selectedPin != INVALID_PIN) {
            ::tone(_selectedPin, frequency, duration);
        }
    }
}

// Original method
void SelfAbot::servoAttachPins(byte servoLeftPin, byte servoRightPin) {
    if (servoLeftPin != INVALID_PIN) {
        _servoLeftPin = servoLeftPin;
        _servoLeft.attach(_servoLeftPin);
    }
    if (servoRightPin != INVALID_PIN) {
        _servoRightPin = servoRightPin;
        _servoRight.attach(_servoRightPin);
    }
}

// Original method
void SelfAbot::servoAttachAngle(byte servoAnglePin) {
    if (servoAnglePin != INVALID_PIN) {
        _servoAnglePin = servoAnglePin;
        _servoAngle.attach(_servoAnglePin);
    }
}
// Overloaded method
void SelfAbot::servoAngle(int angle) {
    _servoAngle.write(angle);
}
// Overloaded method
void SelfAbot::servoSpeed(int speed) {
    if (_selectedPin == _servoLeftPin) {
        _servoLeft.writeMicroseconds(1500 + speed);
    } else if (_selectedPin == _servoRightPin) {
        _servoRight.writeMicroseconds(1500 + speed);
    }
}
// Overloaded method
void SelfAbot::servoSpeed(int leftSpeed, int rightSpeed) {
    _servoLeft.writeMicroseconds(1500 + leftSpeed);
    _servoRight.writeMicroseconds(1500 + rightSpeed);
}
// Overloaded method

```

```

void SelfAbot::detachServos() {
    _servoLeft.detach();
    _servoRight.detach();
}

// Original method
int SelfAbot::irDetect(byte irLedPin, byte irReceiverPin, int frequency) {
    _irLedPin = irLedPin;
    _irReceiverPin = irReceiverPin;
    pinMode(_irLedPin, OUTPUT);
    pinMode(_irReceiverPin, INPUT);
    tone(_irLedPin, frequency, 8);
    delay(1);
    int _lastReadValue_ir = ::digitalRead(_irReceiverPin);
    delay(1);
    return _lastReadValue_ir;
}

// Original method
int SelfAbot::irDistance(byte irLedPin, byte irReceivePin) {
    int distance = 0;
    for(long f = 38000; f <= 42000; f += 500) {
        distance += irDetect(irLedPin, irReceivePin, f);
    }
    return distance;
}

// Original method
void SelfAbot::pulseOut(byte pin, unsigned long duration) {
    pinMode(pin, OUTPUT);
    digitalWrite(pin, HIGH);
    delayMicroseconds(duration);
    digitalWrite(pin, LOW);
}

// Original method
unsigned long SelfAbot::pulseIn(byte pin, int state) {
    pinMode(pin, INPUT);
    return ::pulseIn(pin, HIGH);
}

// Original method
unsigned long SelfAbot::pulseCount(byte pin, unsigned long duration) {
    pinMode(pin, INPUT);
    unsigned long startTime = micros();
    unsigned long endTime = startTime + duration;
    unsigned long count = 0;

    while (micros() < endTime) {
        if (::digitalRead(pin) == HIGH) {
            count++;
            while (::digitalRead(pin) == HIGH);
        }
    }
    return count;
}

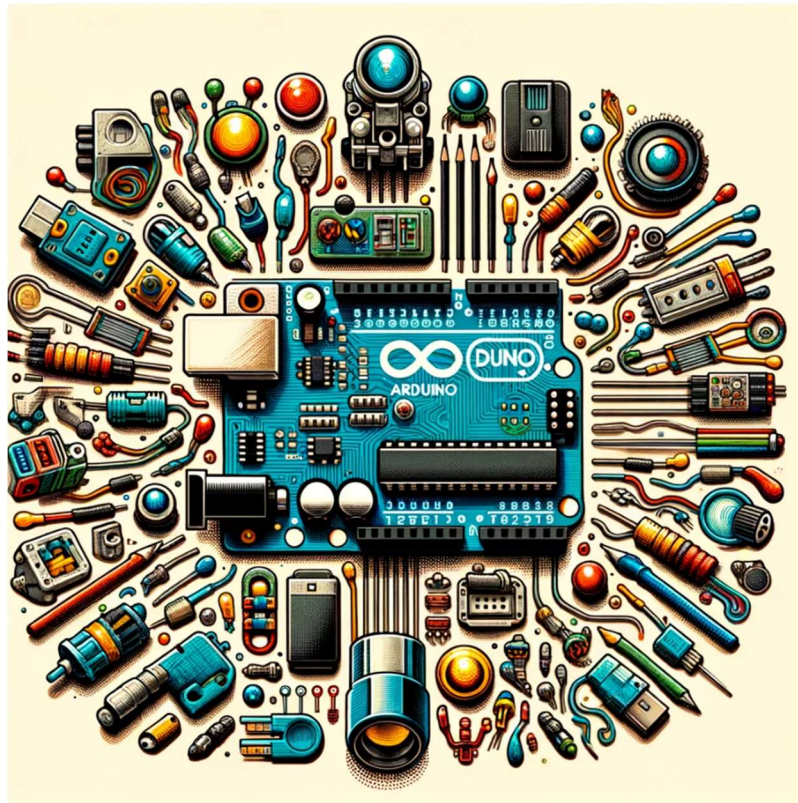
// Original method
unsigned long SelfAbot::rcTime(byte pin) {

```

```
pinMode(pin, OUTPUT);
digitalWrite(pin, HIGH);
delay(1);
pinMode(pin, INPUT);
digitalWrite(pin, LOW);
unsigned long startTime = micros();
unsigned long elapsedTime = 0;
while (digitalRead(pin) == HIGH) {
    elapsedTime = micros() - startTime;
}
return elapsedTime;
}
```

Chapter 2 Arduino Functions and Overloading

- 2.1 Adding the 'SelfAbot' Library to Arduino and Utilizing Examples
- 2.2 Executing Arduino Functions
- 2.3 Simplification through Function Overloading



This book is a textbook designed to learn C++ object-oriented programming using the 'Arduino Uno' microcontroller for educational robots. It includes standard Arduino input and output functions in the 'SelfAbot' class, which is designed to learn C++ coding. Therefore, before diving into C++ robot coding, it guides you to first learn how standard Arduino functions are used in the class.

If you're more interested in the operation of the C++ Arduino robot, you may skip Chapter 2 and go directly to Chapter 3.

2.1 Adding the 'SelfAbot' Library to Arduino and Utilizing Examples

The 'SelfAbot' library, designed to easily learn the concept of classes in the C++ language through Arduino robots, has been introduced at the beginning, but you can receive notifications on the Fribot website (<https://fribot.com>) or download the file made public on GitHub (<https://github.com/wookjin-chung/SelfAbot>). Even if you found the class concept explanation in Chapter 1 to be tedious, from now on, you will be guided to feel interested through direct practice. (Note: After downloading from GitHub, you should rename the file to “SelfAbot.zip“.)

There are various ways to add libraries in Arduino. Here, the method of directly adding the 'SelfAbot' library is described. Download the Arduino IDE program and install the library. The 'Arduino robot' used in the practice is the 'Fribee EDU' Arduino Uno compatible board based on the Arduino UNO.

Adding a Library to the Arduino IDE Program

- (1) Download the SelfAbot.zip file. (※ Download from GitHub)
- (2) Select 'Add Library' in the Arduino IDE.

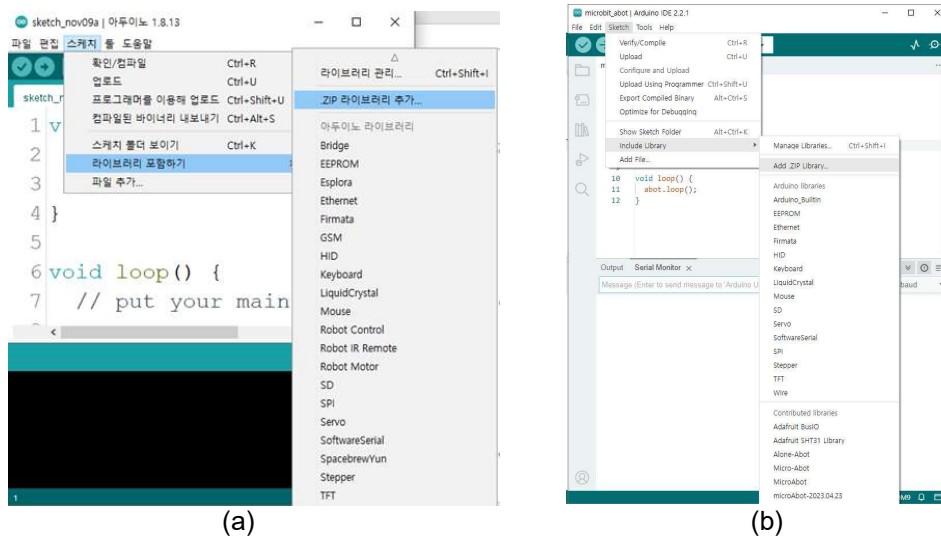


Figure 2.1 (a) Older version of the Arduino IDE program (b) Newer version of the Arduino IDE program

- Click and select the 'SelfAbot.zip' file you downloaded on your computer a moment ago.
- (3) Now your computer is ready to use the 'SelfAbot' library.

Utilizing Arduino 'SelfAbot' Library .ino Examples

Once you add the library to the Arduino program, you can use the various .ino examples included in the File->Examples->SelfAbot folder. The examples are organized by chapter, making it easy to find and use the examples introduced in each chapter of this book.

The 'SelfAbot' library folder in Arduino also includes the EnhancedSelfAbot library along with SelfAbot.

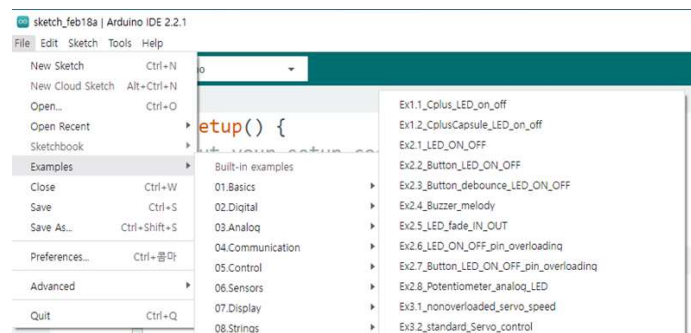


Figure 2.2: Utilizing Arduino SelfAbot Library Examples

If the content up to Chapter 7 uses examples utilizing the 'SelfAbot' library, Chapter 8 can use examples utilizing the subclass library 'EnhancedSelfAbot', which makes use of class inheritance. Try using methods to handle the robot more delicately and concisely.

2.2 Executing Arduino Functions

2.2.1 Turning the LED On and Off with the digitalWrite() Function

Connect the anode (+) of the LED to pin 5 of the Arduino, then connect it in series with a 220Ω resistor, and finally connect it to the ground (- : GND). Refer to Figure 2.3 for the method of connecting the LED. Once the hardware setup is complete, upload the code below to the Arduino and observe the LED light turning on and off at 1-second intervals.

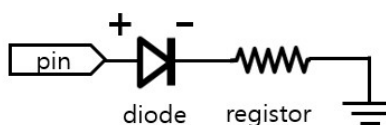


Figure 2.3: How to Connect an LED Circuit to Arduino Pins

What are the differences between the Ex2.1_LED_ON_OFF.ino code in C++ format (left code) and the code for turning the Arduino LED on and off in C language format (right code)?

Upload the Ex2.1_LED_ON_OFF.ino file. The left code is written in C++ language, and the right code is written in C language.

| | |
|--|--|
| <pre>#include "SelfAbot.h" SelfAbot abot; int pin = 9; void setup() { // abot.setup(); } void loop() { abot.digitalWrite(pin, HIGH); delay(1000); abot.digitalWrite(pin, LOW); delay(1000); }</pre> | <pre>int pin = 9; void setup() { pinMode(pin, OUTPUT); } void loop() { digitalWrite(pin, HIGH); delay(1000); digitalWrite(pin, LOW); delay(1000); }</pre> |
|--|--|

The line `#include "SelfAbot.h"` includes the definition of the 'SelfAbot' class. It instructs the Arduino IDE to use the code defined in the SelfAbot.h file. The library code must be in the same folder as this sketch or in the Arduino libraries folder.

`SelfAbot abot;` Here, an object (or instance) called `abot` of the `SelfAbot` class is created. Just as a personal car is a specific instance of the general concept of a car, think of 'abot' as a specific instance of the 'SelfAbot' class.

`int pin = 9;` This line declares an integer variable named `pin` and initializes it with the value 9. This number represents the digital pin number on the Arduino board to be used in this sketch.

`//abot.setup();` This line is commented out (`//` indicates a comment). If activated, it would call the `setup` method for the `abot` instance. This method is used for initial setup of the 'SelfAbot' class, such as setting pin modes or initializing components.

`abot.digitalWrite(pin, HIGH);` Calls the `digitalWrite` method of the 'SelfAbot' class within the `abot` instance. It sets the digital pin (in this case, pin number '9') to 'HIGH'. This means 5V, turning the pin on. The method internally includes the `pinMode()` setting, so it does not need to be executed separately.

```
void SelfAbot::digitalWrite(byte pin, byte value) {
  pinMode(pin, OUTPUT);
  ::digitalWrite(pin, value);
}
```

```
}
```

delay(1000); This function pauses the program for 1000 milliseconds (or 1 second). It is used here to maintain the pin at a high level for a short period.

abot.digitalWrite(pin, LOW); Similar to the previous digitalWrite, but this time it sets the pin to 0V, or LOW, turning off anything connected to that pin.

delay(1000); Pauses the program for another second, keeping the pin low during that time.

2.2.2 Using the digitalWrite() Function for Button Input and Output

The code below is understood in the same context as the previously explained code, but it uses a new method of the 'SelfAbot' class, digitalWrite().

Figure 2.4 shows how to connect a digital signal button input to an Arduino pin and, at the same time, connect an LED circuit to output a digital signal according to the button input signal. The pin numbers for the LED and button pins should match the sketch number of the user.

The code abot.digitalRead(buttonPin) is a method defined to use Arduino's digitalRead function. It is used with the abot instance, and the value is stored in the buttonState variable. Then, it is written to operate the digitalWrite function differently using an if statement, depending on whether the variable value is '1' or '0'.

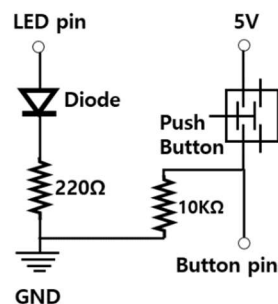


Figure 2.4: How to Connect a Button to Arduino Pins (Pull-Down Method)

Ex2.2_Button_LED_ON_OFF.ino

```
#include "SelfAbot.h"
SelfAbot abot;
const int buttonPin = 8;
const int ledPin = 9;
int buttonState = 0;

void setup() {
  // abot.setup();
}

void loop() {
  buttonState = abot.digitalRead(buttonPin);
  if (buttonState == HIGH) {
    abot.digitalWrite(ledPin, HIGH);
  } else {
    abot.digitalWrite(ledPin, LOW);
  }
}
```

For reference, you can explore more about the button circuit by consulting the 'Playing with Arduino Robots' online practical material distributed a long time ago, or the circuit diagram introduced in Section 3 of Chapter 2 of this book.

One more thing to note about button circuits is that because button switches generate digital signals

through physical/mechanical contact, pressing the button doesn't just create an ON/OFF electrical signal once, but it can happen multiple times. This issue can make the microcontroller's input signal processing unstable.

The debouncing code introduced below is a toggle method that ensures only a single on/off signal is registered when pressing or releasing the button, regardless of the physical bouncing that occurs when the button state changes. Although it's possible to create a debouncing effect using external circuits with resistors and capacitors, the software method is generally used because it's easier and more cost-effective to implement.

The timing for debouncing can be adjusted to best suit the user's convenience in the code below; `debounceDelay` is the debounce time. It enhances the reliability of the signal generation in the button circuit and can be set to sensitive conditions to user reactions.

(Note) What is Debounce? -----

Debounce is a technique that, in the case of multiple requests sent by a user, only executes once at the beginning or end and ignores the events during a set period of time.

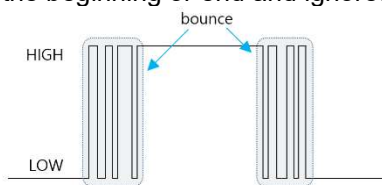


Figure 2.5: Conceptual Diagram of Debouncing

Ex2.3 Button_debounce_LED_ON_OFF.ino

```
#include "SelfAbot.h"
SelfAbot abot;

const int buttonPin = 8;
const int ledPin = 9;

int ledState = HIGH;
int buttonState;
int lastButtonState = LOW;

unsigned long lastDebounceTime = 0;
unsigned long debounceDelay = 50;

void setup() {
  abot.setup();
  abot.digitalWrite(ledPin, ledState);
}

void loop() {
  int reading = abot.digitalRead(buttonPin);

  if (reading != lastButtonState) {
    lastDebounceTime = millis();
  }
  if ((millis() - lastDebounceTime) > debounceDelay) {
    if (reading != buttonState) {
      buttonState = reading;
      if (buttonState == HIGH) {
        ledState = !ledState;
      }
    }
  }
}
```

```

abot.digitalWrite(ledPin, ledState);
lastButtonState = reading;
}

```

2.2.3 Making Sound with the tone() Function

This is an Arduino sketch for playing melodies using a buzzer connected to Arduino. The tone() function is used to create sounds of various pitches, and here, we introduce a practice of playing melodies using the tone method written in the 'SelfAbot' class. The tone method written in the class is as follows. Two methods are versions that do not use function overloading and versions that do.

```

void SelfAbot::tone(byte pin, unsigned int frequency, unsigned long duration) {
  ::tone(pin, frequency, duration);
}
void SelfAbot::tone(unsigned int frequency, unsigned long duration) {
  if (_selectedPin != INVALID_PIN) {
    ::tone(_selectedPin, frequency, duration);
  }
}

```

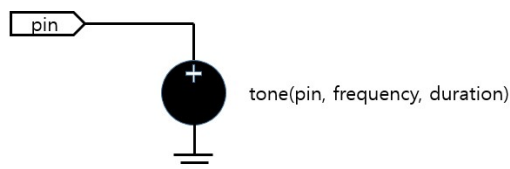


Figure 2.6: Schematic Diagram of Buzzer (Piezo Speaker) to Arduino Pins

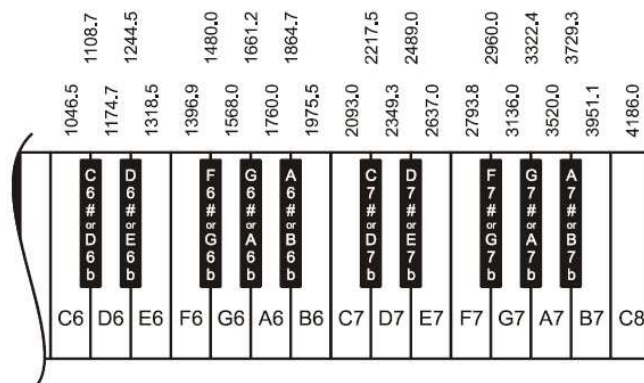


Figure 2.7: Frequency Values for Each Musical Note (Image source: parallax.com)

Ex2.4_Buzzer_melody.ino

```

#include "SelfAbot.h"
#include "pitches.h"

SelfAbot abot;

int melody[] = {
  NOTE_C4, NOTE_G3, NOTE_G3, NOTE_A3, NOTE_G3, 0, NOTE_B3, NOTE_C4
};
int noteDurations[] = {
  4, 8, 8, 4, 4, 4, 4, 4
};

```

```

void setup() {
  // abot.setup();
  for (int thisNote = 0; thisNote < 8; thisNote++) {
    int noteDuration = 1000 / noteDurations[thisNote];
    abot.tone(6, melody[thisNote], noteDuration);

    int pauseBetweenNotes = noteDuration * 1.30;
    delay(pauseBetweenNotes);

    noTone(6);
  }
}

void loop() {
}

```

The code is explained as follows:

(1) Including Libraries:

```

#include "SelfAbot.h"
#include "pitches.h"

```

SelfAbot.h: This contains the definition of the 'SelfAbot' class.

pitches.h: This file includes definitions for the frequencies of musical notes (e.g., NOTE_C4, NOTE_G3) used to play different tones.

(2) Creating an Object:

SelfAbot abot; This line creates an object (or instance) 'abot' of the 'SelfAbot' class. This object is used to access methods provided by this class, such as playing tones.

(3) Defining Melody and Duration:

```
int melody[] = { ... };
```

```
int noteDurations[] = { ... };
```

melody[]: An array containing the frequencies of the notes to be played.

noteDurations[]: An array containing the duration of each note in the melody. Durations are represented as fractions of a whole note (e.g., '4' for a quarter note, '8' for an eighth note).

(4) setup Function:

```

void setup() {
  for (int thisNote = 0; thisNote < 8; thisNote++) {
    ...
  }
}

```

This function is called once when the Arduino starts. It includes a 'for' loop that iterates over the notes of the melody. Each note is played using the 'abot.tone()' method, which generates sound on pin '6'.

(5) Playing the Melody:

```

int noteDuration = 1000 / noteDurations[thisNote];
abot.tone(6, melody[thisNote], noteDuration);

```

noteDuration: Calculates the duration of each note in milliseconds.

abot.tone(6, melody[thisNote], noteDuration): Plays the current note. The tone method uses three parameters: the pin number (6) where the buzzer is connected, the frequency of the note (melody[thisNote]), and the note's playtime (noteDuration).

(6) Pause Between Notes (pauseBetweenNotes variable):

```

int pauseBetweenNotes = noteDuration * 1.30;
delay(pauseBetweenNotes);
noTone(6);

```

The pause between notes is very important in expressing the melody. This variable represents the rest time between the sound notes. After playing a note, there is a short pause (pauseBetweenNotes) equivalent to 1.30 times the note duration, followed by the noTone function stopping the tone function on pin 6.

(7) The loop Function:

```
void loop() { }
```

In this case, since the melody is played only once in the 'setup' function, the 'loop' function is left empty.

2.2.4 Controlling an LED with the analogWrite() Function

After connecting an LED component to the Arduino hardware, the code below introduces how to control an LED using the analogWrite() function in a sketch uploaded to Arduino. The sketch creates a fading effect on an LED connected to pin 9 of the Arduino.

For the Arduino Uno, there are six pins capable of analog output (PWM), which are pins ~D3, ~D5, ~D6, ~D9, ~D10, ~D11. These are the pins marked with a tilde on the board.

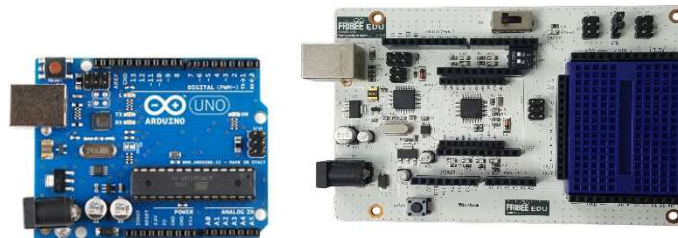


Figure 2.8: Arduino Uno Board and Fribee_Edu Uno Compatible Board

The 'Fribee Board (FRIBEE EDU)' used for Arduino robot practices uses the same setup method as the 'Arduino Uno'. It includes 3-pin ports and circuit functions that support additional servo motor operations and comes with built-in sockets for mounting XBee/Zigbee antennas, Bluetooth antennas, and WiFi antennas, allowing for easy wireless expansion.

The types of antennas suitable for wireless functionality expansion of the Fribee Board and how to use them will be explained again in Chapter 9.

Ex2.5_LED_fade_IN_OUT.ino

```
#include "SelfAbot.h"

SelfAbot abot;
int pin = 9;

void setup() {
  // abot.setup();
}
void loop() {
  for (int fadeValue = 0 ; fadeValue <= 255; fadeValue += 5) {
    abot.analogWrite(pin, fadeValue);
    delay(30);
  }
  for (int fadeValue = 255 ; fadeValue >= 0; fadeValue -= 5) {
    abot.analogWrite(pin, fadeValue);
    delay(30);
  }
}
```

(1) Including the SelfAbot Library:

```
#include "SelfAbot.h" This line includes the 'SelfAbot' library, a custom library for controlling various components of a robot or electronic system.
```

(2) Creating a SelfAbot Instance:

```
SelfAbot abot;
```

An instance (object) named abot of the 'SelfAbot' class is created. This object is used to access the functionalities provided by the 'SelfAbot' class.

(3) Pin Setup:

```
int pin = 9; Declares the integer variable 'pin' and initializes it with the value '9', which is the pin number on the Arduino board to which the LED (or other component) is connected.
```

(4) setup Function:

```
void setup() {  
  //abot.setup();  
}
```

The setup() function is executed once when the Arduino is powered on or reset. This code includes a commented-out line //abot.setup(); that can be used to initialize settings if necessary.

(5) loop Function:

```
void loop() {  
  // Code inside this function  
}
```

Unlike the setup(), the loop() function is executed repeatedly. It contains logic for implementing a fading effect. The logic for fading is explained below.

(6) Fading Logic:

The code uses two for loops to create a fading effect.

First for loop: Gradually increases brightness.

```
for (int fadeValue = 0; fadeValue <= 255; fadeValue += 5) {  
  abot.analogWrite(pin, fadeValue);  
  delay(30);  
}
```

This loop increments fadeValue by 5 from 0 to 255.

abot.analogWrite(pin, fadeValue): This line sends fadeValue to the pin, subtly expressing the brightness of the LED with an analog function. The range of analog values is from 0 (off) to 255 (maximum brightness).

delay(30): Pauses the loop for 30 milliseconds between each brightness change to create a noticeable fading effect.

Second for loop: Gradually decreases brightness.

```
for (int fadeValue = 255; fadeValue >= 0; fadeValue -= 5) {  
  abot.analogWrite(pin, fadeValue);  
  delay(30);  
}
```

This loop decreases fadeValue back down to 0 from 255.

The rest of the loop operates similarly to the first loop but in reverse, gradually dimming the LED's brightness.

2.3 Simplification through Function Overloading

2.3.1 Turning an LED On and Off with the digitalWrite() Function

The code below is an Arduino coding example that passes the pin value as an instance variable to the abot. This use of code is called function overloading. Using function overloading allows for clearer transmission of Arduino pin information.

Recall the practice from Section 2.2.1 where we used the digitalWrite() function without overloading to turn an LED on and off in the .ino example Ex2.1_LED_ON_OFF.ino. Then, re-upload the example below and practice achieving the same effect.

Ex2.6 LED_ON_OFF_pin_overloading.ino (overloaded type)

| overloaded | non-overloaded |
|---|---|
| <pre>#include "SelfAbot.h" SelfAbot abot; int pin = 9; void setup() { // abot.setup(); } void loop() { abot(pin).digitalWrite(HIGH); delay(1000); abot(pin).digitalWrite(LOW); delay(1000); }</pre> | <pre>#include "SelfAbot.h" SelfAbot abot; int pin = 9; void setup() { // abot.setup(); } void loop() { abot.digitalWrite(pin, HIGH); delay(1000); abot.digitalWrite(pin, LOW); delay(1000); }</pre> |

From now on, we will show how to use the 'SelfAbot' class with the overloaded operator() method. To explain the usage of the overloaded code simply:

abot(pin).digitalWrite(HIGH); This line is where the function is overloaded. The abot instance object is used together with the parentheses abot(pin), which calls the operator() method of the 'SelfAbot' class. This method is designed to accept a pin number and return a reference to the 'SelfAbot' object. Then, the digitalWrite(HIGH) function is called on this object, setting the specified pin (in this case, pin 9) to HIGH (typically 5V), turning on the LED or other devices connected to that pin.

abot(pin).digitalWrite(LOW); Similar to the previous line, but sets the pin to LOW (0V), turning off any connected device.

(Note) -----

The reason for using **function overloading** is that it allows for the explicit passage of values that can be passed as instance variables. Furthermore, by using the same function name, you can create multiple versions of a function that allow for different types based on the number of parameters or the data types of the parameters. This provides more flexibility in how functions are used, making the code more versatile and easier to read.

2.3.2 Practicing Button Circuit with Function Overloading

A button circuit is a method of transmitting external switch signals to a microcontroller like Arduino. Especially when using buttons with microcontrollers like Arduino, it is common to use pull-up or pull-down resistors to ensure a stable button state (HIGH or LOW). Both configurations have their unique characteristics and are used to prevent "floating" states of the digital input pins when the button is not pressed.

Compare the pull-up and pull-down button circuits below in the table. Pull-up and pull-down resistors are used to ensure a stable and predictable state for the digital input pin of the button circuit, preventing unpredictable readings caused by floating states. Remember how digital pin 2 (D2) is connected in the circuit diagrams below.

Table 2.1 Two types of electronic circuits for configuring button circuits

| | pull-up circuit | pull-down circuit |
|-------------------------|--|--|
| Circuit diagram | | |
| Circuit characteristics | <p>In a pull-up resistor configuration, the input pin is typically pulled to a HIGH state (connected to Vcc). When the button is pressed, the circuit is connected to ground (GND), causing the input pin to become LOW.</p> | <p>In a pull-down resistor configuration, the input pin is typically pulled to a LOW state (connected to GND). When the button is pressed, the circuit is connected to positive voltage (Vcc), causing the input pin to become HIGH.</p> |

Ex2.7 Button_LED_ON_OFF_pin_overloading.ino

```
#include "SelfAbot.h"

SelfAbot abot;
const int buttonPin = 8;
const int ledPin = 9;
int buttonState = 0;

void setup() {
  // abot.setup();
}
void loop() {
  buttonState = abot(buttonPin).digitalRead();
  if (buttonState == HIGH) {
    abot(ledPin).digitalWrite(HIGH);
  } else {
    abot(ledPin).digitalWrite(LOW);
  }
}
```

The code within the loop function is explained as follows:

`buttonState = abot(buttonPin).digitalRead();` Executes the `digitalRead` function to read the state value of the button pin. This function is a method of the 'SelfAbot' class. It uses the function overloading of the `abot` instance to pass the pin information, `buttonPin` value.

Then, when executing an `if` statement, if the `buttonState` value is `HIGH`, it executes `abot(ledPin).digitalWrite(HIGH);` if it is `LOW`, it executes `abot(ledPin).digitalWrite(LOW);`. For example, `abot(ledPin).digitalWrite(HIGH);` uses function overloading to pass the `ledPin` information and turns on the LED.

2.3.3 Analog Input

Another practice example is a code that expresses the change in LED brightness as an analog PWM output according to the size of the analog input value when the input voltage is changed with a variable resistor. There is no method defined in the 'SelfAbot' class for processing analog input signals. Therefore, standard Arduino coding can be used.

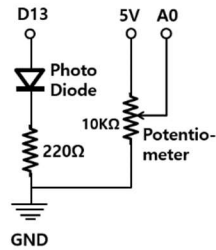


Figure 2.9: Analog Input Circuit

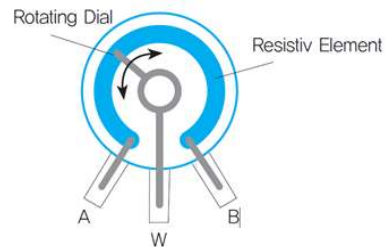


Figure 2.10: Variable Resistor

The A and B pins of the variable resistor are connected to the total resistance, and turning the handle knob of the variable resistor changes the resistance value connected to the W pin, A-W and W-B resistance.

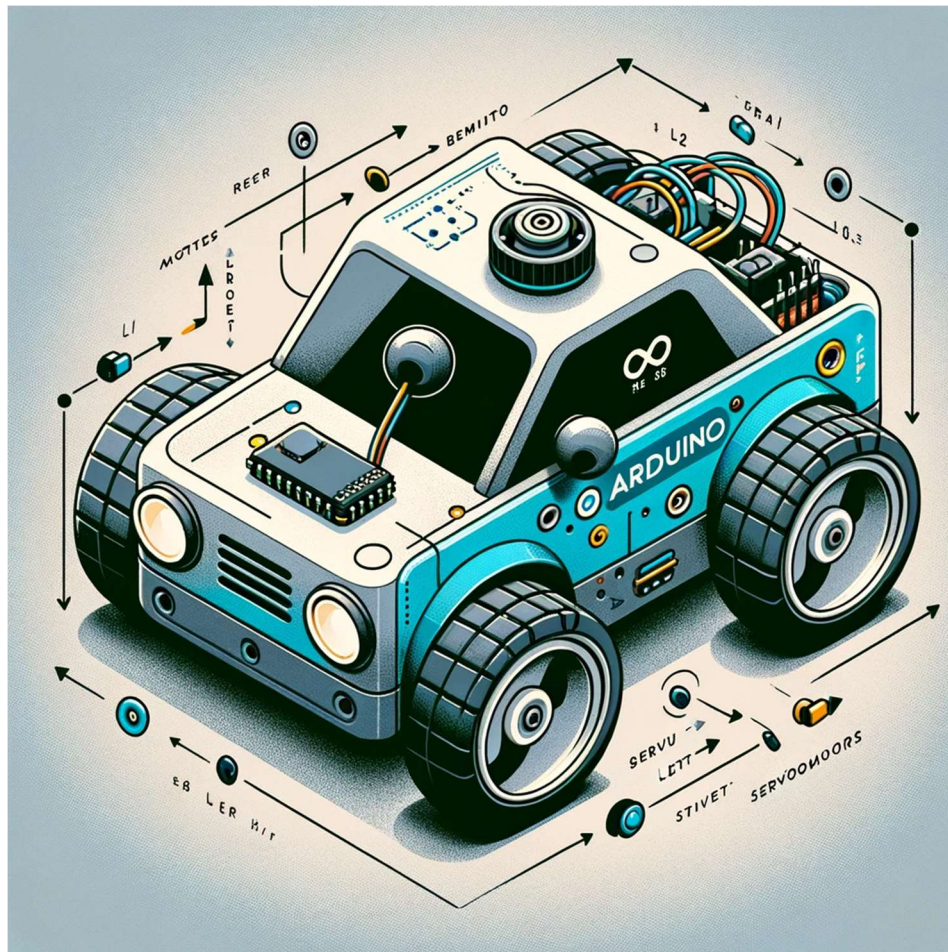
Ex2.8 Potentiometer_analog_LED.ino

```
#include "SelfAbot.h"

SelfAbot abot;
int pin = 9;
int sensorPin = A0;
int sensorValue = 0;
void setup() {
  // abot.setup();
}
void loop() {
  sensorValue = analogRead(sensorPin)/4;
  abot.analogWrite(pin, sensorValue);
}
```

Chapter 3 Robot Initialization and Basic Action Methods

- 3.1 Implementing the setup Method for Robot Initialization
- 3.2 Robot Movement Properties and Methods
- 3.3 Simplification through Function Overloading
- 3.4 Standard Servo Motor Basic Actions
- 3.5 Continuous Rotation Servo Motor Basic Actions



The 'neo-Arduino car' has been used so far for practicing C language coding. Many users feel that learning C language alone is insufficient, and thus, they are looking for ways to utilize Arduino robots using C++ language.

In this chapter, we will explain the methods of the 'SelfAbot' class and examine how each method is specifically connected and used in robot actions. Please refer to Section 1.4 for a brief introduction to the header file (.h) and source file (.cpp) of the 'SelfAbot' class used in the code below.

3.1 Implementing the setup Method for Robot Initialization

The Arduino `setup()` method is responsible for the robot's initial settings. This function checks if the servo motor pins are valid and, if so, connects the servo motors to those pins. Initialization is essential for the proper operation of the robot. The initialization process involves setting up the servo motors, pin configurations, and other fundamental settings of the robot.

Here's how to use the `setup()` function to initialize the robot:

(1) Servo motor pin configuration: Specify the pin numbers to which the robot's servo motors will be connected. This is usually done when creating the robot object.

For example, you can pass the pin numbers for the left and right servo motors when creating the robot object, like so: `SelfAbot abot(LEFT_SERVO_PIN, RIGHT_SERVO_PIN)`. This method creates an instance object with parameters, thereby passing the servo motors' pin numbers.

(2) Calling the `setup()` method: The `setup()` method actually connects the robot's servo motors to the pins and performs additional initialization tasks if necessary. Call the robot object's `setup()` method within the main `setup()` function of Arduino. Thus, by adding the necessary initialization code in the `setup()` method rather than in the constructor, you can set the robot's initial conditions.

```
void setup() {  
    abot.setup(); // Calls the robot's setup method  
}
```

(3) Necessary additional initialization: Depending on the robot, additional initialization tasks may be required. For example, initializing sensors or setting certain pins as input or output falls under this category.

The `setup()` function is called only once when the Arduino program starts and is used to perform the basic settings of the robot. Through this process, the robot is prepared to execute various commands that will be run later.



Figure 3.1: Neo-Arduino Robot Car

3.2 Robot Movement Properties and Methods

In the 'SelfAbot' class, we introduce member variables and functions for moving the servo motors of an Arduino robot. Let's look at each of their roles.

Member Variables

```
Servo _servoAngle, _servoLeft, _servoRight;
```

These are objects of the 'Servo' class from the Arduino Servo library. Each object controls a specific servo motor. The `_servoAngle` variable is a member variable for moving a standard servo motor, and the other two, `_servoLeft` and `_servoRight`, are member variables for moving continuous rotation servo motors connected to the wheels on both sides of the robot.

```
byte _servoAnglePin, _servoLeftPin, _servoRightPin;
```

Variables that store the pin numbers to which each servo motor is connected. They store the pin numbers connected to the standard servo motor and continuous rotation servo motors.

byte _selectedPin;

This variable is intended to store the pin number of a specific servo motor or other component currently being controlled or interacted with.

Use in Overloaded Functions:

Using _selectedPin in overloaded member functions allows the use of the same function name for operations on different components by internally referencing the pin number stored in _selectedPin. This eliminates the need to explicitly pass the pin number each time the function is called.

Single Instance Variable Operation:

By setting _selectedPin to the pin number of the desired component, subsequent operations can be performed on that specific component. This is particularly useful in classes managing multiple components (e.g., multiple servo motors).

Member Functions

The two member functions servoAttachAngle(byte ServoAnglePin) and servoAttachPins(byte ServoLeftPin, byte ServoRightPin) are written to use the original function form without using overloading. The remaining servo motor operation-related member functions (or methods) are written to use code that employs overloading.

servoAttachAngle(byte ServoAnglePin);

Connects the servo used for angle control to the specified pin (servoAnglePin). This function is for explicitly passing the pin value of the standard servo motor. Although this function does not use overloading, additional code is needed to employ overloading.

servoAttachPins(byte ServoLeftPin, byte ServoRightPin);

Connects the left and right servo motors to their respective pins. The servoAttachPins() function explicitly passes the pin values of the left and right servo motors as two parameter values. This function does not use overloading. To use overloading, a separate method must be additionally written.

Ex3.1_nonoverloaded_servo_speed.ino

```
#include "SelfAbot.h"

SelfAbot abot;

void setup() {
  abot.setup();
  abot.servoAttachPins(13, 12);

  abot.servoSpeedNonoverload(150, -150);
  delay(2000);
}

void loop() {
}
```

In this code example, we use the method servoSpeedNonoverload() which does not employ function overloading.

First, we will explain the operating principle from the library code in an easy-to-understand manner.

(1) Method: SelfAbot::setup()

```
void SelfAbot::setup() {
  if (_servoLeftPin != INVALID_PIN && _servoRightPin != INVALID_PIN) {
    _servoLeft.attach(_servoLeftPin);
```

```

    _servoRight.attach(_servoRightPin);
  }
}

```

This function initializes the servo motors mounted on the robot.

It checks if the pins for the left and right servos (`_servoLeftPin` and `_servoRightPin`) are valid (by checking if they are not equal to the already defined constant `INVALID_PIN`, which represents an incorrect or unused pin).

If the pins are valid, it attaches the servo objects (`_servoLeft` and `_servoRight`) to the respective pins. Here, "attaching" means passing the pin connected to the servo motor to the servo library for control.

(2) Method: `SelfAbot::servoAttachPins(byte ServoLeftPin, byte ServoRightPin)`

```

void SelfAbot::servoAttachPins(byte servoLeftPin, byte servoRightPin) {
  if (servoLeftPin != INVALID_PIN) {
    _servoLeftPin = servoLeftPin;
    _servoLeft.attach(_servoLeftPin);
  }
  if (servoRightPin != INVALID_PIN) {
    _servoRightPin = servoRightPin;
    _servoRight.attach(_servoRightPin);
  }
}

```

This code does not use overloaded functions; instead, it uses the pin values for the left and right servos as parameters to store them in the variables corresponding to the two wheels' servo motors, and runs the `attach()` function of the servo motor to prepare the servo motors for operation.

This feature is used to specify which pins the left and right servo motors are connected to.

It individually checks each pin. If the pin is not `INVALID_PIN`, it stores the new pin number and attaches the left or right servo to the left or right pin.

(3) Method: `SelfAbot::servoSpeedNonoverload(int leftSpeed, int rightSpeed)`

```

void SelfAbot::servoSpeedNonoverload(int leftSpeed, int rightSpeed) {
  _servoLeft.writeMicroseconds(1500 + leftSpeed);
  _servoRight.writeMicroseconds(1500 + rightSpeed);
}

```

This feature sets the speed for the left and right servo motors.

It uses the 'writeMicroseconds' method, a precise way to control servo motors. '1500' is the neutral point of the servo, and adding a speed variable allows the servo to rotate clockwise or counterclockwise. The method `servoSpeedNonoverload()` for operating the servo motor does not use additional servo pin information. Instead, the servo pin information is passed through the `servoAttachPins()` method, and the same servo pin information is implicitly transmitted and applied by being used in the `servoSpeedNonoverload()` method.

Below is a detailed explanation of each line of the Arduino .ino sketch code introduced above.

(1) Including the SelfAbot Library

```
#include "SelfAbot.h"
```

This code includes the SelfAbot library, declaring that the SelfAbot class and its functionalities can be used.

(2) Robot Setup

```
SelfAbot abot;
void setup() {
```

```

abot.setup();
abot.servoAttachPins(13, 12);
abot.servoSpeed(150, -150);
delay(2000);
}

```

The code line `SelfAbot abot;` creates a `SelfAbot` object named `abot`. The `setup()` function runs once when Arduino is reset or powered on.

The `abot.setup()` method initializes the servo motors. `abot.servoAttachPins(13, 12)` sets the pins for the left and right servos to 13 and 12, respectively. `abot.servoSpeed(150, 150)` sets the speed for the left and right servos. `delay(2000)` pauses the code for 2000 milliseconds (or 2 seconds). The `loop()` function is empty. This means that Arduino does not perform any repetitive code execution tasks at the moment.

(Note) -----

Preparatory work to operate the 'neo-Arduino robot car' servo motors

- (1) Upload the Arduino sketch to the board using the IDE program.
- (2) Connect all the servo motor connection pins to the Arduino board. Ensure to connect the white, red, and black wires in the correct orientation. (White: Signal, Red: 5V Power, Black: Ground, 0V)
- (3) If power is supplied via a USB cable, there is no need to prepare separate AA batteries. However, if you wish to test the operation of the servo motors after removing the USB cable, you must install 5 AA batteries and connect the power jack to the board.

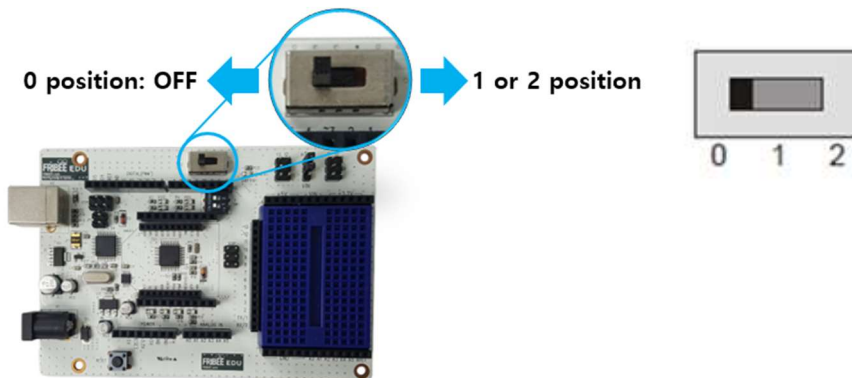


Figure 3.2: The 3-Point Switch on the Arduino Robot Board (Fribee-Edu Board)

- (4) The 3-point switch on the Arduino board must be switched in the direction of the arrow shown below. The 3-point switch is differentiated into 0, 1, 2 positions; the '0' position cuts off the power from the AA batteries, the '1' position supplies power for the operation of the Arduino program only, and the '2' position supplies power for the operation of the servo motors as well.

If the robot behaves very abnormally when power is supplied to the servo, you may need to first perform the 'Centering the Servo' in Section 4.1, and then rerun the example code.

3.3 Simplification through Function Overloading

Overloaded methods can be used to make the functionalities of the robot more concise and flexible. The Arduino robot methods below are all written using code that employs function overloading.

servoAngle(int angle);

This function sets the servo angle, positioning a standard servo motor used for angle control within a range of 0 ~ 180 degrees to a specific angle. If the condition in the conditional statement is true, it executes the `_servoAngle.write(angle)` method. The `write(angle)` method is an Arduino function for operating a standard servo. In this case, the servo instance representing `_servoAngle` is passed as an overloaded value. Therefore, only the angle value needs to be passed as the parameter value for the servo motor method.

```
void SelfAbot::servoAngle(int angle) {
    if (angle >= 0 && angle <= 180) {
        _servoAngle.write(angle);
    } else {
        Serial.println("Error: Invalid input value");
    }
}
```

servoSpeed(int speed);

The speed of the currently selected servo motor is set. This method allows you to adjust the servo to a specific speed. The `servoSpeed(int speed)` method uses the `writeMicroseconds()` function of the Arduino servo class to operate one continuous rotation servo motor. This method uses instance variables to pass the corresponding pin information. Therefore, only the speed value for the servo motor connected to the corresponding pin is passed as a parameter.

One consideration is to determine whether the pin in use is for the left wheel or the right wheel and to ensure that the servo motor operates according to the intent of the code. In the case of an overloaded instance variable with one parameter, `_selectedPin` is used for transmission.

Thus, the value of `_selectedPin` is determined by a conditional statement to see whether it is the instance value of the left wheel stored by the user through the execution of the `servoAttachPins(byte servoLeftPin, byte servoRightPin)` method or the instance value of the right wheel, performing the operation in accordance with the user's code intent. The method used in actual code examples will be explained again in Section 3.5.

```
void SelfAbot::servoSpeed(int speed) {
    if (_selectedPin == _servoLeftPin) {
        _servoLeft.writeMicroseconds(1500 + speed);
    } else if (_selectedPin == _servoRightPin) {
        _servoRight.writeMicroseconds(1500 + speed);
    }
}
```

servoSpeed(int leftSpeed, int rightSpeed);

Independently sets the speed of the left and right servo motors. The setting function for the operation of both wheels allows independent control of the speed for the left and right servos.

The `servoSpeed(int leftSpeed, int rightSpeed)` method uses the `writeMicroseconds()` function of the Arduino servo class to operate two continuous rotation servo motors simultaneously. This method uses the `leftSpeed` and `rightSpeed` values to operate the left and right servo motors, respectively.

The pin information for operating the left and right wheels is passed as instance values using the function overload feature. The method used in actual code examples will be explained again in Section 3.5.

```
void SelfAbot::servoSpeed(int leftSpeed, int rightSpeed) {
    _servoLeft.writeMicroseconds(1500 + leftSpeed);
```



```
    _servoRight.writeMicroseconds(1500 + rightSpeed);  
}
```

detachServos();

This code detaches all servo motors from the pins to prevent control and also reduces power consumption. Calling the method `detachServos()` disconnects the connections of the servo motors assigned to the left and right wheels. If you want to detach only a specific wheel using the function overload feature, additional code is required.

```
void SelfAbot::detachServos() {  
    _servoLeft.detach();  
    _servoRight.detach();  
}
```

These methods are used to precisely control the robot's servo motors and perform a variety of movements and actions. This design facilitates a modular approach that allows each motor to be controlled independently or in conjunction with other motors, depending on the requirements of the robot.

3.4 Standard Servo Motor Basic Actions

Examine the differences between overloaded methods and the original methods, and see how overloaded methods are used in robot actions. Using overloaded methods has the effect of explicitly passing the pin information of the servo motors that contribute to robot actions.



Figure 3.3: Parallax Standard Servo Motor

Upload the below `Arduino Ex3.2_standard_Servo_control.ino` file and observe how the standard servo motor operates. This example introduces both primitive method forms without overloading and methods with function overloading applied.

To use a servo motor for the first time, you must set the servo motor to an electrical ready state. In Arduino, this action is performed with the command `servo.attach()`.

For the same purpose, the code below `abot.servoAttachAngle(servoPin)` is intended to create the electrical ready state for a standard servo motor. By explicitly passing the pin number as a parameter in the initial setup of the servo motor, it has the effect of making the servo motor pin information written by the user in the code recognizable.

The method `abot(servoPin).servoAngle(pos)` to create movement for the standard servo motor uses the `abot` instance variable to pass the pin information 10, and only the angle value, which represents the actual size of the movement, is passed as a parameter.

Method without overloading:

```
abot.servoAttachAngle(servoPin);
```

Method with overloading:

```
abot(servoPin).servoAngle(pos);
```

Ex3.2_standard_Servo_control.ino

```
#include "SelfAbot.h"
const byte servoPin = 10;
SelfAbot abot;

void setup() {
  abot.setup();
  abot.servoAttachAngle(servoPin);
}

void loop() {
  for (int pos = 0; pos < 180; pos += 1) {
    abot(servoPin).servoAngle(pos);
    delay(15);
  }
  for (int pos = 180; pos >= 1; pos -= 1) {
    abot(servoPin).servoAngle(pos);
    delay(15);
  }
}
```

3.5 Continuous Rotation Servo Motor Basic Actions

The robot's two wheels use methods to create continuous rotation servo motor movements. One method is `abot(13).servoSpeed(0)`, and the other method is `abot(13,12).servoSpeed(-150,150)`. The two methods differ in that one uses a single parameter while the other uses two parameters. Both types of methods use function overloading to pass the servo motor's pin information as instance variables, in addition to the method's parameters.



Figure 3.4: Parallax Continuous Rotation Servo Motor

Just like with standard servo motors, continuous rotation servo motors also require initial setup. To set up the continuous rotation servo motors connected to both wheels, we first use the method `servoAttachPins(byte servoLeftPin, byte servoRightPin)` that does not use function overloading. This code execution sets the pin information corresponding to the left and right wheels as determined by the user. In the method, these values are stored in instance variables corresponding to the left wheel, and simultaneously, the initial setup function `servo.attach()` for the servo motor of the left wheel is executed. The right wheel is set up in the same manner. Let's examine the example code below.

Below detailed explanation of the example code is focused on using the 'SelfAbot' class to control a robot, with a particular focus on servo motor operation.

(1) Including the 'SelfAbot' Library:
`#include "SelfAbot.h"`

This includes the SelfAbot library, granting access to the 'SelfAbot' class and its functionalities.

(2) Setting Constants for Servo Pins:
`const byte servoLeftPin = 255;`

```
const byte servoRightPin = 255;
```

Ex3.3_continuous_Servo_speed.ino

```
#include "SelfAbot.h"

const byte servoLeftPin = 255;
const byte servoRightPin = 255;
SelfAbot abot(servoLeftPin, servoRightPin);

void setup() {
  abot.setup();
  abot.servoAttachPins(13, 12);
}

void loop() {
  abot(13).servoSpeed(0);
  abot(12).servoSpeed(0);
  delay(1000);

  abot(13).servoSpeed(150);
  abot(12).servoSpeed(-150);
  delay(1000);

  abot(13, 12).servoSpeed(-150, 150);
  delay(1000);
}
```

Here, two constants, `servoLeftPin` and `servoRightPin`, are defined with the value 255. The value 255 is used as a temporary placeholder to prevent uninitialized states since it's not an actual pin value on the Arduino.

(3) Creating an Instance of 'SelfAbot':

```
SelfAbot abot(servoLeftPin, servoRightPin);
```

This code creates an object, `abot`, of the 'SelfAbot' class, initializing it with the left and right servo pins. Since the initial value of 255 is used, the actual pin numbers are set in the code later.

(4) The `setup()` Function:

```
void setup() {
  abot.setup();
  abot.servoAttachPins(13, 12);
}
```

`setup()` is a function called once by the Arduino when the program starts, used for initializing settings. `abot.setup()` calls the `setup` method for the `abot` object, which initializes the servo. `abot.servoAttachPins(13, 12)` connects the servos to digital pins 13 and 12 on the Arduino board.

※ **Are you curious if the `abot` instance uses a default constructor or a constructor with two parameters? Let's take a closer look.**

The `abot` instance is created with the code `SelfAbot abot(servoLeftPin, servoRightPin);` using a two-parameter constructor of the 'SelfAbot' class. This constructor uses two arguments, 'servoLeftPin' and 'servoRightPin'. Despite both parameters being initialized to 255, it calls the constructor expecting two-byte values as parameters.

Later, the code `abot.servoAttachPins(13, 12);` calls a method of the `SelfAbot` class on the already created `abot` instance. It does not create a new instance but uses the already created 'abot' instance to connect the servos to digital pins 13 and 12 on the Arduino board.

The next code, `abot(13).servoSpeed(0);`, also does not create a new instance of the 'SelfAbot' class but modifies the state of the already created `abot` instance. The `abot(13)` syntax utilizes the `operator()` function of the 'SelfAbot' class. This operator is overloaded in the 'SelfAbot' class to set the `_selectedPin` member variable to the value passed as an argument. In this case, it sets `_selectedPin` to 13.

After this operator call, `servoSpeed(0)` is called on the same `abot` instance, operating based on the just-set value of `_selectedPin`, which is now 13. This pattern allows for a flexible interface that enables chaining method calls on the same object in a readable and expressive manner.

To clarify, `abot(13).servoSpeed(0);` performs the following steps:

Calls `operator()(13)` on the `abot` instance, setting `_selectedPin` to 13.

Then, calls the `servoSpeed(0)` method on the `abot` instance.

This method operates based on the value of `_selectedPin`, which is currently 13.

(5) The `loop()` Function:

```
void loop() {
  abot(13).servoSpeed(0);
  abot(12).servoSpeed(0);
  delay(1000);

  abot(13).servoSpeed(150);
  abot(12).servoSpeed(-150);
  delay(1000);

  abot(13, 12).servoSpeed(-150, 150);
  delay(1000);
}
```

The code within the `loop()` function operates as follows:

`abot(13).servoSpeed(0);` and `abot(12).servoSpeed(0);` stop the servos connected to pins 13 and 12, respectively. A servo motor speed value of 0 represents a stopped state.

`delay(1000);` pauses the program for 1000 milliseconds (1 second).

Subsequently, the servos are set to different speeds, for example, 150 or -150, causing the servo motors to rotate clockwise or counterclockwise by the specified amount.

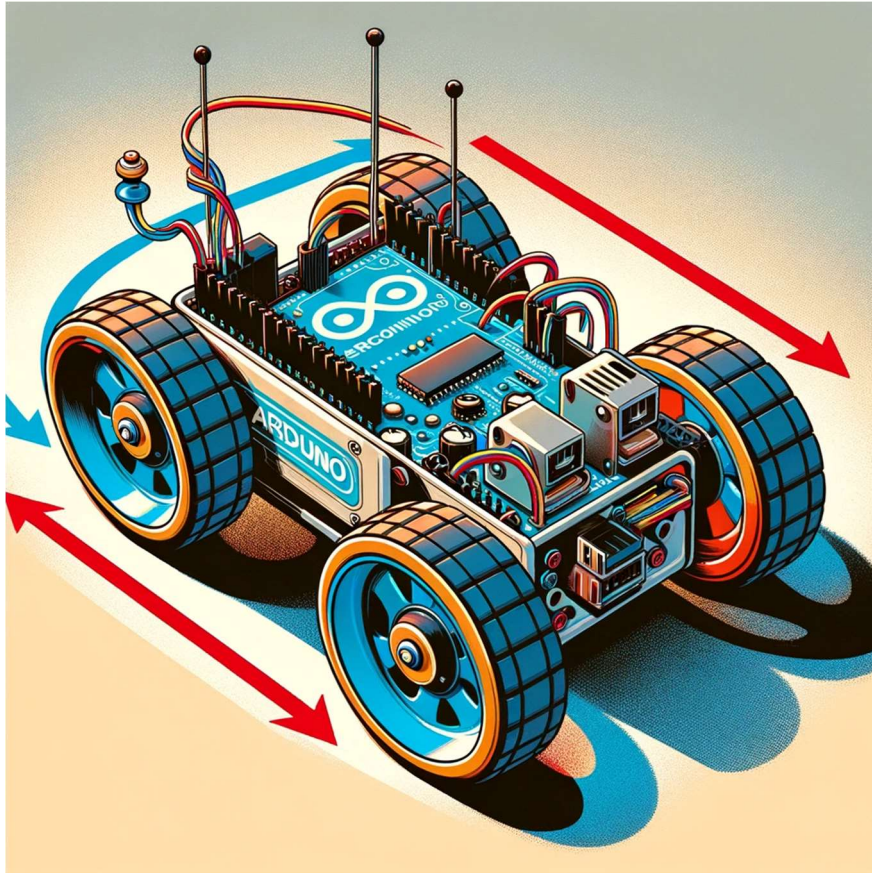
`abot(13, 12).servoSpeed(-150, 150);` sets different speeds for the two servos simultaneously, allowing the robot to rotate or move in a specific pattern. The first parameter value controls the speed of the wheel connected to the left servo motor, and the second parameter value controls the speed of the wheel connected to the right servo motor. This code essentially provides both parameters simultaneously to control the two servo motors, managing the two wheels' stop, start, and direction change actions.

Another 1-second delay loop repeats.

The 'SelfAbot' class offers an organized way to control robots using function overloading and easy-to-understand commands.

Chapter 4 Controlling the Movement of the Arduino Robot

- 4.1 Centering the Servo Motor
- 4.2 Straight-line Travel and Robot Correction
- 4.3 Left Turn / Right Turn and Continuous Actions
- 4.4 Gradual Acceleration and Deceleration
- 4.5 Additional Method: maneuver()



Until now, we focused on explaining the methods within the 'SelfAbot' class. Starting from Chapter 4, we will focus on how to use the 'SelfAbot' class to operate Arduino robots.

Users who have handled Arduino robots with C language will be able to compare the practices more familiarly. For users encountering this material for the first time, we aim to explain as simply and concisely as possible for easy understanding. If you feel the explanations are insufficient, reading the 'Arduino Robot Practice Guide' written in C language or watching public YouTube videos is recommended.

4.1 Centering the Servo Motor

Connect the continuous rotation servo motor to the Fribee board by referring to the diagram below, Figure 4.1. When connecting the servo pins to the Fribee board, insert them with the black wire facing towards the breadboard. You can use pins 13 and 12, or pins 11 and 10.

When power is first supplied to the Arduino board, a flashing signal is detected from LED pin 13 as the bootloader starts to operate (a default behavior of the Arduino board). Therefore, every time a reset signal is used on the robot, there is a tendency for the robot to move briefly. To avoid this issue, using servo motor pins 11 and 10 can resolve the problem.

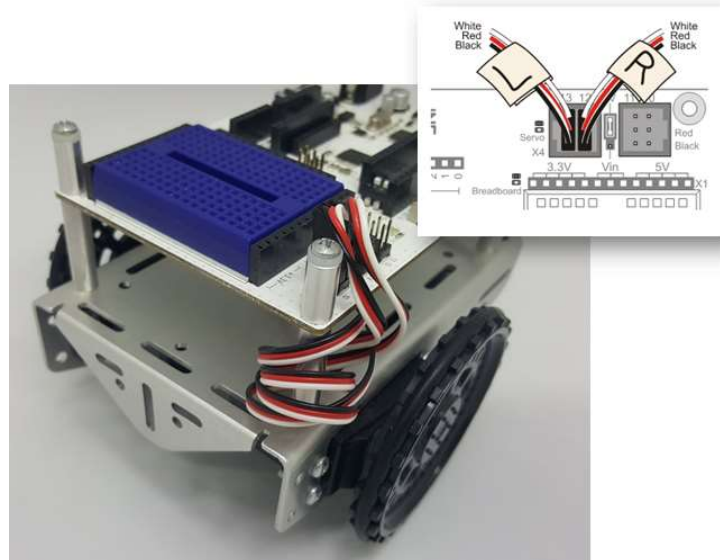


Figure 4.1: How to Connect Servo Pins on an Arduino Robot

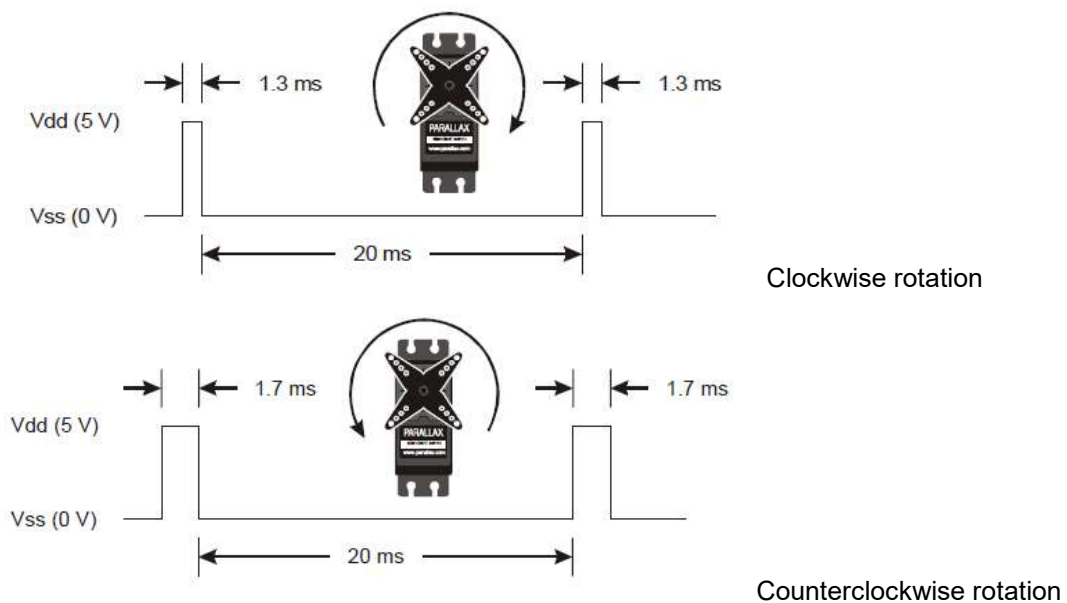


Figure 4.2: PWM modulation control method for the Parallax continuous rotation servo (Source: parallax.com)

(Note) -----

Understanding the Basic Operation of Servo Motors for Robot Locomotion

To attempt the servo wheel rotation for forward motion of the 'Arduino robot', the right wheel must rotate

clockwise and the left wheel counter-clockwise when viewed from the direction of travel.

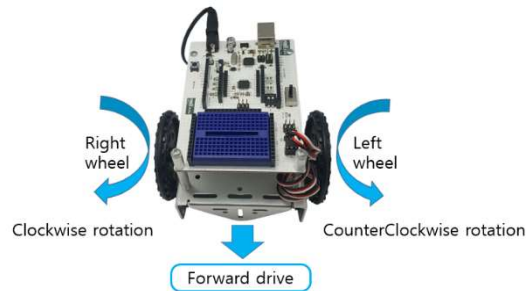


Figure 4.3: Forward Motion Servo Motor Rotation Direction

Servo motor operation uses Pulse Width Modulation (PWM) control within a fixed cycle. The rotation of the servo motor, either clockwise or counter-clockwise, varies from the center value '0' of the pulse width in PWM control. In other words, the direction and magnitude of the servo motor's rotation are controlled by changing the active pulse width of the PWM signal. If the pulse within the standard range is shorter or longer, it precisely controls the amount of rotation from the neutral position in either clockwise or counter-clockwise direction.

When servo motors are shipped from the factory for use, especially in robotics where precise control is needed, it is necessary to physically calibrate the servo's movement. Servo motors incorporate a variable resistor for this purpose of physical calibration. Users can find and align the physical midpoint by adjusting a screw.



Figure 4.4: Adjusting the screw for servo motor centering (Source: parallax.com)

Ex4.1_servo_centering.ino

```
#include "SelfAbot.h"
#define INVALID_PIN 255;

const byte servoLeftPin = INVALID_PIN;
const byte servoRightPin = INVALID_PIN;

SelfAbot abot(servoLeftPin, servoRightPin);

void setup() {
  abot.setup();
  abot.servoAttachPins(13, 12);
}

void loop() {
  abot(13).servoSpeed(0);
  abot(12).servoSpeed(0);
  delay(1000);

  abot(13, 12).servoSpeed(0, 0);
```

```

delay(1000);
}

```

Centering the servo motor is introduced with example code. For Parallax's continuous rotation servo motors, a value of '1500' corresponds to the center position, and in the 'SelfAbot' class's servo method, a speed value of '0' represents the center position. Uploading the code below to the Arduino robot should result in the wheels being in a stationary state. If there is slight movement, adjust the screw to eliminate this minor movement. This process is referred to as 'centering the servo motor.'

Upload the code below to ensure the wheels do not move and to perform the servo motor centering task. In the conditions of the practice code, the servo motors are connected to pins 13 and 12.

4.2 Straight-line Travel and Robot Correction

We introduce a graph showing how the speed of the Parallax continuous rotation servo motor can be varied.

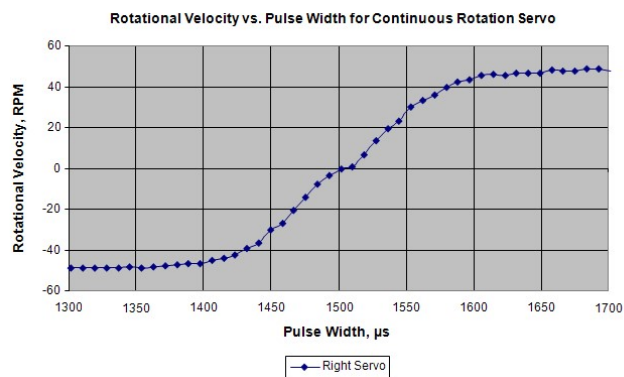


Figure 4.5: Changes in Pulse Width for Speed of Parallax Continuous Rotation Servo Motor (Source: parallax.com)

The range of speed values can be adjusted from -200 to +200 based on a reference of 1500. As the table below demonstrates, the effective speed change mostly occurs within the range of -100 to +100. Furthermore, a speed value of -40 or +40 nearly represents a mid-speed.

In robotics, using two servo motors for straight-line robot motion can result in performance discrepancies due to manufacturing differences, wear, or mechanical characteristics, even when the same speed values are supplied. To address this issue and achieve more precise straight-line movement, a method to adjust each motor's speed for correction can be implemented.

This example introduces code design for fine-tuning the rotation speed of two servos, considering mechanical inconsistencies. Identifying performance differences can involve measuring RPM or assessing deviation from a straight line over a set distance. Such adjustments may need to be repeated due to battery wear or load changes, and should be done using only AA battery power, without USB power, for accuracy.

Real-time accurate motion tracking can be achieved with encoders or similar sensors. Even without encoders, calibrating in consistent external conditions, like surface texture, can improve straight-line accuracy.

Below is a correction code using the abot instance that incorporates normalization and variance concepts.

```

float deviationFactor = 0.0;
void driveStraight(int leftSpeed, int rightSpeed) {
    // Apply normalization to calculate correction coefficients

```



```

float leftCoefficient = 1.0 + deviationFactor; // Increase for left servo
float rightCoefficient = 1.0 - deviationFactor; // Decrease for right servo

int adjustedLeftSpeed = leftSpeed * leftCoefficient;
int adjustedRightSpeed = rightSpeed * rightCoefficient;

  abot.servoSpeed(adjustedLeftSpeed, adjustedRightSpeed);
}

```

The deviationFactor is a single floating-point number between -1.0 and 1.0. A value of 0 indicates no deviation (both servos run at the same speed), positive values increase the speed of the left servo and decrease the speed of the right servo, and negative values do the opposite.

In the driveStraight method, the deviation factor adjusts each servo's speed for normalized speed adjustment. The speed of the left servo increases with a positive deviation, and the speed of the right servo decreases, and vice versa.

You're encouraged to add the code introduced here to your 'SelfAbot' library or Arduino .ino file for testing. We will revisit the method of implementing precise wheel action correction codes through class inheritance later.

First, follow the straight-line driving practice as shown in the figure below. The concept involves aligning one wheel of the robot with a ruler and positioning both wheels so they are perpendicular to the ruler. For correction, after the Arduino robot has moved forward for a certain time (delay(2000) in the code below), check the distance (deviationDistance) by which the wheel aligned with the ruler has deviated. Repeat the correction practice until this distance is minimized. Adjust the deviationFactor value between -0.15 and +0.15, remembering to use decimal values as it's a float variable.

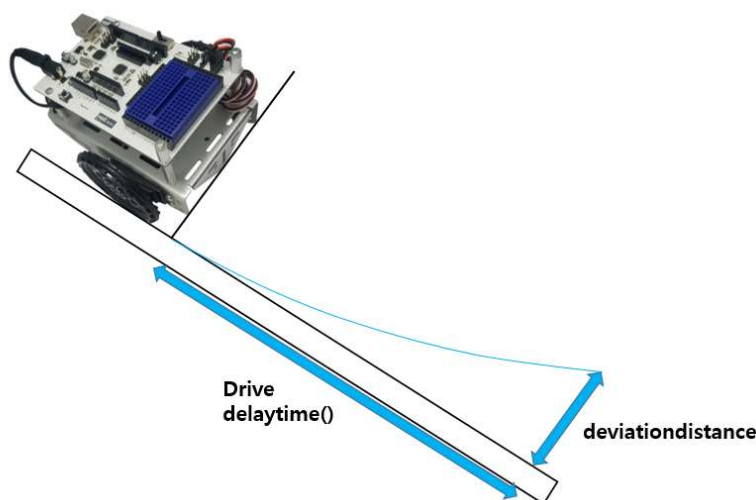


Figure 4.6: Conceptual Diagram for Correction Practice for Straight-Line Driving of Arduino Robot

Ex4_2_abot_driveStraight.ino

```

#include "SelfAbot.h"
#define INVALID_PIN 255;

const byte servoLeftPin = INVALID_PIN;
const byte servoRightPin = INVALID_PIN;
float deviationFactor = 0.0; // min. val -0.15 max. val +0.15

SelfAbot abot(servoLeftPin, servoRightPin);

void setup() {

```

```

    abot.setup();
    abot.servoAttachPins(11, 10);
    driveStraight(40, -40);
    delay(8000);
    abot.servoSpeed(0, 0);
    delay(1000);
}

void loop() {

}

void driveStraight(int leftSpeed, int rightSpeed) {
    // Apply normalization to calculate correction coefficients
    float leftCoefficient = 1.0 + deviationFactor; // Increase for left servo
    float rightCoefficient = 1.0 - deviationFactor; // Decrease for right servo

    int adjustedLeftSpeed = leftSpeed * leftCoefficient;
    int adjustedRightSpeed = rightSpeed * rightCoefficient;

    abot.servoSpeed(adjustedLeftSpeed, adjustedRightSpeed);
}

```

This code is an example for controlling a robot based on Arduino (using the 'SelfAbot' class here). The aim of the code is to make the robot with two servo motors drive straight, even when the physical operating characteristics of the two servos are not identical, by applying fine corrections. Users need to iteratively adjust the 'deviationFactor' value to ensure the robot drives straight accurately.

When the robot advances with two servo motors, the speed difference between the wheels may not be noticeable at speeds above 100. However, at the lowest speed of 20, the robot might veer significantly left or right from a straight-line path, creating a curved trajectory. This behavior implies that different correction coefficients need to be applied for various speed values between 20 and 100.

For convenience in correction, a speed value '40', which is almost the median, is used to find an approximate value for straight-line driving, and the same conditions are applied to minimize errors in straight-line driving at other speeds of the robot. The principle of finding the correction value deviationFactor is to apply a positive value in the range of 0.0 to 0.15 if the robot veers left, and a negative value in the range of -0.15 to 0.0 if it veers right, aiming to find the optimum condition value.

Let's take a closer look at each part of the provided code.

(1) Library and Variables, Object Initialization:

```

#include "SelfAbot.h"
#define INVALID_PIN 255

const byte servoLeftPin = INVALID_PIN;
const byte servoRightPin = INVALID_PIN;
float deviationFactor = 0.0; // range -0.15 to +0.15
SelfAbot abot(servoLeftPin, servoRightPin);

```

The 'SelfAbot.h' library is included to utilize the functionalities defined in the SelfAbot class. The variables servoLeftPin and servoRightPin represent the pin numbers to which the servo motors will be connected. Initially set to 255, these may need to be updated to actual pin numbers. The deviationFactor is used as a speed correction coefficient for the servo motors, set to 0 here.

(2) In the setup() function, the abot object is created to control the robot.

```

void setup() {
    abot.setup();
}

```

```

    abot.servoAttachPins(11, 10);
    driveStraight(40, -40);
    delay(2000);
    abot.servoSpeed(0, 0);
    delay(1000);
}

```

The robot is initialized with the call to `abot.setup()`. The pins connected to the left and right servo motors are set with `abot.servoAttachPins(11, 10)`. The function `driveStraight(40, -40)`; is called to make the robot move straight, where 40, -40 represents the speeds of the left and right servos, respectively. After waiting for 2 seconds with `delay(2000)`;, the servo motors are stopped with `abot.servoSpeed(0, 0)`;

(3) The `loop()` function:

```

void loop() {
    // Currently empty.
}

```

The `loop()` function is currently empty, and you can add any code here that you want to run repeatedly.

(4) The `driveStraight()` function:

```

void driveStraight(int leftSpeed, int rightSpeed) {
    // ... (code omitted) ...
    abot.servoSpeed(adjustedLeftSpeed, adjustedRightSpeed);
}

```

This function makes the robot move straight. The `leftCoefficient` and `rightCoefficient` are used to correct the speed of the left and right servo motors, respectively.

The `servoSpeed(adjustedLeftSpeed, adjustedRightSpeed)` is called with `abot.servoSpeed()` to set the servo motor speeds for the `driveStraight()` to enable as straight a drive as possible. This code controls the two servo motors of the robot to perform a straight-line motion.

Note that when testing `driveStraight()` using the example sketch `Ex4_2_abot_driveStraight.ino`, the robot tended to veer to the right. The most accurate straight-line drive was achieved when 'deviationFactor' was corrected to -0.06. It is recommended that you also test this with various driving speed values.

4.3 Left Turn / Right Turn and Continuous Actions

When creating robot movements with two wheels, various types of rotations are depicted in Figure 4.7.

Spot Rotation: First, by operating both wheels at the same speed but in opposite directions to the direction of travel, the robot can be made to spin in place. We will cover what experiments can be done with this movement in later sections.

Pivot Rotation: By stopping one wheel and rotating the other, the robot rotates around the stationary wheel. This action is called pivot rotation.

Curved Drive: Similar to the forward movement of the robot, operating both servo motors in the same direction but at different speeds will cause the robot to rotate largely in the direction of the slowly rotating wheel, drawing a wide arc. This action is called a curved drive.

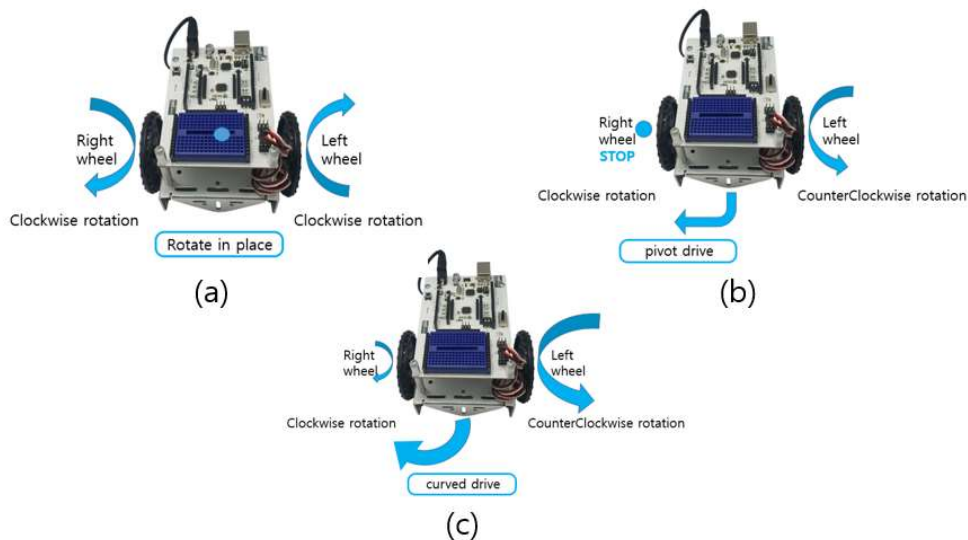


Figure 4.7: Robot Rotation (a) Spot Rotation (b) Pivot Rotation (c) Curved Drive

Now, we introduce the code for stopping, moving forward, pivot turning left, pivot turning right, and moving backward with the Arduino robot. The robot action code below is represented in the sketch for both cases: using one parameter and using two parameters. Function overloading is used to control the wheel actions of the robot in any case.

Ex4.3 abot_basic_drive.ino

```
#include "SelfAbot.h"
#define INVALID_PIN 255;
const byte servoLeftPin = INVALID_PIN;
const byte servoRightPin = INVALID_PIN;

SelfAbot abot(servoLeftPin, servoRightPin);

void setup() {
  abot.setup();
  abot.servoAttachPins(11, 10);
}

void loop() {
  abot(11).servoSpeed(0);
  abot(10).servoSpeed(0);
  delay(1000);

  abot(11).servoSpeed(150);
  abot(10).servoSpeed(-150);
  delay(1000);

  abot(11, 10).servoSpeed(0, -150);
  delay(1000);
  abot(11, 10).servoSpeed(0, 150);
  delay(1000);
  abot(11, 10).servoSpeed(-150, 150);
  delay(1000);
}
```

The practice example maintains a stationary state for 1 second, then moves forward, and with the left

wheel stopped, the right wheel rotates clockwise and counterclockwise for 1 second each, followed by a backward motion. Try creating robot actions by setting various wheel speed changes for different cases.

4.4 Gradual Acceleration and Deceleration

When using code to linearly accelerate the speed of servo motors moving wheels, there are several considerations compared to the basic method of instantaneous speed change:

(1) Energy Efficiency

Gradual Speed Change: This method can be more energy-efficient in some cases. Sudden changes in speed require higher current spikes, which can strain the battery.

Instantaneous Speed Change: Immediate changes can lead to high inrush currents, which are not only inefficient but can also reduce the lifespan of the battery and motor.

(2) Mechanical Stress

Reduced Stress with Gradual Changes: Gradually changing speeds reduces mechanical stress on the motor and all connected mechanical components, potentially extending their lifespan.

Increased Stress with Instant Changes: Sudden changes can cause more wear due to abrupt forces applied to the motor and load.

(3) Battery Life and Performance

Gradual Changes for Battery Life: Smooth transitions in power demand prevent excessive current draw that could heat and damage the battery, helping to maintain its condition over time.

Instant Changes and Potential Battery Strain: High peak currents from sudden speed changes can burden the battery, potentially decreasing its overall lifespan and performance.

(4) Control and Precision

Better Control through Gradual Changes: Gradual changes often allow for better motor control, crucial for applications requiring precision.

Control Challenges with Instant Changes: While instant changes are simpler to implement, they may not offer the same level of control, especially in tasks requiring finesse or precision.

(5) Application-Specific Considerations

Load Characteristics: The efficiency of each method can vary depending on the characteristics of the load driven by the motor.

Operating Environment: Factors like temperature, usage frequency, and required torque can influence which method is more suitable.

The example `gradualServoSpeed(int targetLeftSpeed, int targetRightSpeed)` for Arduino robots describes a method for gradually moving wheels. This approach doesn't immediately execute the target speed but gradually changes it, increasing or decreasing the current speed in steps until the target speed is reached, contrasting with the instantaneous wheel speed change method.

```
void SelfAbot::gradualServoSpeed(int targetLeftSpeed, int targetRightSpeed) {
    int currentLeftSpeed = 0; // Initialize current speed for left servo
    int currentRightSpeed = 0; // Initialize current speed for right servo
    int step = 5; // Determine the step size for speed change

    while (currentLeftSpeed != targetLeftSpeed || currentRightSpeed != targetRightSpeed) {
        unsigned long currentMillis = millis();

        // Accelerate or decelerate the left servo
        if (currentLeftSpeed < targetLeftSpeed) {
            currentLeftSpeed += step;
        } else if (currentLeftSpeed > targetRightSpeed) {
            currentLeftSpeed -= step;
        }
    }
}
```

```

// Accelerate or decelerate the right servo
if (currentRightSpeed < targetLeftSpeed) {
    currentRightSpeed += step;
} else if (currentRightSpeed > targetRightSpeed) {
    currentRightSpeed -= step;
}

// Update servo speeds
_servoLeft.writeMicroseconds(1500 + currentLeftSpeed);
_servoRight.writeMicroseconds(1500 + currentRightSpeed);

// Wait for ms before the next change
while (millis() - currentMillis < 10) {
    // Small delay to wait until ms has passed
}
}
}

```

The operation of the above code is explained below:

currentLeftSpeed and currentRightSpeed are used to track the current speed of each servo. The while loop continues until the current speed matches the target speed for both servos. If statements determine whether to increase or decrease the speed. The millis() function is used for timing, allowing the speed to change every 10ms. Experimenting with time intervals between 10ms to 20ms, 20-50ms, and 50-100ms can help find the optimal value that satisfies both responsiveness and precision.

(Note) -----

What is the **Blocking Code**:

The acceleration/deceleration code for wheels connected to servo motors, due to its nature of changing speed over time, especially when real-time sensor signal processing is required, presents a consideration.

The example code's gradualServoSpeed(int targetLeftSpeed, int targetRightSpeed) method, while executing, prevents the program from performing any other tasks until it completes (i.e., until the servo motors reach the target speed). This effect is referred to as "blocking code." In single-threaded environments like Arduino, code blocking is a limiting factor, where the program cannot respond to sensor inputs, process other tasks, or perform concurrent operations during blocking conditions. If the Arduino robot needs to react in real-time (e.g., obstacle detection sensors), using blocking functions can be problematic since it cannot effectively process such inputs while the function is running.

To address this issue, "state machine logic" or "timer interrupts or callbacks" can be used, allowing for "non-blocking code" that requires more complex setup.

The current example code is only applicable for the initial wheel movement from a stopped state to any arbitrary speed. However, if the wheel's speed continues to change, the current speed values need to be considered further. The code below applies the concept using current speed values as parameters (int currentLeftSpeed, int currentRightSpeed).

```

void SelfAbot::gradualServoSpeed(int currentLeftSpeed, int currentRightSpeed, int targetLeftSpeed,
int targetRightSpeed) {
    int step = 5; // Determine the step size for speed change

    while (currentLeftSpeed != targetLeftSpeed || currentRightSpeed != targetRightSpeed) {
        unsigned long currentMillis = millis();

```

```

// Accelerate or decelerate the left servo
if (currentLeftSpeed < targetLeftSpeed) {
    currentLeftSpeed += step;
} else if (currentLeftSpeed > targetLeftSpeed) {
    currentLeftSpeed -= step;
}

// Accelerate or decelerate the right servo
if (currentRightSpeed < targetRightSpeed) {
    currentRightSpeed += step;
} else if (currentRightSpeed > targetRightSpeed) {
    currentRightSpeed -= step;
}

// Update servo speeds
_servoLeft.writeMicroseconds(1500 + currentLeftSpeed);
_servoRight.writeMicroseconds(1500 + currentRightSpeed);

// Wait for 5ms before the next change
while (millis() - currentMillis < 5) {
    // Small delay to wait until 5ms has passed
}
}
}

```

Managing both the current and target speed values for two wheels can create inconvenience for the user. Therefore, once you start moving the servo motors, you can use a method to automatically store the current speed values in member variables.

In the modified code, the previously introduced parameters can be encapsulated as member variables of the class and applied to the code. You can add the example code below to the 'SelfAbot' class and test it.

```

class SelfAbot {
private:
    int _currentLeftSpeed; // Current speed for left servo as a class member
    int _currentRightSpeed; // Current speed for right servo as a class member

public:
    SelfAbot(): _currentLeftSpeed(0), _currentRightSpeed(0) {
        // Default constructor logic, if any
    }
    // Constructor with servo pin numbers
    SelfAbot(byte servoLeftPin, byte servoRightPin): _currentLeftSpeed(0), _currentRightSpeed(0) {
        attachServos(servoLeftPin, servoRightPin);
    }
    void gradualServoSpeed(int targetLeftSpeed, int targetRightSpeed);
};

void SelfAbot::gradualServoSpeed(int targetLeftSpeed, int targetRightSpeed) {
    int step = 5; // Determine the step size for speed change
    while (_currentLeftSpeed != targetLeftSpeed || _currentRightSpeed != targetRightSpeed) {
        unsigned long currentMillis = millis();
        // Accelerate or decelerate the left servo
        if (_currentLeftSpeed < targetLeftSpeed) {
            _currentLeftSpeed += step;
        } else if (_currentLeftSpeed > targetLeftSpeed) {
            _currentLeftSpeed -= step;
        }
    }
}

```

```

// Accelerate or decelerate the right servo
if (_currentRightSpeed < targetRightSpeed) {
    _currentRightSpeed += step;
} else if (_currentRightSpeed > targetRightSpeed) {
    _currentRightSpeed -= step;
}
// Update servo speeds
_servoLeft.writeMicroseconds(1500 + _currentLeftSpeed);
_servoRight.writeMicroseconds(1500 + _currentRightSpeed);

// Wait for 10ms before the next change
while (millis() - currentMillis < 10) {
    // Small delay to wait until 10ms has passed
}
}
}
}

```

The Arduino sketch below demonstrates how to use the `gradualServoSpeed()` function. Upload the sketch below to practice gradual speed changes. The current method for creating gradual speed changes increments the speed from 0 to 100 by 5, taking 10ms for each increase, thus taking 200ms to reach the target speed. Keep this in mind when operating the robot. To create different patterns of gradual servo speed changes, you can adjust the step intervals or modify the value of 10 in the `while (millis() - currentMillis < 10)` code to change the delay time.

The `gradualServoSpeed()` function uses the global variables `currentLeftSpeed` and `currentRightSpeed` to store and use the current speed states, allowing for the robot's gradual speed changes.

This method will be reintroduced as a practice example for gradual servo speed changes in class inheritance discussed in Chapter 8.

Ex4.4_abot_gradual_speed.ino

```

#include "SelfAbot.h"
#define INVALID_PIN 255;

const byte servoLeftPin = INVALID_PIN;
const byte servoRightPin = INVALID_PIN;

int currentLeftSpeed = 0;
int currentRightSpeed = 0;

SelfAbot abot(servoLeftPin, servoRightPin);

void setup() {
    abot.setup();
    abot.servoAttachPins(11, 10);
}
void loop() {
    gradualServoSpeed(100, -100);
    delay(2000);
    gradualServoSpeed(20, -20);
    delay(2000);
    gradualServoSpeed(-100, 100);
    delay(2000);
    gradualServoSpeed(0, 0);
    delay(2000);
}

```



```

void gradualServoSpeed(int targetLeftSpeed, int targetRightSpeed) {
  int step = 5; // Determine the step size for speed change
  while (currentLeftSpeed != targetLeftSpeed || currentRightSpeed != targetRightSpeed) {
    unsigned long currentMillis = millis();
    // Accelerate or decelerate the left servo
    if (currentLeftSpeed < targetLeftSpeed) {
      currentLeftSpeed += step;
    } else if (currentLeftSpeed > targetLeftSpeed) {
      currentLeftSpeed -= step;
    }
    // Accelerate or decelerate the right servo
    if (currentRightSpeed < targetRightSpeed) {
      currentRightSpeed += step;
    } else if (currentRightSpeed > targetRightSpeed) {
      currentRightSpeed -= step;
    }
    // Update servo speeds
    abot(11, 10).servoSpeed(currentLeftSpeed, currentRightSpeed);
    // Wait for 10ms before the next change
    while (millis() - currentMillis < 10) {
      // Small delay to wait until 10ms has passed
    }
  }
}

```

(1) Robot Hardware Setup:

Connect the robot's left/right servos to the designated pins (11 and 10) on the Arduino board. Ensure the Arduino board is properly powered and the servos are correctly aligned and mounted on the robot.

(2) Sketch Upload:

Connect the Arduino to your computer via USB. Upload the Ex4.4_abot_gradual_speed.ino sketch to the board using the Arduino IDE.

(3) Arduino Robot Operation:

Once the code is uploaded, and the 3-position switch is turned to position 2, the servos should start moving. Observe how the servo speed gradually increases from 0 to 100, maintains for 2 seconds, and then gradually decreases back to 0. The gradual increase and decrease in servo speed should make the robot's movements smoother.

(4) Practice with Various Speeds and Durations:

Modify the gradualServoSpeed parameters within the loop function to experiment with different speeds. Adjust the delay duration to allow the robot to move for longer periods or change directions. Observe how these changes affect the robot's movement and speed transitions.

(5) Understanding the Code:

Examine the gradualServoSpeed function to understand how gradual speed changes are implemented. Pay attention to the use of the while loop and how each servo's speed is adjusted in small steps (step variable) to reach the target speed.

This practice scenario helps understand how to smoothly and controlledly adjust servo motor speeds, providing a foundation for creating more complex and subtle movements in robot projects.

4.5 Additional Method: maneuver()

The maneuver function is a method that takes servo motor movement speed values as parameters and adds duration as an additional parameter. Let's implement the maneuver() function used in the C language basics as a method. The previous maneuver() function moved two servo motors simultaneously and added a delay time as a parameter.

The maneuver method below is slightly different from the maneuver() function used in the C language basics. Let's examine what differs. It can be added to a class for use or directly utilized in an .ino file.

```
void SelfAbot::maneuver(int speedLeft, int speedRight, int msTime) {
    servoSpeed(speedLeft, speedRight);
    if (msTime == -1) {
        // Indefinite operation, servos remain engaged
    } else {
        delay(msTime);
        _servoLeft.detach();
        _servoRight.detach();
    }
}
```

The method for setting speed first involves setting the speed for both the left and right servos. This is achieved using the servoSpeed method of the SelfAbot class, which utilizes two speed values for each the left and right servo.

The servo's delay time checks the msTime parameter:

- (1) If msTime is -1, the servo continues to operate indefinitely. This is useful for continuous operation without a predefined stop time.
- (2) If msTime has a different value, the method waits for that duration (using delay(msTime)) and then detaches the servo to stop it. This servo operation slightly differs from the maneuver() function introduced in the "Online Tutorial Manual" published long ago.

The maneuver() function used in the 'Basics of C language' can be used to execute functions continuously, but the current code is suitable for executing the maneuver() function only once because it is written to automatically stop the servo after the scheduled time. The provided code executes the detach() function as a command to stop the servo, stopping the servo after the scheduled time.

If you want to create continuous movement of the robot using the maneuver() function, you can modify the current code as shown in the example below and apply it.

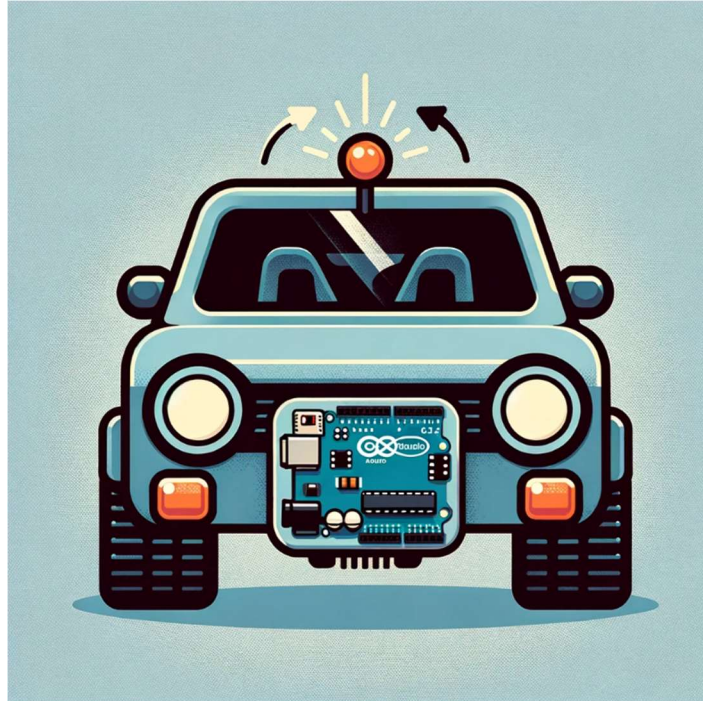
Ex4.5_abot_maneuver_speed.ino

```
#include "SelfAbot.h"
SelfAbot abot; // SelfAbot class instance generation

void setup() {
    Serial.begin(9600); // serial communication begin
    abot.servoAttachPins(11, 10); // servo pin connect to 11, 10
}
void loop() {
    maneuver(100, -100, 200);
    delay(2000); // 2 seconds delay
}
void maneuver(int speedLeft, int speedRight, int msTime) {
    abot.servoSpeed(speedLeft, speedRight);
    if (msTime == -1) {
        // Indefinite operation, servos remain engaged
    } else {
        delay(msTime);
    }
}
```

Chapter 5 Phototransistor Light Signal: rcTime()

- 5.1 Practice for Processing Analog Input Signals
- 5.2 Digital Input Signal Method: rcTime() Method
- 5.3 Recognizing Light Intensity with Two Phototransistors
- 5.4 Arduino Robot Light Following Exercise



This image represents the wavelength band of the Infrared (IR) sensor. The infrared light region consists of wavelengths that cannot be identified with the naked eye. The phototransistor used in the exercises of this chapter is most sensitive at a wavelength of 850nm, and the infrared sensor used in Chapter 6 responds most sensitively at 980nm.

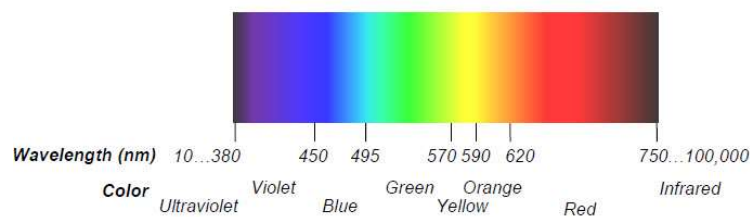


Figure 5.1: The Colors of Visible Light and the Wavelength Bands of Invisible Light (Source: parallax.com)

Analog and digital sensors are commonly used in robotics, each with unique characteristics and uses. Understanding the differences between these two types of sensors is crucial for effectively integrating them into robotic systems.

Features of analog sensors include:

- Analog sensors generate a continuous signal that represents physical measurements, with the signal varying across a continuous range.

- The resolution of analog sensors is theoretically infinite, allowing for the measurement and representation of very small changes in the measured quantity. However, in practice, resolution is limited by noise levels and the precision of the analog-to-digital converter (ADC) used for signal digitization.
- Analog sensors typically output a voltage proportional to the measurement, ranging between a minimum and maximum value (e.g., 0-5V). They are generally simpler and cheaper than digital sensors but may incur higher costs and complexity for signal processing, especially for high precision.
- Analog signals are more susceptible to noise and interference, potentially degrading signal quality over long distances. Common examples include thermistors (temperature sensors), photodiodes (light sensors), and potentiometers (position sensors).

Features of digital sensors:

- Digital sensors typically generate discrete signals in a binary data format. The output is either high (1 or HIGH) or low (0 or LOW) depending on various conditions of the measured parameter.
- The resolution of digital sensors is limited by the number of bits of digital output. For example, a 10-bit sensor can represent 2^{10} (1024) discrete states. These sensors usually communicate using digital communication protocols like I2C, SPI, or UART.
- Digital sensors are more complex and may be more expensive but often include signal processing capabilities for more accurate and stable readings.
- Digital signals are less sensitive to noise than analog signals, enhancing the reliability of digital sensors in long-distance transmissions and electrically noisy environments. Examples include digital thermometers, infrared sensors, and digital accelerometers.

5.1 Practice for Processing Analog Input Signals

This section introduces an exercise from a C language Arduino robotics tutorial guide for displaying analog input signals on a screen. Connect the analog sensor input to Arduino pin A2 and practice with the code provided.

Analog input signals tend not to be used directly for robot actuation due to reasons introduced below. The 'SelfAbot' library does not implement methods for processing analog input signals. Practice with the code that operates independently of the library.

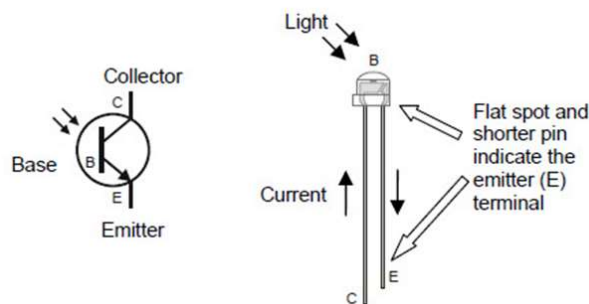


Figure 5.2: Phototransistor Symbol and Schematic (Source: parallax.com)



Figure 5.3: Analog Signal Processing Circuit for Phototransistor Light Sensor

It explains why analog signals are less often used as external input signals for robot actuation, with a tendency to prefer digital input signals for their advantages in terms of task specificity, precision, simplicity, and robustness against noise, leading to a preference for digital control.

To process the analog signal of a phototransistor, connect it as shown in Figure 5.3. To adjust the sensitivity of the light sensor signal, change the size of the resistor value as illustrated in Figure 5.3.

Ex5.1_phototransistor_analogReadvolts.ino

```
int sensorPin = A2;

void setup() {
  // abot.setup();
  Serial.begin(9600);
}

void loop() {
  Serial.print("A2 = ");
  Serial.print(volts(A2));
  Serial.println(" volts");
  delay(1000);
}

float volts(int adpin) // Measures volts at adPin
{ // Returns floating point voltage
  return float(analogRead(adpin)) * 5.0 / 1024.0;
}
```

(1) Precision and Noise: Analog signals can be more susceptible to noise and interference, potentially leading to less accurate or fluctuating readings. This can be a significant disadvantage in robotics, where precision and consistency are crucial. In contrast, digital signals are more resistant to noise.

(2) Processing Complexity: Analog signals require conversion to digital values using Analog-to-Digital Converters (ADC) for processing. This conversion can add complexity and latency, especially in cases where high-resolution readings are needed.

(3) Demand for Discrete Control: Many robotic functions, such as motor control, are inherently digital (on/off, forward/backward, stop/go). Using digital signals for these controls is simpler and more efficient.

(4) Limited Analog Inputs: Arduino boards typically have a limited number of analog input pins compared to digital pins. In robotics, where many sensors and actuators can be used, the availability of digital pins makes complex designs more practical.

(5) Standardization and Compatibility: Many robotic components, like servos, DC motors with drivers, and sensors, are designed to work with digital control signals. Using these standard components with analog signals may require additional circuits or conversion methods.

(6) Variability with Power Supply Levels: Analog signals can vary with changes in the power supply voltage, potentially leading to inconsistent operation. Digital signals are less affected by these fluctuations, offering more stable operation.

(7) Programming and Debugging Ease: Often, using digital signals can be easier in terms of programming and debugging. The binary nature of digital signals (high/low, true/false) aligns well with programming logic, making it simpler to implement control algorithms and troubleshoot issues.

However, analog inputs may be more appropriate for sensors providing a continuous range of readings (e.g., temperature, luminosity, or distance sensors) or situations requiring gradual changes and fine control (e.g., controlling the position of a servo motor using a potentiometer).

5.2 Digital Input Signal Method: rcTime() Method

The earlier sections covered processing a phototransistor light sensor using an analog signal method. Now, it's time to introduce processing the same phototransistor light sensor using a digital signal method.

Figure 5.4 shows how to combine a light sensor signal with an RC circuit to process digital signals. In the circuit shown in the figure, transitioning the digital pin's mode from HIGH to LOW creates an RC circuit.

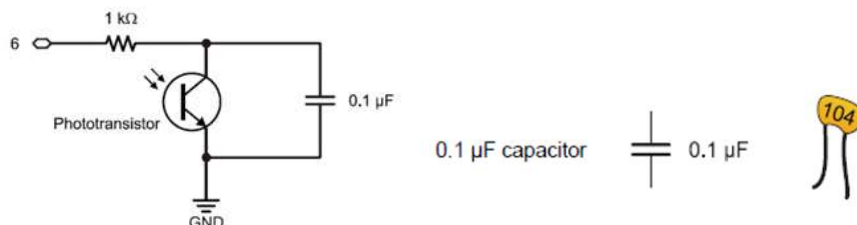


Figure 5.4: Circuit for Measuring Digital Signals Figure 5.5: Capacitor Symbol and Notation (Source: parallax.com)

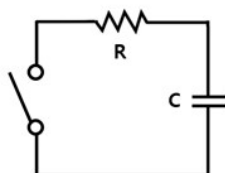
The below Ex5.2 .ino example code uses the 'SelfAbot' class's rcTime method to read digital input signals from two pins (8 and 9) of an Arduino. This method processes digital input signals using a common technique called RC time, an engineering concept of time constants, similar to reading analog-like values from digital pins.

Understanding RC Circuits

An RC circuit is a basic electrical circuit made up of a resistor and a capacitor used together. When voltage is applied, the capacitor charges through the resistor. When the voltage is removed, the capacitor discharges. The time it takes to charge or discharge depends on the values of the resistor and capacitor. The time constant of an RC circuit, denoted by the Greek letter tau (τ), measures the time it takes for the voltage across the capacitor to charge to about 63.2% of its maximum value during charging or to decrease to about 36.8% during discharging, referred to as 1 tau.

(Note) -----

Understanding RC Time Constant



The RC time constant represents how quickly a capacitor can charge or discharge through a resistor, forming a simple electrical circuit with components to limit or control the flow of current (R) and store electrical energy in an electric field (C).

The time constant τ is calculated using the formula:

$$\tau = R \times C$$

where τ (tau) is the time constant in seconds, R is resistance in ohms (Ω), and C is capacitance in farads (F).

This defines the time constant or tau as a relative representation of the total 100%, allowing the state of charging and discharging to be represented by the same time constant (e.g., 1 tau for 63.2% charging or 36.8% discharging) across any combination of components in an RC circuit, only the conversion time value for 1 tau changes based on the component combination.

(1) Charging the Capacitor: When power is supplied to the circuit, the capacitor begins to charge through the resistor. 1 time constant (τ) is the time it takes for the voltage across the capacitor to reach about 63.2% of the supply voltage.

(2) Discharging the Capacitor: When the circuit is disconnected from power, the capacitor begins to discharge through the resistor. Here, 1τ is the time it takes for the capacitor's voltage to drop to about 36.8% of its initial value.

(3) 5 Time Constants: Generally, a capacitor is considered almost fully charged after about 5τ (five time constants) or almost fully discharged.

Example Scenario

Imagine an RC circuit with a 1000Ω (1k Ω) resistor and a $1\mu\text{F}$ capacitor. The time constant τ for this circuit would be:

$$\tau = 1000\Omega \times 1\mu\text{F} = 1000 \times 10^{-6} \text{ seconds} = 1 \text{ millisecond}$$

This means it takes 1 millisecond for the capacitor to charge to 63.2% of the supply voltage or discharge to 36.8% of its initial voltage. The time it takes to almost fully charge or discharge is about 5 milliseconds.

Microcontroller Role

An Arduino (or other microcontrollers) is used to control the charging and discharging processes and measure the time these processes take, representing the characteristics of the RC circuit.

Executing `abot.rcTime(8)` in the code reads the delay time until the `digitalRead` value changes from High to Low state, processing the digital input signal based on a specific moment when the external input signal changes. For Arduino Uno, the microcontroller should recognize the threshold value for HIGH at $0.6 * VCC$ (5V) and for LOW at $0.3 * VCC$ (5V). In the RC circuit, starting from an initial 5V, as the voltage gradually decreases to a specific point where Arduino changes recognition from HIGH to LOW state, the code measures the time taken.

Refer to Figure 5.4 for external circuit configuration.

In implementing the `rcTime()` method below, function overloading was not used, based on the design principle that applies only to the pins moving the wheels. Subscribers conducting the experiment can add methods that use function overloading if desired. All sensor signal processing methods described later do not use function overloading.

Introduction to the implementation content of the 'SelfAbot' class's `rcTime()` method:

```
unsigned long SelfAbot::rcTime(byte pin) {
    pinMode(pin, OUTPUT);
    digitalWrite(pin, HIGH);
    delay(1);
    pinMode(pin, INPUT);
    digitalWrite(pin, LOW);
    unsigned long startTime = micros();
    unsigned long elapsedTime = 0;
    while (digitalRead(pin) == HIGH) {
        elapsedTime = micros() - startTime;
    }
    return elapsedTime;
}
```

The 'rcTime' method in the 'SelfAbot' class measures the time it takes for the state of a digital input pin to change from HIGH to LOW. This method is commonly used to measure the charging or discharging time of a capacitor in an RC (resistor-capacitor) circuit. Breaking it down for easier understanding, the steps are as follows:

Understanding the rcTime Method Step by Step:

(1) Set pin mode to output: `pinMode(pin, OUTPUT);`

This line configures the specified 'pin' as an output. To charge a capacitor in the RC circuit, you first need to set up the pin for output.

(2) Set the pin to HIGH: `digitalWrite(pin, HIGH);`

This line sends a HIGH signal (5V on most Arduino boards) to the pin, starting to charge the capacitor connected to that pin.

(3) Short delay: `delay(1);`

This introduces a brief pause to allow time for the capacitor to charge. This delay period can affect the charging time.

(4) Switch the pin to input: `pinMode(pin, INPUT);`

The pin is reconfigured as an input. This is done to measure the discharging time of the capacitor.

(5) Ensure the pin is LOW: `digitalWrite(pin, LOW);`

Even though the pin is set to input, this line is often used to ensure that the internal pull-up resistor is deactivated. It ensures that the pin is influenced only by the external circuit, not by Arduino.

(6) Start timing: `unsigned long startTime = micros();`

This line records the start time using Arduino's internal function 'micros()', which provides the time in microseconds since the Arduino started running the current program.

(7) Measure the time until the pin becomes LOW:

The while loop `while(digitalRead(pin) == HIGH)` runs as long as the pin remains HIGH. Within the loop, `elapsedTime = micros() - startTime;` calculates the time elapsed since the start time. This effectively measures the time it takes for the capacitor to discharge through the resistor until the pin is indicated as LOW.

(8) Return the elapsed time: `return elapsedTime;`

Finally, this method returns the elapsed time in microseconds. This value represents a specific value of the RC time, characteristic of the specific RC circuit connected to the pin.

The code below is an example that can use the rcTime method for detecting external signals.

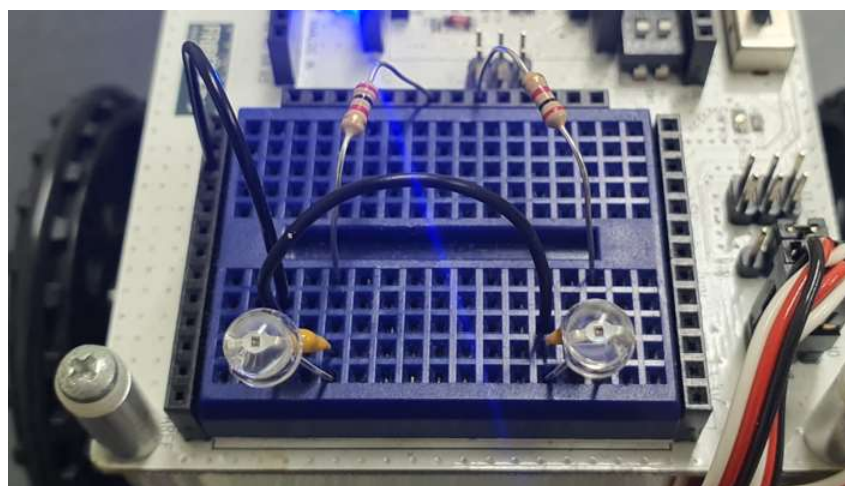


Figure 5.6: Mounting a Phototransistor Light Sensor on a Robot

Ex5.2_phototransistor_rcTime.ino

```

#include "SelfAbot.h"
SelfAbot abot;

void setup() {
  // abot.setup();
  Serial.begin(9600);
}
void loop() {
  unsigned long rctimeValLeft = abot.rcTime(8);
  Serial.print("rcTime Left = ");
  Serial.print(rctimeValLeft);
  Serial.println(" us || ");

  unsigned long rctimeValRight = abot.rcTime(6);
  Serial.print("rcTime Right = ");
  Serial.print(rctimeValRight);
  Serial.println(" us");
  delay(500);
}

```

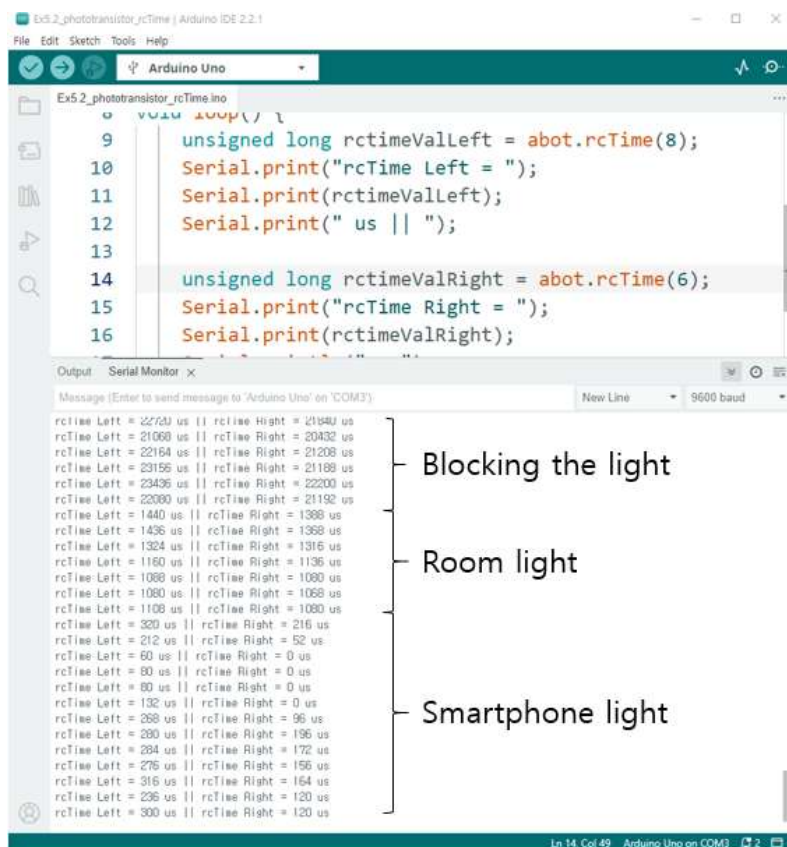


Figure 5.7: Sensor Output Value Changes When Exposed to External Light Versus When Not

Figure 5.7 displays the change in output values of two light sensors when exposed to smartphone light, ambient fluorescent light, and when covered by hand to block external light. Compared to the analog input circuit method, the change in data width due to variations in external light is significantly larger.

Advantages and Disadvantages of Digital Signal Processing Method:

Advantages:

- (1) **Simplicity:** Utilizes the abundant digital I/O pins on Arduino boards compared to analog pins, offering a more varied approach for a broader range of applications.
- (2) **No Need for ADC:** Operates on digital pins, simplifying circuit design and coding without the need for analog-to-digital conversion (ADC).
- (3) **Cost-Efficiency:** Utilizing digital pins for analog-like readings eliminates the need for additional hardware like external ADCs, making it cost-effective.
- (4) **Customizable Resolution:** Timing and code logic manipulation allow for some degree of measurement resolution customization to specific application requirements.

Disadvantages:

- (1) **Lower Precision:** Digital readings may lack accuracy, especially where wide sensor output ranges or high resolutions are needed, compared to actual analog readings.
- (2) **Sensitivity to Noise:** Digital signal processing can be more susceptible to electrical noise, affecting the accuracy of timing-based measurements.
- (3) **Processor Time:** Methods like `rcTime` can be processor-intensive, relying on loops to measure the time taken for pin state changes, potentially limiting the processor's ability to handle concurrent tasks.
- (4) **Calibration and Consistency Issues:** The accuracy of timing-based measurements can be affected by factors like temperature, voltage changes, or variances across different boards, making consistent and reliable calibration challenging.
- (5) **Limited Application Scope:** This approach may not suit all sensor types, especially those requiring very accurate and stable analog readings.

5.3 Recognizing Light Intensity with Two Phototransistors

Using two light sensors, it's possible to compare the intensity of light on the left and right sides. This method allows for differentiation between the darker and brighter sides. By integrating such sensor recognition with robot movement, code can be written to make the robot follow bright light or avoid it, moving towards darker areas.

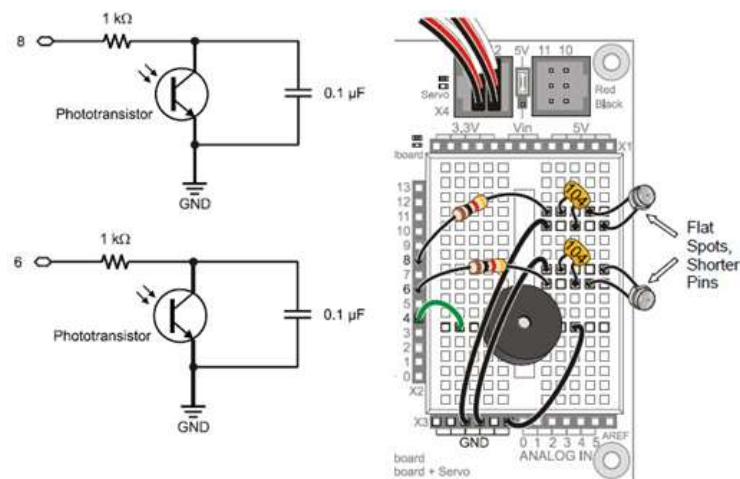


Figure 5.8: Arduino Robot Connected with Phototransistors (Source: parallax.com)

The method of writing the coding example introduced in C language using the 'SelfAbot' class in C++ code is as follows. The code below reads the left light sensor signal using the `abot.rcTime(8)` method, and the right light sensor signal using the `abot.rcTime(6)` method, then normalizes it in the manner of: $\text{rcTimeValLeft} / (\text{rcTimeValLeft} + \text{rcTimeValRight}) - 0.5$; This approach to normalizing sensor data simplifies further data processing and facilitates data visualization.

Ex5.3_normalize_two_phototransistors.ino

```
#include "SelfAbot.h"
SelfAbot abot;

void setup() {
  // abot.setup();
  Serial.begin(9600);
}
void loop() {
  float rctimeValLeft = float(abot.rcTime(8));
  float rctimeValRight = float(abot.rcTime(6));

  float ndShade;
  ndShade = rctimeValLeft / (rctimeValLeft + rctimeValRight) - 0.5;
  for(int i = 0; i < (ndShade * 40) + 20; i++) {
    Serial.print(' ');
  }
  Serial.println('*');
  delay(100);
}
```

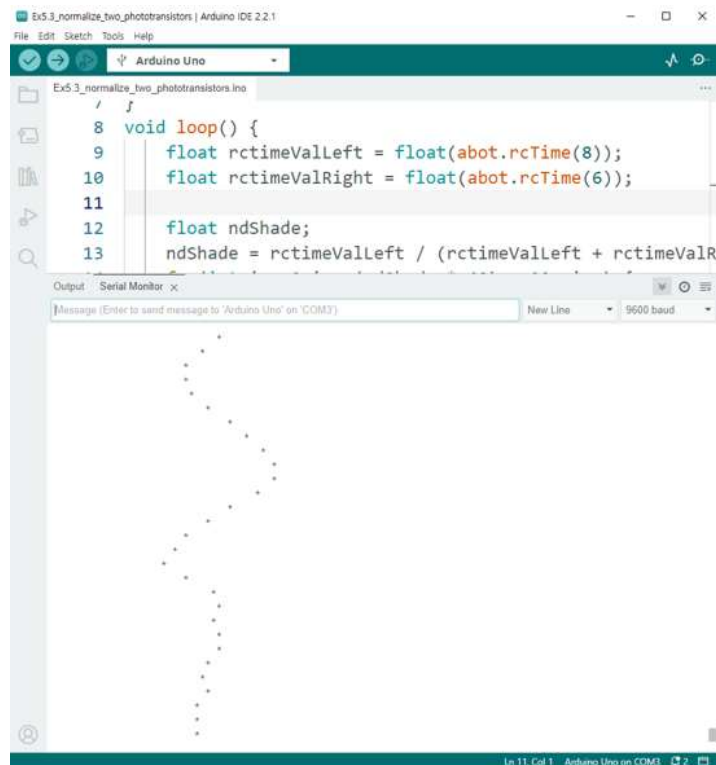


Figure 5.9: Displaying the Difference in Light Between Two Phototransistors as a Serial Output Graph

Refer to the following explanation for the example code introduced above:

(1) Library and Object Initialization:

```
#include "SelfAbot.h"
SelfAbot abot;
```

This includes the SelfAbot.h library to utilize the functionalities defined in the SelfAbot class. An abot object is created to control the robot.

(2) setup() Function:

```

void setup() {
  //abot.setup();
  Serial.begin(9600);
}

```

Serial.begin(9600); is called to initiate serial communication, enabling data transmission and reception through the serial monitor.

(3) loop() Function:

```

void loop() {
  float rctimeValLeft = float(abot.rcTime(8));
  float rctimeValRight = float(abot.rcTime(6));
  ...
}

```

The loop() function reads values from sensors connected to pins 8 and 6 using the rcTime method. rctimeValLeft and rctimeValRight store the read values from the left and right sensors, respectively.

(4) Processing and Visualizing Sensor Values:

```

float ndShade;
ndShade = rctimeValLeft / (rctimeValLeft + rctimeValRight) - 0.5;
for(int i = 0; i < (ndShade * 40) + 20; i++) {
  Serial.print(' ');
}
Serial.println("*");

```

ndShade is calculated based on the ratio of the two sensor values, representing the difference between them, with -0.5 subtracted to center the value.

A for loop and Serial.print(' '); are used to output spaces on the serial monitor proportional to the ndShade value, visually representing the difference in sensor values.

Serial.println("*"); prints an asterisk, indicating the position of the difference in sensor values.

delay(100); adds a 100-millisecond delay between each iteration of the loop, preventing too rapid repetition of data reading and output.

5.4 Arduino Robot Light Following Exercise

This code demonstrates the process of detecting light intensity using phototransistor sensors, converting it into digital signals, and then changing the values of variables to operate servo motors. As previously explained, the normalized sensor data value ndShade varies between -0.5 and +0.5. If the ndShade value is negative, it indicates that the numerator value is relatively small, meaning the rctimeValLeft value is small, implying that the left side is brighter.

When the robot is moving forward, adjusting the servo speed values of both wheels allows the robot to follow the light. If the left side is brighter, indicating a left turn, adjusting the left wheel's speed value lower than the right wheel's speed value fulfills this condition. The same principle applies to the servo motor on the right wheel if the opposite condition is met.

Ex5.4_photoTr_rctime_serialprint.ino

```

#include "SelfAbot.h"
SelfAbot abot;

void setup() {
  // abot.setup();
  Serial.begin(9600);
}

```

```

void loop() {
  int leftSpeed, rightSpeed;
  float rctimeValLeft = float(abot.rcTime(8));
  float rctimeValRight = float(abot.rcTime(6));

  float ndShade;
  ndShade = rctimeValLeft / (rctimeValLeft + rctimeValRight) - 0.5;

  if (ndShade < 0.0) {
    leftSpeed = int(200.0 + (ndShade * 1000.0));
    leftSpeed = constrain(leftSpeed, -200, 200);
    rightSpeed = 200;
  } else {
    rightSpeed = int(200.0 - (ndShade * 1000.0));
    rightSpeed = constrain(rightSpeed, -200, 200);
    leftSpeed = 200;
  }
  Serial.print(leftSpeed, DEC);
  Serial.print(" ");
  Serial.print(ndShade, DEC);
  Serial.print(" ");
  Serial.println(rightSpeed, DEC);
  delay(1000);
}

```

The code below is intended to check, via Arduino's serial output, how the speed values required for the operation of the robot's servo motors change according to external light changes, before directly activating the servo motors. If the changes in values due to external light variations are deemed appropriate on the serial output screen, it is then acceptable to proceed to the code example that directly operates the servo motors.

```

19     rightSpeed = 200;
20   } else {
21     rightSpeed = int(200.0 - (ndShade * 1000.0));
22     rightSpeed = constrain(rightSpeed, -200, 200);
23     leftSpeed = 200;
24   }
25
26   Serial.print(leftSpeed, DEC);
27   Serial.print(" ");
28   Serial.print(ndShade, DEC);
29   Serial.print(" ");
30   Serial.println(rightSpeed, DEC);
31   delay(1000);
32 }
33

```

Output Serial Monitor x

Message (Enter to send message to 'Arduino Uno' on 'COM3')

New Line 9600 baud

```

181 -0.0187940114 200
200 0.0091242539 191
196 -0.0038520693 200
137 -0.0624309492 200
159 -0.0403800582 200
200 0.0000000000 200
196 -0.0037257969 200
200 0.0151659654 184
200 0.0318204685 168
28 -0.1718987226 200
200 0.0329719781 167
200 0.0080000162 191
200 0.0189556411 181
200 0.1388910583 61

```

Figure 5.10: Serial output converting the difference in light sensor detection values into robot speed values

If it is determined that the servo motor speed values are being generated correctly in the previous example, the code can be modified to make the robot operate according to the direction of light from the sensors.

The principle of creating robot driving actions responsive to the direction of external light involves passing the values of the leftSpeed and rightSpeed variables as parameters to the robot's driving method. The method used to drive the robot using the library is servoSpeed(leftSpeed, -rightSpeed). Using the instance with the code allows for the creation of robot actions responsive to external light, introduced below as an example.

Ex5.5 light following abot.ino

```
#include "SelfAbot.h"
#define INVALID_PIN 255;
const byte servoLeftPin = INVALID_PIN;
const byte servoRightPin = INVALID_PIN;

SelfAbot abot(servoLeftPin,servoRightPin);

void setup() {
  abot.setup();
  abot.servoAttachPins(11, 10);
  Serial.begin(9600);
}

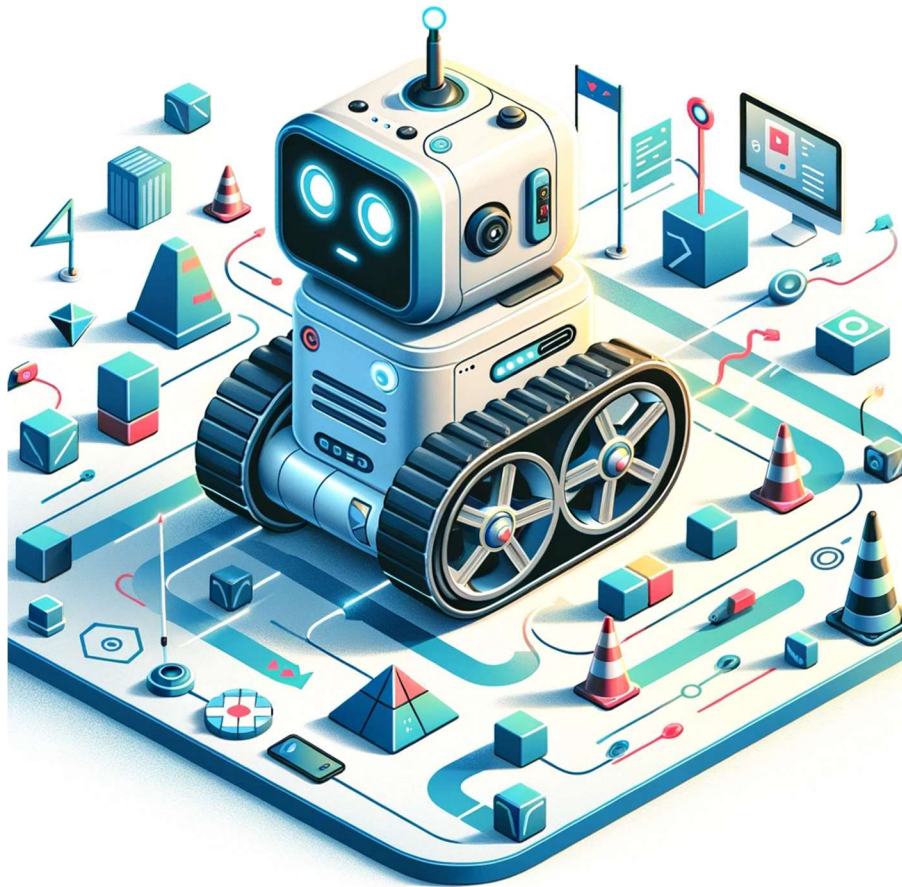
void loop() {
  int leftSpeed, rightSpeed;
  float rctimeValLeft = float(abot.rcTime(8));
  float rctimeValRight = float(abot.rcTime(6));

  float ndShade;
  ndShade = rctimeValLeft / (rctimeValLeft + rctimeValRight) - 0.5;

  if (ndShade < 0.0) {
    leftSpeed = int(200.0 + (ndShade * 1000.0));
    leftSpeed = constrain(leftSpeed, -200, 200);
    rightSpeed = 200;
  } else {
    rightSpeed = int(200.0 - (ndShade * 1000.0));
    rightSpeed = constrain(rightSpeed, -200, 200);
    leftSpeed = 200;
  }
  abot(11, 10).servoSpeed(leftSpeed, -rightSpeed);
}
```

Chapter 6 Driving Robot with Infrared Sensors

- 6.1 Infrared Sensor Signal Processing for Obstacle Detection
- 6.2 Integrating Infrared Sensor Signals into the Robot
- 6.3 Using the irDistance Method with Infrared Sensors
- 6.4 Scanning Surrounding Objects with Infrared Sensors
- 6.5 Finding Nearby Objects by Scanning with Infrared Sensors



Infrared (IR) sensors are an excellent subject for students to learn how to connect sensors to an Arduino robot and read output data. Especially since IR sensors can produce both digital and analog signals, as explained in previous sections, they enable a variety of experimental experiences.

This chapter introduces methods for handling detection signals in infrared transmission and reception modes. It also introduces the principle of detecting objects in front of the robot using this method. Through projects utilizing IR sensors, it's hoped that students will understand and learn 'how robots interact with their environment,' a core concept in fields like robotics and automation.

6.1 Infrared Sensor Signal Processing for Obstacle Detection

The IR sensor introduced in Figure 6.1 is composed of a transmitting LED and a receiving receiver component. Different from the phototransistor components used in previous sections, the IR LED with housing is used to enhance the directivity of the infrared light. The method of connecting the IR sensor transmitter and receiver components to Arduino pins is to follow the circuit diagram below.

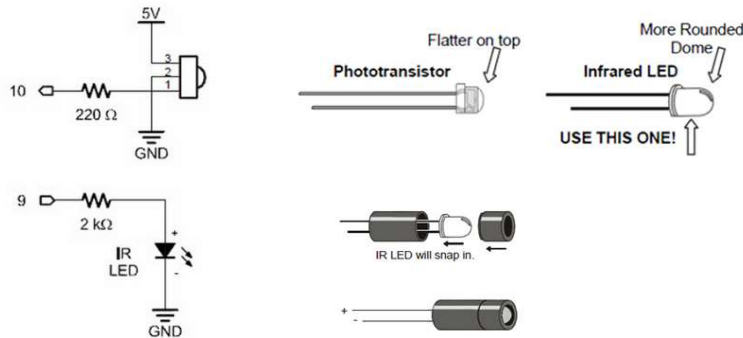


Figure 6.1: IR Transmitter and Receiver Sensor Circuit and Component Overview (Source: parallax.com)

This is a method from the 'SelfAbot' class for detecting IR sensor signals. First, the method `irDetect` for reading the IR sensor is introduced and explained.

```
int SelfAbot::irDetect(byte irLedPin, byte irReceiverPin, int frequency) {
    _irLedPin = irLedPin;
    _irReceiverPin = irReceiverPin;

    pinMode(_irLedPin, OUTPUT);
    pinMode(_irReceiverPin, INPUT);

    tone(_irLedPin, frequency, 8);
    delay(1);
    int _lastReadValue_ir = ::digitalRead(_irReceiverPin);
    delay(1);

    return _lastReadValue_ir;
}
```

(1) Function Declaration

`int SelfAbot::irDetect(byte irLedPin, byte irReceiverPin, int frequency)` defines the member function `irDetect` in the `SelfAbot` class. The function takes three parameters: `irLedPin`, `irReceiverPin`, and `frequency`.

(2) Variable Assignment

`_irLedPin = irLedPin;` and `_irReceiverPin = irReceiverPin;` These lines of code assign the pin numbers passed to the function to the class's internal variables.

(3) Pin Mode Setting

`pinMode(_irLedPin, OUTPUT);` and `pinMode(_irReceiverPin, INPUT);` The `pinMode` function sets the pins to input or output mode. Here, `_irLedPin` is set to output (used to send signals to the infrared LED), and `_irReceiverPin` is set to input (used to read signals from the infrared receiver).

(4) Infrared Signal Generation

`tone(_irLedPin, frequency, 8);` The `tone` function generates an infrared signal at the specified frequency on `_irLedPin`. This signal lasts for a certain duration (here, 8).

(5) Sensor Reading and Return


```
int _lastReadValue_ir = ::digitalRead(_irReceiverPin);
```

The digitalRead function reads the digital signal at _irReceiverPin and stores it in the _lastReadValue_ir variable. This represents the value detected by the IR receiver.

```
return _lastReadValue_ir; Returns the value of the _lastReadValue_ir variable for return.
```

To use the irDetect() method, refer to the figure below to install the sensor components.

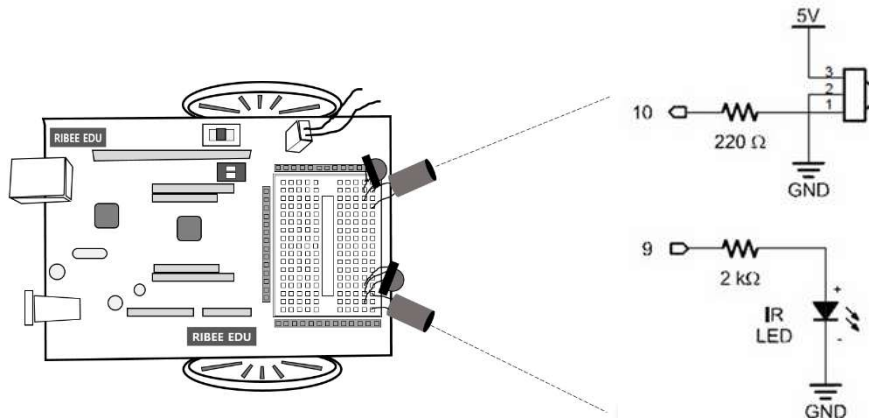


Figure 6.2: Sensor Exploration Direction and Circuit Diagram for Autonomous Arduino Robot

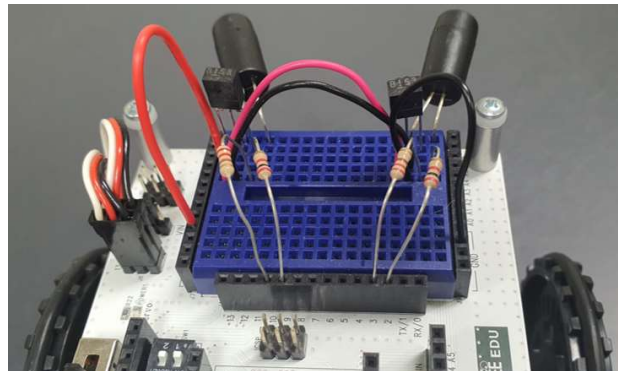


Figure 6.3: Arduino Robot Equipped with Infrared Sensor

The usage of infrared sensors on the Arduino robot as shown in Figure 6.2 involves independently detecting and processing signals from objects in the left and right front directions. The sensors are uniquely positioned to face more towards the left and more towards the right from the center.

The following example sketch utilizes the infrared detection methods of the 'SelfAbot' class to process the signals from the infrared sensors mounted on the robot.

This method of using sensors is identical to the content described in the online tutorial 'Playing with Arduino Robot' distributed for C language education. Try achieving the same practical effects using the 'SelfAbot' written in C++ language. Running this code will allow you to display the values of both the left and right infrared sensors on the screen.

Ex6.1_sensor_irdetect.ino

```
#include "SelfAbot.h"
SelfAbot abot;

void setup() {
  Serial.begin(9600);
  abot.setup();
}
```



```

if ((irLeft == 0) && (irRight == 0)) { // no obstacle
  abot(13, 12).servoSpeed(-100, -100);
  delay(20);
} else if ((irLeft == 0) && (irRight == 1)) { // left detect
  abot(13, 12).servoSpeed(-100, 100);
  delay(1000);

  abot(13, 12).servoSpeed(100, 0);
  delay(400);
} else if ((irLeft == 1) && (irRight == 0)) { // right detect
  abot(13, 12).servoSpeed(-100, 100);
  delay(1000);

  abot(13, 12).servoSpeed(0, -100);
  delay(400);
} else { // both detect
  abot(13, 12).servoSpeed(-100, 100);
  delay(1000);

  abot(13, 12).servoSpeed(100, 0);
  delay(800);
}

```

The if-else structure works by checking each condition sequentially. The movement of the robot is determined by the combination of 'irLeft' and 'irRight' sensor values. The advantage of this method is that it's easy for beginners to understand and is straightforward, clearly showing the logic combination of each sensor value. However, as the number of conditions increases, the 'if-else' statements can become lengthy and more challenging to manage.

Let's consider changing to a 'switch' statement for the same robot movement control effect.

To convert if statements to a 'switch' statement, additional steps are required. First, use the combinedState variable to combine irLeft and irRight into a single 2-byte value. Then, the switch statement selects operations based on this combined value.

Ex6.2 irdetect_navigation_abot_switch.ino

```

#include "SelfAbot.h"
#define INVALID_PIN 255;

const byte servoLeftPin = INVALID_PIN;
const byte servoRightPin = INVALID_PIN;

SelfAbot abot(servoLeftPin, servoRightPin);

void setup() {
  //Serial.begin(9600);
  abot.setup();
  abot.servoAttachPins(13, 12);
}

void loop() {
  int irLeft = abot.irDetect(9, 10, 38000); // left IR sensor data read
  int irRight = abot.irDetect(2, 3, 38000); // right IR sensor data read
  int combinedState = (irLeft << 1) | irRight;

  switch (combinedState) {
    case 0: // Both sensors detect no obstacle
      abot(13, 12).servoSpeed(100, -100);
      break;
    case 1: // irLeft = 0, irRight = 1 left obstacle

```

```
    abot(13, 12).servoSpeed(-100, 100);
    delay(1000);
    abot(13, 12).servoSpeed(100, 0);
    delay(400);
    break;
case 2: // irLeft = 1, irRight = 0 right obstacle
    abot(13, 12).servoSpeed(-100, 100);
    delay(1000);
    abot(13, 12).servoSpeed(0, -100);
    delay(400);
    break;
case 3: // Both sensors detect an obstacle
    abot(13, 12).servoSpeed(-100, 100);
    delay(1000);
    abot(13, 12).servoSpeed(100, 0);
    delay(800);
    break;
}
delay(20);
}
```

In this code, combinedState represents a value that indicates the states of both irLeft and irRight. It is created by shifting the irLeft value left by 1 bit and then performing a bitwise OR with irRight. In this way, each possible combination of 'irLeft' and 'irRight' values (00, 01, 10, 11) is represented by a unique 'combinedState' value (0, 1, 2, 3).

The advantage of this approach is that the switch statement can be more concise and easier to read, especially when there are many conditions. Additionally, it is generally more efficient in execution. However, for beginners, understanding how the combinedState is calculated can be somewhat complex.

Explanation for Beginners

Which method to choose? Both methods are valid, but the choice depends on the situation and what you are comfortable with. 'If-else' is simpler, while 'switch' can be more efficient and tidy when dealing with multiple conditions.

Adding to the logical understanding, regardless of the method, understanding the logic behind the sensor readings and the corresponding actions is crucial. Both methods require practice, and it's important to implement both in projects to see which one you prefer in various situations. When starting to code, begin with simple logic and gradually add complexity as you become familiar with the syntax and concepts. Remember, the best way to learn programming is by doing it yourself. Experiment with both methods and see which works best in different scenarios.

6.3 Using the irDistance Method with Infrared Sensors

Looking at the operational characteristics of the infrared sensor by frequency in Figure 6.4, it shows the highest sensitivity at 38000 Hz. Moreover, the sensitivity shows almost linearly decreasing characteristics at frequencies higher or lower than 38000 Hz. Observing the sensitivity characteristics to frequency in Figure 6.5, linear characteristics are presented in the range of 38000 ~ 42000 Hz, indicating that this linearity can be easily converted for use in measuring the distance to an object.

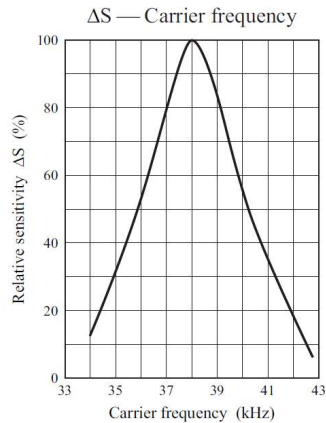
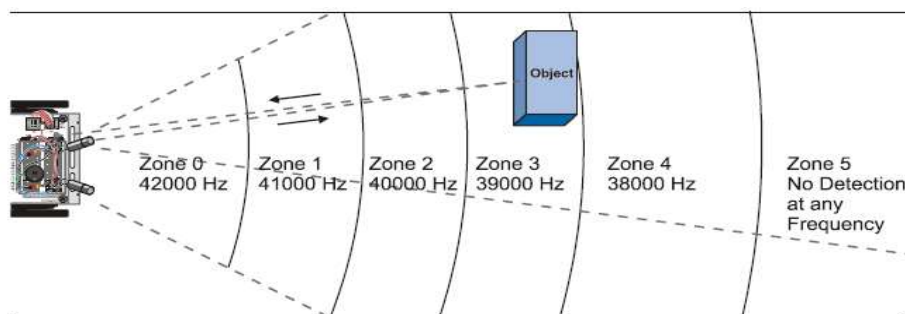


Figure 6.5: Frequency Characteristics Graph of the Infrared Sensor (Panasonic PNA4602M)

The principle of measuring the distance to an object using the frequency scanning method is that the further the distance to the object, it is detected only under the condition of 38000 Hz, and the closer the distance to the object, it can detect the object up to the frequency of 42000 Hz. The irDistance method introduced below differentiates the relative distance to an object by scanning frequencies and summing up the detection signals for each frequency.



NOTE: This diagram is just an illustration. The actual detection range is several cm away from the front of the BOE Shield-Bot and also only covers a few cm of distance detection.

Figure 6.6: Principle of irDistance() for Distance Measurement by Frequency (Source: parallax.com)

The irDistance method of the 'SelfAbot' class can measure the distance to an object. Logic has been added to use the previously introduced irDetect method to measure the distance to an object.

```
int SelfAbot::irDistance(byte irLedPin, byte irReceivePin) {
    int distance = 0;
    for(long f = 38000; f <= 42000; f += 500) {
        distance += irDetect(irLedPin, irReceivePin, f);
    }
    return distance;
}
```

(1) Function Declaration

`int SelfAbot::irDistance(byte irLedPin, byte irReceivePin):` This is a method of the 'SelfAbot' class. The method utilizes two parameters: 'irLedPin' and 'irReceivePin', which are the pin numbers connected to the IR LED and receiver on the microcontroller. This method returns an integer value representing the measured distance.

(2) Distance Variable Initialization

`int distance = 0;` This line initializes the distance variable to '0'. This variable is used to accumulate the distance values measured at various frequencies.

(3) Frequency Scan for Distance Measurement

`for(long f = 38000; f <= 42000; f += 500) { ... }` This is a for loop iterating over a range of frequencies, starting from 38000Hz, increasing by 500Hz each time, and stopping at 42000Hz. These frequencies are used to transmit and measure the IR signal reflections. The scan range of frequencies depends on the characteristics of the infrared sensor being used, indicating its sensitivity.

(4) Measuring Distance

`distance += irDetect(irLedPin, irReceivePin, f);` Inside the loop, the `irDetect` function is called with the current frequency `f`. The 'irDetect' method sends an IR signal using 'irLedPin', receives it with 'irReceivePin', and measures the distance, summing up all occurrences where the output is '1'. This value is then recorded in the distance variable.

(5) Return Total Measured Distance

After completing the loop, the accumulated total distance is returned. This value represents the outcomes of readings from all different frequencies, akin to the basic concept of radar.

Instead of just processing digital output signals to determine if an object is present with infrared sensors, the `irDistance()` method measures the distance to an object ahead. A sketch example that uses the `irDistance()` method for this purpose is introduced. To detect an object using two sensors, align the sensors' direction of focus towards a hypothetical object as shown in the diagram below.

The sketch results allow us to determine the spatial area between the closest distance at which object detection starts and the furthest distance at which it can detect, by observing the execution results of the sketch below. We can place a hypothetical object within the expected detection area and see if the robot scans and correctly perceives it.

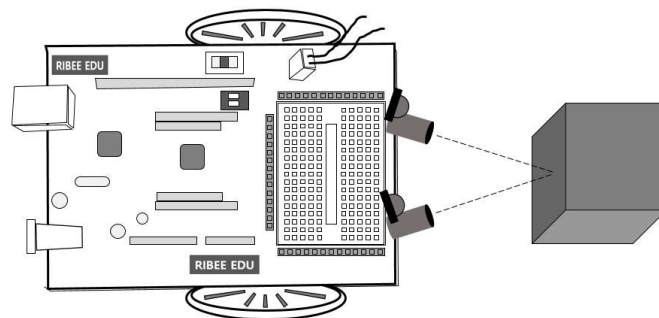


Figure 6.7: Schematic for Scanning Distance to an Object Ahead with Infrared Sensors

Concept Understanding for Beginners

The 'irDistance' feature acts like a tool that can measure distance by observing how IR light reacts at various frequencies.

The for loop is akin to conducting a series of experiments with slightly different settings (different frequencies) each time to gain a comprehensive understanding. Summing up all these measurements (`distance += irDetect(...)`) makes the overall distance measurement more stable and accurate. Finally, the function returns this total distance value. This method, using changes in IR signal response at

different frequencies depending on the measured distance, is a simple yet effective way to detect obstacles or for navigation purposes in robotics. Test your Arduino code to measure the distance to an object with the 'SelfAbot' class's irDistance method.

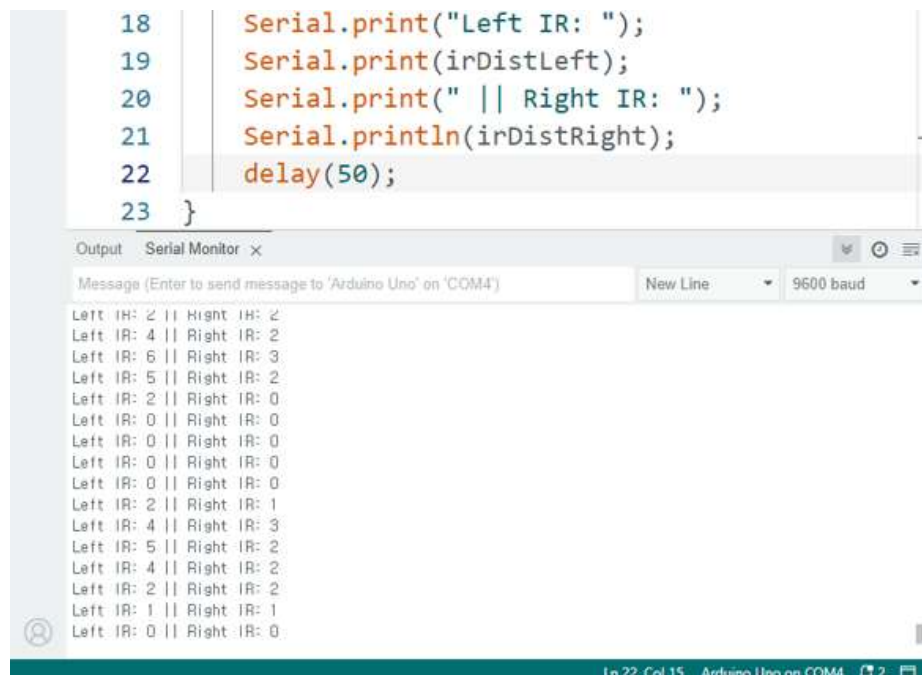
Ex6.3_irdistance_serialprint.ino

```
#include "SelfAbot.h"
SelfAbot abot;

void setup() {
  Serial.begin(9600);
  abot.setup();
}

void loop() {
  byte irLeftled = 9;
  byte irLeftreceiver = 10;
  byte irRightled = 2;
  byte irRightreceiver = 3;
  int irDistLeft = abot.irDistance(irLeftled, irLeftreceiver);
  int irDistRight = abot.irDistance(irRightled, irRightreceiver);

  Serial.print("Left IR: ");
  Serial.print(irDistLeft);
  Serial.print(" || Right IR: ");
  Serial.println(irDistRight);
  delay(50);
}
```



The image shows a screenshot of an Arduino IDE. The top part displays the code from the 'loop()' function, lines 18 through 23. The code prints the distance measured by the left and right infrared sensors. Below the code is the Serial Monitor window, which shows the output of the program. The output consists of multiple lines of text, each representing a single loop iteration. Each line contains two pairs of values: 'Left IR: [value] || Right IR: [value]'. The values range from 0 to 9. The Serial Monitor window also shows a message input field, a 'New Line' button, and a baud rate of 9600.

```
18 Serial.print("Left IR: ");
19 Serial.print(irDistLeft);
20 Serial.print(" || Right IR: ");
21 Serial.println(irDistRight);
22 delay(50);
23 }
```

Output Serial Monitor x

Message (Enter to send message to 'Arduino Uno' on 'COM4') New Line 9600 baud

Left IR: 2 || Right IR: 2
Left IR: 4 || Right IR: 2
Left IR: 6 || Right IR: 3
Left IR: 5 || Right IR: 2
Left IR: 2 || Right IR: 0
Left IR: 0 || Right IR: 0
Left IR: 0 || Right IR: 0
Left IR: 0 || Right IR: 0
Left IR: 2 || Right IR: 1
Left IR: 4 || Right IR: 3
Left IR: 5 || Right IR: 2
Left IR: 4 || Right IR: 2
Left IR: 2 || Right IR: 2
Left IR: 1 || Right IR: 1
Left IR: 0 || Right IR: 0

Figure 6.8: Serial Output for Distance Measurement with Infrared Sensors

In Figure 6.8, an output value of '9' indicates that the object is not detected even at the most sensitive frequency condition of 38,000 Hz, indicating a far distance. An output of '0' means the object is detected even at the least sensitive frequency of 42,000 Hz, indicating close proximity.

6.4 Scanning Surrounding Objects with Infrared Sensors

We will use two infrared sensors to measure the distance of objects in front of the robot, similar to implementing a radar system with infrared sensors. Using multiple sensors introduces various considerations, especially when sensors vary in type and quantity. Below are some methods to consider when integrating sensor data. The choice of method often depends on the specific requirements of the application, the characteristics of the data, and environmental conditions.

(Note) -----

Various Methods for Integrating Sensor Data:

(1) Weighted Average

Explanation: Assign different weights to each sensor's readings based on reliability or accuracy. For example, if one sensor is known to be more accurate at a certain distance, it can be given more weight.

Applications: Useful when sensor performance varies under different conditions.

(2) Median Filtering

Explanation: Instead of calculating an average, the median value of a set of readings is used. This method is particularly effective in reducing the impact of outliers or sporadic false readings.

Applications: Ideal for environments with high sensor noise levels or where sudden and incorrect readings are common.

(3) Mode Filtering

Explanation: Use the most frequently occurring value from a set of readings. This method can be useful when a consistent repeating value indicates stable measurements.

Applications: Useful in scenarios where sensors need to return repetitive distance readings.

(4) Kalman Filtering

Explanation: An advanced method that uses a series of measurements observed over time, including statistical noise and other inaccuracies, to produce estimates of unknown variables.

Applications: Widely used in applications needing real-time data processing, such as tracking and navigation systems.

(5) Sensor Fusion Algorithms

Explanation: Combine data from multiple sensors to increase accuracy and reduce uncertainty. This can be done using various algorithms, including Bayesian networks, machine learning models, or simple logical rules.

Applications: Effective in complex systems that measure different aspects of the environment with multiple sensors.

(6) Regression Analysis

Explanation: Fit a regression model to sensor data to predict values. This method can be used to identify relationships between readings from different sensors.

Applications: Useful when sensor readings are expected to follow a specific trend or pattern.

(7) Data Smoothing Techniques

Explanation: Apply smoothing techniques like moving average, exponential smoothing, or low-pass filters to reduce data noise and fluctuations.

Applications: Suitable for applications where data is expected to change gradually over time.

(8) Decision Trees or Rule-Based Systems

Explanation: Implement decision trees or rule-based systems that use sensor data to make decisions or classifications.

Applications: Useful in automated systems that need to make decisions based on sensor readings.

Introducing a method to detect objects in front of the robot using two installed infrared sensors. Instead of using the two sensors separately for left and right object detection, they will be used together to detect a single object. Arithmetic mean can be used for data from two sensors.

For two sensors to act as one, several considerations must be addressed:

(1) Field of View: If both sensors have similar fields of view and target the same object, averaging can help mitigate random errors or noise in individual sensor readings. Efficiency varies with the distance to the target and the angle at which sensors are mounted. Sensors should be positioned to ensure the most accurate readings at the desired point.

(2) Sensor Alignment: Both sensors must be aligned to target the same area or object. Misalignment can lead to inaccurate readings. Effective averaging requires aligning the sensors' views to converge on the point of interest. Accurate placement and orientation of sensors are necessary.

(3) Consistent Environmental Conditions: IR sensors can be affected by ambient light, surface reflectivity, temperature, etc., so both sensors must operate under the same environmental conditions.

(4) Overlapping Detection Range: In applications like obstacle detection, if there is overlap in the sensors' coverage, averaging can provide a more reliable estimate of the central distance.

To scan objects in front of the robot with fixed sensors, the robot must rotate in place. A method to rotate the robot in place for 360 degrees is introduced, along with considerations for calibrating the rotation angle and duration for accurate movement.

```
void rotateRobot(int angle) {  
    float factor = 10.0;  
    int duration = (int)(angle * factor);  
  
    abot.servoSpeed(100, 100); // Adjust speed as needed  
    delay(duration);  
  
    abot.servoSpeed(0, 0);  
}
```

To match the duration of the robot's rotation with the angle of rotation in place, an experimental coefficient (here referred to as a factor) must be found. This factor can vary slightly for each robot. Here are some tips for finding the factor value:

(1) Test Setup: Rotate the robot 360 degrees and measure the time it takes. If the robot takes 2000 milliseconds (2 seconds) to complete a 360-degree rotation, the factor would be $2000 / 360 \approx 5.56$. Therefore, the robot should rotate approximately 5.56 milliseconds per degree. The factor value '10.0' applied in the code should be replaced with the experimentally determined '5.56'.

Alternatively, you can repeatedly test to find the optimal factor value for a 360-degree rotation only. For example, if the servoSpeed speed value of the robot is 100, the factor might be around 5, but for a speed of 40, the factor could be about 10.

(2) Adjustment and Measurement: Change the rotation time until the robot rotates as close to 360 degrees as possible. Record the time it takes for this rotation.

(3) Calculate and Apply the Coefficient: Divide the time in milliseconds by 360 to calculate the coefficient (time per degree of rotation).

Introducing a sketch code for calibrating the factor value during the robot's rotation in place. Please upload the example below and perform the optimal calibration first. The result of this code ensures that when the robot is intended to rotate in place by a specified angle as a parameter, it does so accurately.

The speed at which the robot rotates must be kept constant while adjusting the factor value. If the rotation speed changes, the optimal factor value found before the change will no longer be valid. When calibrating with AA batteries, if servoSpeed(40, 40) is used, the factor value is approximately 10.

Upload the Ex6.4_rotateRobot_in_place.ino sketch and perform the in-place rotation calibration.

```
#include "SelfAbot.h"
#define INVALID_PIN 255;
const byte servoLeftPin = INVALID_PIN;
const byte servoRightPin = INVALID_PIN;

SelfAbot abot(servoLeftPin, servoRightPin);
float factor = 10.5; // Factor to convert angle to duration

void setup() {
  abot.setup();
  abot.servoAttachPins(13, 12);
  rotateRobot(360);
}

void loop() {
  // put your main code here, to run repeatedly:
}

// The rotateRobot function
void rotateRobot(int angle) {
  int duration = (int)(angle * factor);
  abot.servoSpeed(40, 40); // Adjust speed as needed
  delay(duration);
  abot.servoSpeed(0, 0); // Stop the robot after rotating
}
```

If the calibration of the robot's in-place rotation factor value has been completed, upload the code below to the Arduino to test how accurately it can detect objects around it.

Ex6.5_irdistance_point_rotation.ino

```
#include "SelfAbot.h"
#define INVALID_PIN 255;
const byte servoLeftPin = INVALID_PIN;
const byte servoRightPin = INVALID_PIN;

SelfAbot abot(servoLeftPin, servoRightPin);
float factor = 10.5; // Factor to convert angle to duration

void setup() {
  Serial.begin(9600);
  abot.setup();
  abot.servoAttachPins(13, 12);

  byte irLeftled = 9;
  byte irLeftreceiver = 10;
  byte irRightled = 2;
  byte irRightreceiver = 3;

  for (int angle = 0; angle < 180; angle += 10){
    rotateRobot(10);

    int irDistLeft = abot.irDistance(irLeftled, irLeftreceiver);
    int irDistRight = abot.irDistance(irRightled, irRightreceiver);
    int averageDistance = (irDistLeft + irDistRight) / 2;

    Serial.print("Angle: ");
    Serial.print(angle);
```

```

        Serial.print(" | Average Distance: ");
        Serial.println(averageDistance);
        delay(10);
    }
}

void loop() {

}

// The rotateRobot function

void rotateRobot(int angle) {
    int duration = (int)(angle * factor);
    abot.servoSpeed(40, 40); // Adjust speed as needed
    delay(duration);
    abot.servoSpeed(0, 0); // Stop the robot after rotating
}

```

The code introduced can be utilized in a scanning mode to survey the robot's surroundings while stationary. It rotates in place from 0 to 180 degrees in increments of 10 degrees to detect objects, and the process is repeated by the loop circuit. While there is the inconvenience of needing to use a USB cable since data is not transmitted wirelessly, it is hoped that this principle of searching for objects in front can be helpful in understanding.

Try using infrared sensors to implement how far you can judge the perspective of objects surrounding you. Place various types of objects around, large and small, far and near, and feel for yourself the objects identified by the robot. Through such work, you can create better robot actions.

6.5 Finding Nearby Objects by Scanning with Infrared Sensors

Until now, the operation of the Arduino robot has used only one mode of constant speed. And like using two infrared sensors in radar search mode, we will process data using the arithmetic mean method. Under these conditions, if the Arduino robot detects an object in front while advancing, it can stop its progress at that moment, switch to radar search mode, and then find and turn in the direction where objects are furthest away or absent and continue driving.

The following example introduces detecting objects placed between 0 ~ 180 degrees angles by the sensor and finding the direction of the closest object by the robot. Or it is a program that uses the robot to scan the surrounding environment and rotate the robot in the direction that detects the nearest object. The code is composed of several main parts, including robot setup, environmental scanning, and robot rotation control.

Example code will skip explanations on robot setup and sensor initialization and focus on the principle of finding objects with sensors and robot rotation.

(1) If an object is detected, the sensor value decreases, and the minimum value should be '0'. Therefore, the distance variable is initialized to the maximum value of 10, and the directionOfDistance variable is initialized to 0.

(2) The robot uses the infrared sensor to measure distance in each direction, increasing by 10° from 0° to 180°, using the in-place rotation method. If you wish to make the rotation angles denser, you can reflect smaller values in the increment of the loop and the parameters of the rotateRobot() function used to rotate the robot.

This code calculates the average distance obtained for each direction, and if this average distance is smaller than the currently detected minimum distance, the distance and direction are stored in variables.

(3) After all scans are completed, the `setTravelDirection` function is used to rotate the robot in the direction where the closest object was detected.

The function to rotate the robot in place is the `rotateRobot()` function, which can rotate in positive or negative angles. The `setTravelDirection` function is used to rotate the robot in the direction where the closest object was detected.

Ex6.6 `irscan_to_findObject.ino`

```
#include "SelfAbot.h"
#define INVALID_PIN 255;
const byte servoLeftPin = INVALID_PIN;
const byte servoRightPin = INVALID_PIN;

SelfAbot abot(servoLeftPin, servoRightPin);
float factor = 10.5;

void setup() {
  abot.setup();
  abot.servoAttachPins(13, 12);

  byte irLeftled = 9;
  byte irLeftreceiver = 10;
  byte irRightled = 2;
  byte irRightreceiver = 3;

  int distance = 10;
  int directionOfdistance = 0;

  for (int angle = 0; angle <= 180; angle += 10) {
    rotateRobot(10);

    int irDistLeft = abot.irDistance(irLeftled, irLeftreceiver);
    int irDistRight = abot.irDistance(irRightled, irRightreceiver);
    int averageDistance = (irDistLeft + irDistRight) / 2;

    if (averageDistance < distance) {
      distance = averageDistance;
      directionOfdistance = angle;
    }

    delay(100);
  }
  // Set robot's direction to the direction of max distance
  setTravelDirection(directionOfdistance);
}

void loop() {

}

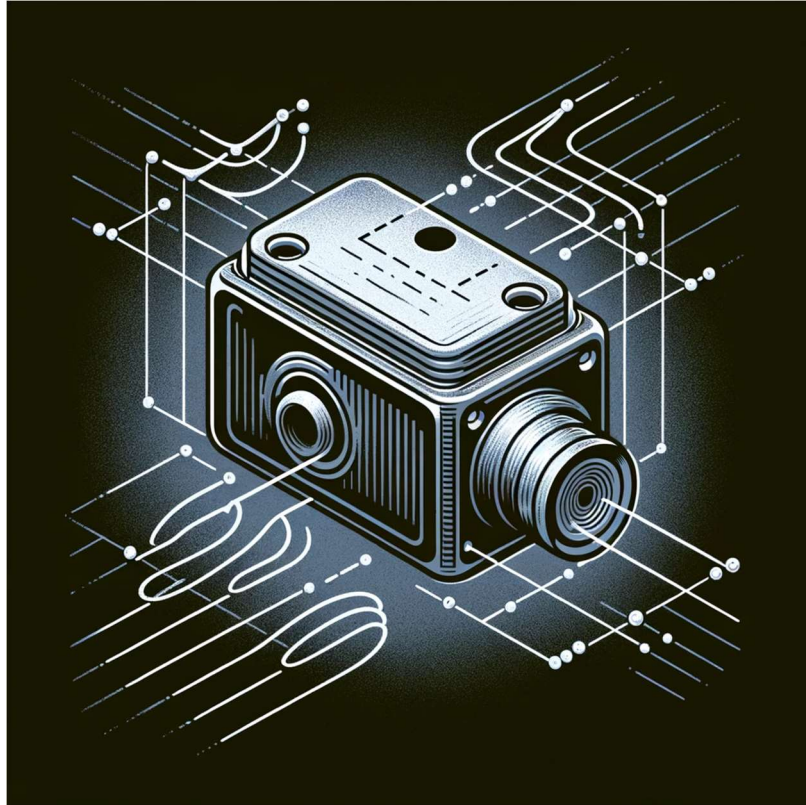
void rotateRobot(int angle) {
  if (angle >= 0) {
    int duration = (int)(angle * factor);
    abot.servoSpeed(40, 40);
    delay(duration);
    abot.servoSpeed(0, 0);
  } else { // reverse rotate
    int duration = (int)(abs(angle) * factor);
    abot.servoSpeed(-40, -40);
  }
}
```

```
        delay(duration);
        abot.servoSpeed(0, 0);
    }
}

void setTravelDirection(int angle) {
    // Rotate the robot to the desired direction
    rotateRobot(angle-180);
}
```

Chapter 7 Arduino Robot Equipped with Ultrasonic Sensor

- 7.1 Processing Ultrasonic Sensor Signals
- 7.2 Displaying the Radar Screen for Objects Ahead
- 7.3 Driving with an Ultrasonic Sensor Robot



In this chapter, the goal is to utilize Arduino's standard function, `pulseIn()`. In addition to the `pulseIn()` method, the 'SelfAbot' class also defines a `pulseOut()` method and a `pulseCount()` method.

This section explains how to operate ultrasonic sensors in a slightly different manner than the phototransistor light sensors and infrared transceiver sensors discussed in previous chapters.

Both light sensors and ultrasonic sensors are commonly used to detect obstacles and measure distances to objects. Here, we compare the features of ultrasonic sensors, which utilize "ultrasound" sound waves, and infrared sensors, which utilize the wavelength of "infrared" light, in Tables 7.1 and 7.2. Let's briefly compare these two types of sensors and explore handling ultrasonic sensor signals.

Table 7.1 Comparison of Various Features of Ultrasonic and Infrared Sensors

| | Ultrasonic Sensor | Infrared Sensor |
|-----------------------|---|--|
| Principle | Emits ultrasound waves (frequencies higher than human hearing) and measures the time it takes for the echo to bounce back from an object. | Uses infrared light to detect objects. Some measure the intensity of reflected light, while others (e.g., IR distance sensors) measure the angle or time of reflected light to determine distance. |
| Components | Typically consists of a transmitter (ultrasound emission) and a receiver (echo reception). | Includes an IR transmitter (typically an LED) and an IR receiver (e.g., photodiode). |
| Measurement Range | Can measure longer distances compared to IR sensors, typically from several centimeters to meters. Sound waves can travel farther than IR light waves. | Typically limited to shorter distances (often up to a few meters) and greatly varies with object reflectivity and environmental lighting conditions. IR light decreases more rapidly with distance than sound waves. |
| Accuracy | Can be very accurate but decreases with distance. Accuracy is also affected by the surface material and angle of the object. | Generally accurate at short distances but can be affected by the color and reflectivity of the object. Darker colors absorb more IR light, reducing reading accuracy. |
| Environmental Factors | Less affected by lighting conditions but can be influenced by temperature and wind. Soft materials or angular surfaces can absorb sound waves, leading to inaccurate readings. | Affected by ambient light, especially sunlight, which can interfere with the IR signal. Dust or fog can scatter IR light, requiring a clear IR path. |
| Application Areas | Commonly used in robot obstacle avoidance, tank level sensing, vehicle parking sensors, etc., where accurate distance measurement is required. Suitable for both indoor and outdoor applications, but performance may vary with environmental conditions. | Widely used for proximity detection, line-following robots, object detection/counting, TV remotes, and automatic faucets. Sensitive to sunlight and external light sources, so primarily used in indoor applications or controlled environments. |

(Note) -----

The Constant Trigger Signal and Variable Echo Signal of Ultrasonic Signals

The key to ultrasonic signal measurement is not that the signal is always the same, but that the time it takes for the signal to return (echo) varies with the distance to the reflecting object. More precisely, the signal transmitted from the trigger pin is actually constant, usually a quick HIGH pulse. However, the signal received at the echo pin varies depending on the time it takes for the pulse to travel to the object and back. The further the object, the longer the duration of the HIGH pulse.

One fact not to overlook is that the trigger signal is not the direct ultrasonic transmission signal but rather a signal sent to the component to induce it to generate the ultrasonic signal. Therefore, it's necessary to create and send the signal required by the component's datasheet.

The pulseOut() method is designed to generate a digital pulse of a specified duration on a certain pin. It is a simple yet powerful tool for controlling devices that require precise timing signals and specific pulse widths, like ultrasonic sensors.

We will examine the linearity and viewing angle characteristics of the two types of sensors.

Table 7.2 Linearity and Viewing Angle Characteristics of Ultrasonic and Infrared Sensors

| | Ultrasonic Sensor | Infrared Sensor |
|-----------|---|--|
| Linearity | <p>Linearity: Ultrasonic sensors generally have excellent linearity across their effective range. The time it takes for sound waves to reflect off an object and return is directly proportional to the distance, leading to a linear relationship under ideal conditions.</p> <p>Factors Affecting Linearity: Linearity can be influenced by the texture and angle of the target surface as well as environmental factors such as temperature and humidity, which can affect the speed of sound.</p> | <p>Linearity: The linearity of IR sensors can greatly vary depending on the type of IR sensor. Some IR distance sensors, using triangulation, can provide reasonable linear output across their operating range. However, sensors based on measuring the intensity of reflected light often exhibit non-linear responses, especially at close ranges.</p> <p>Factors Affecting Linearity: The reflectivity of the target object, surrounding lighting conditions, and the angle of incidence can significantly affect the linearity of IR sensor readings.</p> |

| | | |
|---------------|--|---|
| Viewing Angle | <p>Spread: Ultrasonic sensors typically have a wider beam angle compared to IR sensors, meaning they can cover a broader area but with less accuracy in pinpointing the exact location of objects within that area.</p> <p>Implications: A wider spread can lead to "false positives" where the sensor detects objects that are not directly in front but within the broader beam range.</p> | <p>Spread: IR sensors typically have a narrower field of view, allowing for more precise targeting of objects directly in front of the sensor. This makes them more suitable for detecting objects in a specific and narrow area.</p> <p>Implications: A narrower spread reduces the likelihood of detecting off-axis objects but also limits the detection area, requiring more sensors or sensor movement to cover the entire area.</p> |
|---------------|--|---|

The pulseCount() method is particularly useful for counting the number of rotations of a wheel with a sensor attached or the number of objects passing through a sensor in a specified time frame. One can consider using pulseCount() to count the rotations of a wheel.



Figure 7.1: Types of pins on ultrasonic sensors from different manufacturers (a) 3-pin (b) 4-pin

7.1 Processing Ultrasonic Sensor Signals

To process ultrasonic sensor signals, we use the pulseOut, pulseIn methods of the 'SelfAbot' class. A simple example using the pulseOut, pulseIn, and pulseCount methods is introduced. These methods are commonly used with various sensors and actuators, such as ultrasonic sensors or encoders. Ultrasonic sensors typically emit a pulse (pulseOut) and then measure the time it takes for the pulse to return (pulseIn), calculating the measurement distance using the measured time delay.

Let's take a closer look at the pulseIn() method.

- (1) Trigger pin: Sends a short HIGH pulse to start the measurement.
- (2) Echo pin: Receives the HIGH pulse after it bounces off an object.

4-pin HR04 Ultrasonic Sensor

An example of measuring the distance to an object using a 4-pin HR04 ultrasonic sensor. Connect the ultrasonic sensor's trigger pin to a digital pin on the Arduino (e.g., pin 7). Connect the ultrasonic sensor's echo pin to another digital pin (e.g., pin 8).

Ex7.1_HR04_ultrasonic_distance.ino

```
#include "SelfAbot.h"

SelfAbot abot;
const byte triggerPin = 7; // Ultrasonic sensor trigger pin
const byte echoPin = 8; // Ultrasonic sensor echo pin

void setup() {
  abot.setup();
  pinMode(triggerPin, OUTPUT); // Set the trigger pin as an output
  pinMode(echoPin, INPUT); // Set the echo pin as an input
  Serial.begin(9600);
}
```



```

void loop() {
  abot.pulseOut(triggerPin, 10); // 10 microseconds pulse
  unsigned long duration = abot.pulseIn(echoPin, HIGH);

  // Calculate distance (in centimeters or inches)
  // Speed of sound = 34300 cm/s (in air), so time for there and back is duration
  // Distance = (speed of sound * time) / 2
  float distanceCm = duration * 0.0343 / 2;
  float distanceInch = duration * 0.0135 / 2;

  // Print the distance
  Serial.print("Distance: ");
  Serial.print(distanceCm);
  Serial.print(" cm, ");
  Serial.print(distanceInch);
  Serial.println(" inches");

  delay(1000); // Wait for a second before the next measurement
}

```

`pulseOut(triggerPin, 10);` sends a short pulse (10 microseconds) to trigger the ultrasonic sensor. `pulseIn(echoPin, HIGH);` waits for the pulse to return and measures its duration. The distance is calculated based on the pulse duration. Since the sound wave travels to the object and back, it is divided by 2.

This code provides a basic example of using pulse-related methods in robots or sensors. You can adjust and expand this example according to the specific requirements of your project.

3-pin Parallax Ultrasonic Sensor

A 3-pin ultrasonic sensor can create the same effect. Parallax's 3-pin ultrasonic sensor handles both transmitting (trigger pin) and receiving (echo pin) ultrasonic signals with a single pin. Therefore, in the code for the 4-pin method, you can apply the same pin number for both.

```

long duration, distanceCm;
pinMode(pingPin, OUTPUT);
digitalWrite(pingPin, LOW);
delayMicroseconds(2);
digitalWrite(pingPin, HIGH);
delayMicroseconds(5);
digitalWrite(pingPin, LOW);

pinMode(pingPin, INPUT);
duration = pulseIn(pingPin, HIGH);
distanceCm = duration * 0.0343 / 2;

```

The above code is specifically for measuring distance using the Parallax 3-pin ultrasonic sensor. It utilizes the Arduino library's standard function, `pulseIn()`, to measure the time (echo time) it takes for the ultrasonic pulse to travel to an object and return. Let's examine the code and role of `'pulseIn()'`.

`pinMode(pingPin, OUTPUT);` sets the `pingPin` as an output. This pin is connected to the trigger input of the ultrasonic sensor.

Generating the ultrasonic pulse:

`digitalWrite(pingPin, LOW); DelayMicroseconds(2);` ensures a clean transition from LOW to HIGH pulse. Starting in a LOW state ensures the line is clear. Setting the pin to LOW before sending a HIGH pulse stabilizes the signal, preventing false triggering due to residual charge or noise on the pin. The LOW state "resets" the pin to a known state. `'delayMicroseconds(2)'` provides a short buffer time after setting the pin to LOW to ensure the HIGH pulse is clearly transmitted. This brief delay helps differentiate the pulse from previous activities or noise on the pin.

`digitalWrite(pingPin, HIGH); DelayMicroseconds(5);` generates a short HIGH pulse (5 microseconds).

This pulse triggers the sensor to emit ultrasonics.
digitalWrite(pingPin, LOW); returns the pin to LOW. The ultrasonics are now traveling through the air. The concept of directly replacing this sequence with a pulseOut does not exist in the standard Arduino library.

Receiving the echo:

pinMode(pingPin, INPUT); changes the pingPin mode to INPUT to receive the echo signal.

Measuring the ultrasonic travel time:

duration = pulseIn(pingPin, HIGH); this is the crucial part. 'pulseIn()' measures the time between transmitting the ultrasonic pulse and receiving the echo. It waits until the pin goes HIGH (echo starts), starts timing, then waits until the pin goes LOW (echo ends), and stops timing.

The pulseIn() function measures the length of a pulse (in microseconds) on a specified pin. This function takes two arguments: the pin number and the type of pulse to measure (HIGH or LOW). In the code above, pulseIn(pingPin, HIGH) measures the duration of the echo pulse. The pulse begins when the pin goes HIGH (echo received) and ends when the pin goes back to LOW (echo ends).

The 'duration' variable records the time in microseconds between the transmission and reception of the ultrasonic pulse. To convert this time into distance, the speed of sound in air (about 343 meters per second, i.e., 0.0343 cm/ μ s) is typically used, considering that the sound has to travel to the object and back. Thus, the distance to the object is (flight time * 0.0343) / 2 cm.

The pulseOut method of the 'SelfAbot' class can be used to rewrite the above code:

```
long duration, distanceCm;
abot.pulseOut(pingPin, 5);
duration = abot.pulseIn(pingPin, HIGH);
distanceCm = duration * 0.0343 / 2;
```

I hope this explanation helps in understanding the operation of an ultrasonic sensor.

Below, I'll introduce a .ino code that measures the distance to an object using the Parallax 3-pin ultrasonic sensor.

Ex7.2 parallax_3pin_ultrasonic_distance.ino

```
#include "SelfAbot.h"
SelfAbot abot;
const byte pingPin = 7; // sensor trigger/echo pin

void setup() {
  abot.setup();
  Serial.begin(9600);
}
void loop() {
  abot.pulseOut(pingPin, 5); // 5 microseconds pulse
  unsigned long duration = abot.pulseIn(pingPin, HIGH);
  // Calculate distance (in centimeters or inches)
  // Speed of sound = 34300 cm/s (in air), so time for there and back is duration
  // Distance = (speed of sound * time) / 2
  float distanceCm = duration * 0.0343 / 2;
  float distanceInch = duration * 0.0135 / 2;

  // Print the distance
  Serial.print("Distance: ");
  Serial.print(distanceCm);
  Serial.print(" cm, ");
  Serial.print(distanceInch);
  Serial.println(" inches");
  delay(1000); // Wait for a second before the next measurement
}
```

7.2 Displaying the Radar Screen for Objects Ahead

We plan to use the Processing tool to visualize data detected by an ultrasonic sensor while it rotates 180 degrees. Instead of utilizing the method where the Arduino robot pivots in place, we will introduce a method that involves using a standard servo motor to rotate only the ultrasonic sensor.

Ex7.3_ultrasonic_radar_scan.ino

```
#include "SelfAbot.h"

SelfAbot abot;
const byte servoPin = 10;
const byte pingPin = 7; // sensor trigger/echo pin
int distance;

void setup() {
  abot.setup();
  abot.servoAttachAngle(servoPin);
  Serial.begin(9600);
}

void loop() {
  for (int i = 0; i <= 180 ; i++) {
    abot.servoAngle(i);
    delay(15);
    distance = calculateDistance();

    Serial.print(i);
    Serial.print(",");
    Serial.print(distance);
    Serial.print(".");
  }
  for (int i = 180; i > 0 ; i--) {
    abot.servoAngle(i);
    delay(15);
    distance = calculateDistance();

    Serial.print(i);
    Serial.print(",");
    Serial.print(distance);
    Serial.print(".");
  }
}

int calculateDistance(){
  abot.pulseOut(pingPin, 5);
  unsigned long duration = abot.pulseIn(pingPin, HIGH);
  int distanceCm = (int)duration * 0.034/2;
  return distanceCm;
}
```

The Arduino serial output screen only displays the angle and distance values. We want to visually represent the objects detected by the ultrasonic sensor in front of the screen more vividly. By using the Processing programming tool, it's easy to transform distance/angle data into visual image data. Since we won't introduce the entire usage of the Processing tool here, we'll only present how to visually represent the values measured by the ultrasonic sensor from the Arduino board to the Processing IDE screen.

To install the Processing program, download and install it from the Processing website (<https://processing.org/download>). After installing the Processing program, let's see how the servo

motor and ultrasonic sensor operated by Arduino code and the sensor output values from Arduino are connected to the Processing program.

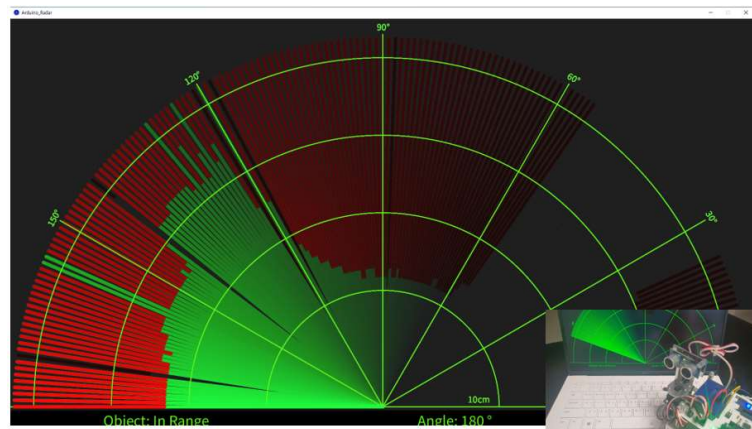


Figure 7.2: Ultrasonic sensor radar screen mounted on a robot

Connecting Processing with Arduino output values is very simple. By looking at the interface, using Processing's `serialEvent()` function allows receiving angle and distance values measured by the ultrasonic sensor from the Arduino board to the Processing IDE. Data is read from the serial COM port and entered into `iAngle` and `iDistance` variables. Lastly, these variables are used to draw the radar, lines, detected objects, and some text. When running the attached Processing code, users only need to provide the correct COM port where Arduino and the computer are connected to the Processing code.

Use the link provided below for the Processing source code:

<https://www.superkitz.com/arduino-radar-using-ultrasonic-sensor-hc-sr04-diy-kit/>

The Processing source code will also be left in the appendix for reference.

7.3 Driving with an Ultrasonic Sensor Robot

This example implements the simplest robot action that detects obstacles in front of the robot using an ultrasonic sensor and avoids them if detected. The code implementation method is introduced in two types: static and dynamic exploration.

The first example introduced operates in static exploration mode. The essence of static exploration is to explore the surroundings while stationary, determine a safe angular range based on a preset standard, and then move. The principle of code operation is as follows:

The `Setup` function starts serial communication and sets up the `SelfAbot` library. Servo motors connected to the two wheels and a standard servo motor with an ultrasonic sensor attached are prepared with the `attach()` function.

The `Loop` function repeats continuously, and its main functions are as follows:

- (1) The `calculateDistance` function measures the distance to objects using the ultrasonic sensor.
- (2) The `isSafeAreaDetected` function checks if the distance detected at the scan angle is beyond a safe distance and if there is a continuous absence of obstacles in an angular range of more than 40 degrees.
- (3) The `getSafeAreaCenterAngle` function finds the widest angle area with the least number of obstacles among several distance measurements and returns the middle angle of that area.
- (4) The `setTravelDirection` function rotates the robot in the desired direction. It calculates the rotation angle from the current angle to the target angle and rotates the robot in place.
- (5) The `alignSensorToFront` function aligns the ultrasonic sensor of the robot to face forward.

This process measures the distance to objects using the ultrasonic sensor, finds a safe area, and rotates the robot.

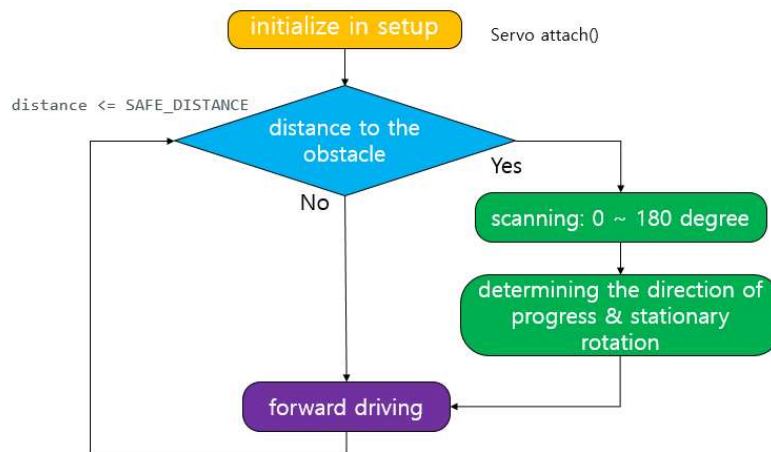


Figure 7.3: Logic flowchart for static exploration with an ultrasonic sensor

The Ex7.4 code below introduces an example of robot action that statically explores using an ultrasonic sensor. This method is mainly used when the environment is static or changes little. This static exploration example uses the ultrasonic sensor to explore the surrounding environment and adjust the angle to identify a safe area.

For reference in implementing the example code, see Example Ex3.2 on standard servo motor operation in Chapter 3 and Example Ex6.4 or Ex6.5 on robot pivoting in place in Chapter 6. The `isSafeAreaDetected` and `getSafeAreaCenterAngle` functions in the code example use the `SafeArea`, as shown in Figure 7.4, which is the range where no obstacles exist within a safe distance in the robot's frontal area.

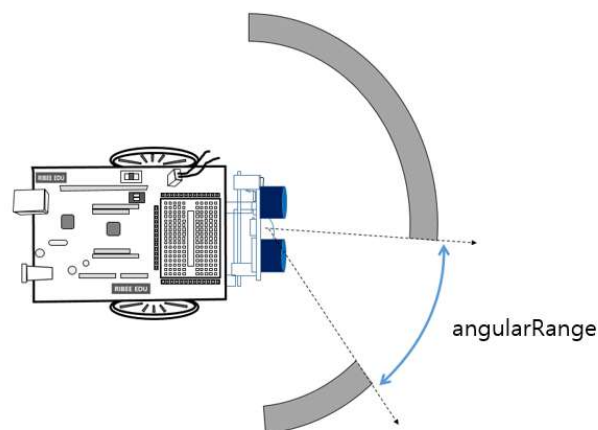


Figure 7.4: Angular range considered as a safe area by the ultrasonic sensor

Using this range value appropriately allows determining whether the robot can pass through obstacles. It needs to be set according to the user's robot operation environment.

Ex7.4 Ultrasonic_sensor_search_driving.ino

```

#include "SelfAbot.h"

#define TRIGGER_PIN 7
#define ECHO_PIN 7
#define MAX_DISTANCE 200 // max distance (cm)
#define SAFE_DISTANCE 20 // safe distance (cm)
  
```

```

#define MAX_DETECTIONS 40

#define INVALID_PIN 255
const byte servoLeftPin = INVALID_PIN;
const byte servoRightPin = INVALID_PIN;

SelfAbot abot(servoLeftPin, servoRightPin);
float factor = 10.5;

struct Detection {
    int angle;
    unsigned int distance;
};

Detection detections[MAX_DETECTIONS];
int detectionCount = 0;
int notdetectedAngularrange = 40; // no obstacles angular range

void setup() {
    Serial.begin(9600);
    abot.setup();
    abot.servoAttachPins(13, 12);
    abot.servoAttachAngle(10);
}

void loop() {
    int distance = calculateDistance();

    if(distance > 0 && distance <= SAFE_DISTANCE) {
        abot.servoSpeed(0, 0);

        detectionCount = 0;
        for(int angle = 0; angle <= 180; angle += 5) {
            abot.servoAngle(angle);
            delay(500);
            distance = calculateDistance();

            if (detectionCount < MAX_DETECTIONS) {

                == Code omitted ==
            }
        }
    }
}

```

For the rest of the code, please refer to the Arduino SelfAbot library examples!

Unlike the previously introduced example, there is a method that combines (1) scanning to store distance data for each angle and (2) analyzing the stored data to align the robot in a direction without obstacles into one action. This method allows for analysis while scanning data and, based on this, makes real-time decisions on the robot's direction of movement. This approach is referred to as dynamic exploration.

Compared to the previously introduced static exploration, dynamic exploration has the following differences:

(1) Real-time data analysis: While the previous code analyzed the data after scanning was complete to decide on a safe direction of movement, the dynamic exploration method analyzes data in real time while moving the angle, allowing the robot to immediately decide and rotate to a safe direction if conditions are met.

(2) Integration of scanning and analysis: In the dynamic exploration method, the robot performs scanning and analysis simultaneously while moving angles. This enables real-time exploration of the environment and immediate reactions.

(3) Efficiency: Since the dynamic exploration method analyzes data while scanning, it can take less time to decide on a direction of movement. Additionally, by omitting the process of deciding on the movement direction after scanning is complete, time is saved.

The following example is a modified version of the previously introduced Ex7.4 example. Unlike the previous example, which separated the process of scanning the entire 0 ~ 180-degree angle range to find one or more sections without obstacles and then executing robot movements in the next order, the example introduced below integrates the scanning work and robot rotation actions. To do this, data collection is immediately followed by data analysis, and as soon as a section without obstacles that meets the conditions is found, the robot is rotated immediately. By processing data in this way, the robot can operate more flexibly.

This code is a more efficient and concise example than the previous code. Upload and run the sketch example Ex7.5 Ultrasonic_sensor_Integrated_scanning_rotation below.

```
#include "SelfAbot.h"

#define TRIGGER_PIN 7
#define ECHO_PIN 7
#define MAX_DISTANCE 200 // max distance (cm)
#define SAFE_DISTANCE 20 // safe distance (cm)

#define MAX_DETECTIONS 40

#define INVALID_PIN 255
const byte servoLeftPin = INVALID_PIN;
const byte servoRightPin = INVALID_PIN;

SelfAbot abot(servoLeftPin, servoRightPin);
float factor = 10.5;

struct Detection {
    int angle;
    unsigned int distance;
};

Detection detections[MAX_DETECTIONS];
int detectionCount = 0;
int notdetectedAngularrange = 40; // Angle range where obstacles do not exist

void setup() {
    Serial.begin(9600);
    abot.setup();

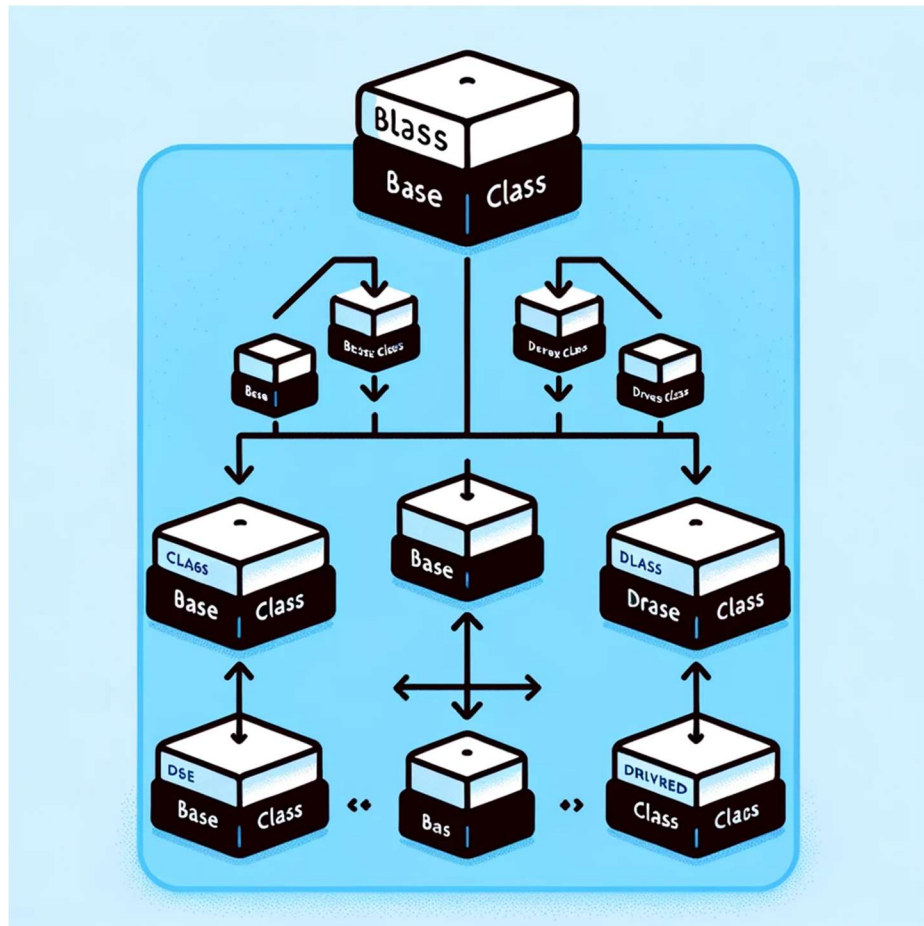
    == Code omitted ==
    For the rest of the code, please refer to the Arduino SelfAbot library examples!
```

Chapter 8 Specialized Robots with Class Inheritance

8.1 Creating Subclasses with 'SelfAbot' as the Superclass

8.2 How to Write an .ino File Using an Inheritance Class

Appendix: 'EnhancedSelfAbot.zip' Library Header and Source Files



Class inheritance is a concept in object-oriented programming (OOP) where a class (child class or subclass) can inherit properties and behaviors (methods and variables) from another class (parent class or superclass). This mechanism allows the creation of a new class based on an existing class, enhancing code reusability. Ultimately, class inheritance is a powerful tool in OOP for organizing code into a hierarchical structure for easy reuse, extension, and management.

While class inheritance deals with vertical relationships ("is-a") between classes, friend declarations address horizontal class access, and composition or containment relationships ("has-a") involve one class including another as a member, which will not be discussed here.

Let's examine the characteristics of class inheritance:

(1) Code Reuse: Through inheritance, a new class can inherit the properties and methods of an existing class, meaning there's no need to rewrite already written code. These can be extended or modified in the subclass.

(2) Hierarchical Relationship: Inheritance creates a hierarchical relationship between classes. The parent class (also called base or superclass) provides general definitions that can be shared among several subclasses (derived classes or child classes).

- (3) Method Overriding: A subclass can override methods from the parent class to provide specific implementations different from the parent class.
- (4) Extensibility: Without affecting the parent class or other classes that inherit from it, new functionalities can be added to the subclass, improving code extensibility.
- (5) Polymorphism: Inheritance supports the concept of polymorphism, where a subclass instance can be treated as an instance of the parent class. This is particularly useful in scenarios where a generalized type is desired for various specific instances.

A simple example illustrating class inheritance:

```
class Vehicle { // Parent class
public:
    void startEngine() {
        // Code to start the engine
    }
};
class Car : public Vehicle { // Child class
public:
    void openTrunk() {
        // Code to open the trunk
    }
};
```

In this C++ example, Car is a subclass inherited from the Vehicle superclass. This means an object of the Car class can use both the startEngine method (inherited from Vehicle) and the openTrunk method (specific to Car).

In summary, when used properly, class inheritance is a powerful tool in object-oriented programming (OOP) that can create elegant and efficient code structures. However, it's important to use it carefully. It's recommended for beginners to start with simple inheritance cases to grasp the concept before moving on to more complex scenarios.

8.1 Creating a Subclass with 'SelfAbot' as the Superclass

In this chapter, we'll create a subclass called 'EnhancedSelfAbot.' This class sets 'SelfAbot' as its superclass and inherits from it. We'll introduce a use case below. Among the codes we've discussed, one that wasn't included in the 'SelfAbot' base class will now be applied in this subclass. The 'EnhancedSelfAbot' subclass includes various methods, one of which is the "gradual acceleration and deceleration" introduced in section 4.4, now represented in the subclass.

```
----- EnhancedSelfAbot.h -----
#include "SelfAbot.h"

class EnhancedSelfAbot : public SelfAbot {

public:
    EnhancedSelfAbot() : SelfAbot(), _currentLeftSpeed(0), _currentRightSpeed(0) {}
    EnhancedSelfAbot(byte servoLeftPin, byte servoRightPin) : SelfAbot(servoLeftPin, servoRightPin),
    _currentLeftSpeed(0), _currentRightSpeed(0) {}
    void gradualServoSpeed(int targetLeftSpeed, int targetRightSpeed);

private:
    int _currentLeftSpeed = 0; // Current speed of the left servo
    int _currentRightSpeed = 0; // Current speed of the right servo
};
----- EnhancedSelfAbot.cpp -----
void EnhancedSelfAbot::gradualServoSpeed(int targetLeftSpeed, int targetRightSpeed) {
    int step = 5; // Speed change step setting
```

```

while (_currentLeftSpeed != targetLeftSpeed || _currentRightSpeed != targetRightSpeed) {
  unsigned long currentMillis = millis();

  if (_currentLeftSpeed < targetLeftSpeed) {
    _currentLeftSpeed += step;
  } else if (_currentLeftSpeed > targetLeftSpeed) {
    _currentLeftSpeed -= step;
  }
  if (_currentRightSpeed < targetRightSpeed) {
    _currentRightSpeed += step;
  } else if (_currentRightSpeed > targetRightSpeed) {
    _currentRightSpeed -= step;
  }
  servoSpeed(_currentLeftSpeed, _currentRightSpeed);
  while (millis() - currentMillis < 10) {
    // Small delay until 10ms have passed
  }
}
}

```

The code addresses potential issues by improving it in the subclass "EnhancedSelfAbot," which can be examined directly in the file.

8.2 How to Write an .ino File Using an Inheritance Class

Using the "EnhancedSelfAbot" subclass is similar to using the "SelfAbot.zip" library; you add the "EnhancedSelfAbot.zip" library to the Arduino IDE program and use it as a separate library. If "SelfAbot.zip" library was installed, the "EnhancedSelfAbot" subclass should already be prepared in the Arduino library folder for use.

```

EnhancedSelfAbot abot;
void setup() {
  abot.servoAttachPins(LEFT_SERVO_PIN, RIGHT_SERVO_PIN); //Set servo pins
  abot.gradualServoSpeed(100, 100); // Start gradual speed adjustment
}
void loop() {
  // Main loop code...
}

```

You can write and experiment with the above example code in the Arduino .ino file using the "EnhancedSelfAbot" subclass.

8.2.1 Straight Driving Correction Exercise

Chapter 4 used a method of directly modifying the deviationFactor value in the .ino code for the robot's straight driving. Here, before running the code to correct the deviationFactor, a subclass method calibratedDeviationFactor(int deviatedDistance) is introduced for measuring the distance deviated from straight driving and correcting it as a parameter.

```

void EnhancedSelfAbot::calibratedDeviationFactor(int deviatedDistance) {
  if (deviatedDistance != 0) {
    float K = 0.005;
    _deviationFactor += K * (float)deviatedDistance;
  } else {
    // blank
  }
}

```

Considerations for measuring with this method include:

(1) The parameter `deviateDistance` of this method is the distance (in millimeters) that the robot has deviated perpendicularly from a straight line when it was supposed to travel straight. Since the degree of deviation from the straight line varies with the speed of the robot, it is advisable to adjust the speed to a moderate level for the practice. A speed close to medium, 40, is recommended for the practice.

(2) Use a measuring ruler to travel a certain distance, then measure how much the robot has deviated from the straight line, input this value into the parameter of the above method, and run it again. If it is possible to input the moment or distance of deviation from the straight line as sensor data, this process can be automated. The calibration practice example below does not reflect the value obtained from the previous practice because the `_deviationFactor` value is always initialized.

(3) The more the calibration work is repeated, the closer the robot will get to driving straight. If the results deviate more than expected, you may need to modify the constant value (e.g., 'K' value) used in the code. (The current 'K' value is not based on precise experimental results.)

(4) If the robot drives straight after repeating the calibration routine, the `_deviationFactor` value corresponding to each driving condition is newly updated and saved, and the `_deviationFactor` value is stored as a class variable to be used as the final calibration value. (For reference, the `driveStraight(leftSpeed, rightSpeed)` method can be used interchangeably with the `servoSpeed(leftSpeed, rightSpeed)` for the same purpose.)

When executing the calibration routine to make the robot drive straight using the example code introduced below, there is a limitation that the practice must be completed without disconnecting the USB cable. If you want to set a longer distance for calibration, you can transmit serial input data from the PC to the robot and receive the output results of the calibration work via Bluetooth or Wi-Fi wireless communication.

The exercise sequence for Example Ex8.1 is briefly introduced as follows:

(1) Connect the servo motor pins of the Arduino robot's left/right wheels to pins 13 and 12, respectively.

(2) Upload the example sketch and align the robot in the space for the exercise.

(3) After uploading the sketch, open the serial monitor window in the IDE program and enter 's' to make the robot drive forward for the scheduled time.

(4) After the robot has moved forward for a certain time and then stops, when the message "Please measure the deviation distance and enter it (mm): " is displayed on the serial monitor, move the robot back to its original starting position and enter the distance it deviated from the straight line in millimeters in the serial monitor window, and press the enter key. (This series of user actions must be completed within 30 seconds. Otherwise, you have to start over from the beginning.)

(※ The deviation distance should be entered in millimeters. To correct the robot to the left, use a '-' sign before the deviation distance, and to correct it to the right, use a '+' sign.)

(5) Once the calibration work is completed, the `_deviationFactor` value used for calibration will be displayed. This value remains unchanged in the class member variable unless the calibration work is repeated again.)

Ex8.1_straightDrive_calibration_practice.ino

```
#include "EnhancedSelfAbot.h"

const byte servoLeftPin = 255;
const byte servoRightPin = 255;
EnhancedSelfAbot abot(servoLeftPin, servoRightPin);

void setup() {
  Serial.begin(9600);
  abot.setup();
  abot.servoAttachPins(13, 12); // Replace with actual servo pins
```

```

Serial.println("Calibration Practice for SelfAbot");
Serial.println("Place the robot and press 's' to start moving.");
}

void loop() {
  if (Serial.available() > 0) {
    char command = Serial.read();
    if (command == 's') {
      performCalibration();
    }
  }
}

void performCalibration() {
  int servospeed = 40;
  int delaytime = 5000; // Time to make the robot go straight (5 seconds)
  int measuredDevidistance; // Save the measured deviation distance (unit : mm)
  unsigned long startTime = millis(); // Start time for timeout
  unsigned long timeout = 30000; // 30 second timeout

  abot.resetDeviationFactor();

  Serial.println("Going straight. Please wait...");
  abot.driveStraight(servospeed, -servospeed); // Straight
  delay(delaytime); // Go straight for the specified delay time
  abot.servoSpeed(0, 0); // Stop the robot

  Serial.println("Please measure the deviation distance and enter it (mm): ");
  delay(500); // Short delay to ensure the user sees the message

  // Clear the serial buffer to remove any unwanted or leftover data
  while (Serial.available() > 0) {
    Serial.read();
  }

  // Wait for new input
  while (Serial.available() == 0) {
    // Check if timeout has been reached
    if (millis() - startTime > timeout) {
      Serial.println("Input waiting time has expired. Please restart calibration.");
      return; // exit the function when timeout is reached
    }
    delay(100); // Small delay to prevent the loop from running too fast
  }

  measuredDevidistance = Serial.parseInt(); // Read the entered value...

  // Check for '0' input to exit without performing corrections
  if (measuredDevidistance == 0) {
    Serial.println("Exiting calibration without correction as the input is '0'.");
    return; // exit the function as '0' was entered
  }

  Serial.print("Measured deviation distance: ");
  Serial.print(measuredDevidistance);
  Serial.println("mm");

  // Convert mm to cm for calibratedDeviationFactor if necessary
  float measuredDevidistanceCm = measuredDevidistance / 10.0;

```

```

abot.calibratedDeviationFactor(measuredDevDistanceCm);

abot.driveStraight(servoSpeed, -servoSpeed); // Straight
delay(delayTime); // Go straight for the specified delay time
abot.servoSpeed(0, 0); // Stop the robot

// Display the final corrected _deviationFactor value
float finalDeviationFactor = abot.getDeviationFactor();
Serial.print("Final corrected _deviationFactor: ");
Serial.println(finalDeviationFactor);

Serial.println("Calibration is complete. Test the robot's movements again.");
}

```

The operation principle and sequence of **the performCalibration() function** are as follows:

Initial values of variables (such as the function's servoSpeed, delayTime, etc.) are adjustable, so set them to suitable values, upload the sketch, and open the serial monitor window. You will see the following message ("**Place the robot and press 's' to start moving.**"). Position the robot's 3-point switch to position 2, enter the letter 's' in the serial monitor, and press Enter.

The robot calls abot.resetDeviationFactor(); to initialize the _deviationFactor value to 0, travels for delayTime, and then stops. The serial monitor will display a message asking you to enter the distance the robot deviated from a straight line in millimeters.

Going straight. Please wait...

Please measure the deviation distance and enter it (mm):

Measure the distance by which the robot deviated from a straight line, enter the approximate deviation distance in millimeters in the serial monitor, and press Enter. Pressing Enter corrects the _deviationFactor value and executes the abot.driveStraight(servoSpeed, -servoSpeed); function. At this point, you must return the robot to its original starting position since the robot will start moving again.

Your entered value will be displayed on the screen as follows:

Measured deviation distance: ** mm

Final corrected _deviationFactor: ***

Calibration is complete. Test the robot's movements again.

The calibration task for straight-line driving is now complete.

*The _deviationFactor: *** value found in this calibration task is an essential member variable for using the driveStraight() method in the subclass.*

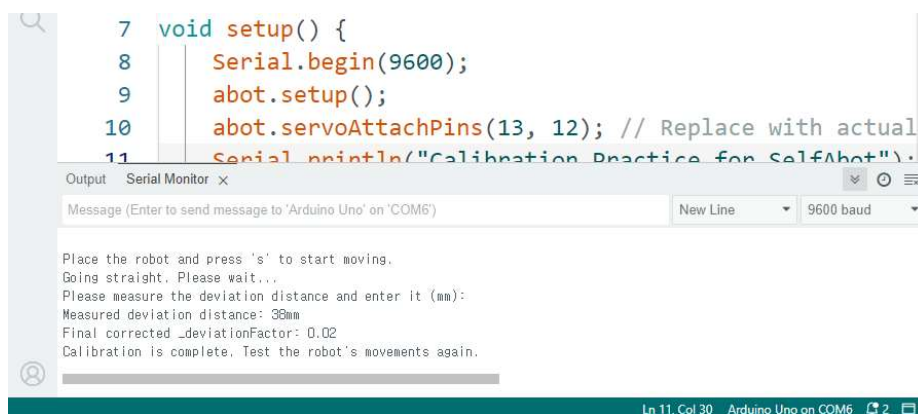


Figure 8.1: Calibration example for the robot's straight-line driving

Remember or note down the final calculated correction factor (`_deviationFactor`) used in the robot you are working with for future use of the subclass.

During the practice, you may experience exceptions, such as not entering the deviation distance within 30 seconds or entering a deviation distance of '0'. You can start over from the beginning. When executing the `abot.driveStraight(servospeed, -servospeed);` function in another practice, all previously stored `_deviationFactor` values are erased.

When starting a new sketch example, you can set and use the `_deviationFactor` value found in the previous calibration function execution in the sketch's `setup` function using the `setter()` method.

(Note) -----

Why use getter() and setter() methods?

The `getter()` and `setter()` methods in the 'EnhancedSelfAbot' library are as follows:

```
float EnhancedSelfAbot::getDeviationFactor() const {
    return _deviationFactor;
}
void EnhancedSelfAbot::setDeviationFactor(float deviationFactor) {
    _deviationFactor = deviationFactor;
}
```

The `_deviationFactor` member variable exists in the class's private area, maintaining the principles of encapsulation and information hiding in object-oriented programming.

8.2.2 Gradual Speed Control Practice

This .ino file example code practices the test drive of 'Arduino robot's gradual speed change' using the `gradualServoSpeed` method. Along with the `servoSpeed()` method, the `gradualServoSpeed` method uses the `driveStraight` method instead of `servoSpeed()` method to address the issue that the speed difference between the robot's left and right wheels becomes noticeably pronounced at speed values below 100, causing the robot to veer significantly to the left or right even during straight-line driving.

The `gradualServoSpeed` method reflects the values found in the previous correction work in the `setup` function using the `setter()` method. Add `abot.setDeviationFactor(-0.05);`, upload the sketch, and proceed. (Note: The `-0.05` value shown in the example may vary for each user, so you must find and use your value.)

Ex8_2_enhanced_gradual_servospeed.ino

```
#include "EnhancedSelfAbot.h"

const byte servoLeftPin = 255;
const byte servoRightPin = 255;
EnhancedSelfAbot abot(servoLeftPin, servoRightPin);

void setup() {
    Serial.begin(9600); // Start serial communication for debugging
    abot.setup();
    abot.servoAttachPins(13, 12); // Replace with actual servo pins

    abot.setDeviationFactor(-0.05);
}

void loop() {
    abot.gradualServoSpeed(100, -100);
    delay(2000); // Drive for 2000 milliseconds (2 seconds)
```

```
abot.gradualServoSpeed(40, -40);
delay(3000);
}
```

After uploading the sketch to Arduino, turn on the robot's power and observe its movement. The robot should gradually reach the target speed. Observe whether the robot moves in the desired direction accurately and whether the speed change is smooth. If the robot's movement path or speed change is different from what you expect, you can adjust the step value.

And if the entered `_deviationFactor` value of `-0.05` is well-calibrated, the robot should not deviate significantly from straight-line driving, even if the speed value is high at 100 or low at 40.

8.2.3 Arduino Robot Following Light

This is an Arduino `.ino` example that utilizes the `lightFollowing` method of the 'EnhancedSelfAbot' class. This example uses a phototransistor to enable the robot to autonomously make decisions and act. Refer to the content and Figure 5.6 in Section 2 of Chapter 5 for setting up the circuit the same way. This implementation is identical to the practice example Ex5.5, but the `.ino` code composition is much more concise because a substantial part of the code's content is contained in the subclass.

A slightly different aspect is that while the previous example Ex5.5 uses the `servoSpeed()` method, the code below uses the `driveStraight()` method, which corrects the robot's straight-line driving. Therefore, the robot's driving action can be more delicate and accurate.

Ex8_3_enhanced_lightfollowing_abot.ino

```
#include "EnhancedSelfAbot.h"

const byte servoLeftPin = 255;
const byte servoRightPin = 255;
EnhancedSelfAbot abot(servoLeftPin, servoRightPin);

void setup() {
  Serial.begin(9600); // Start serial communication for debugging
  abot.setup();
  abot.servoAttachPins(13, 12); // Replace with actual servo pins
}

void loop() {
  unsigned long leftrcTime = abot.rcTime(8);
  unsigned long rightrcTime = abot.rcTime(6);

  abot.lightFollowing(leftrcTime, rightrcTime);
  delay(50);
}
```

When the robot starts to move, turn on a lantern or smartphone light to guide the robot's movements. If the robot begins to move towards the light source, it is functioning correctly.

If there's a need to adjust the robot's subtle actions, you may need to tweak the values in the library code. If you feel your understanding of the code has improved through the lessons, I encourage you to give it a try.

8.2.4 Practice with a Robot Exploring Using Infrared Sensors

For this practice, place infrared LEDs and infrared receivers on the left and right sides of the Arduino robot and complete the circuit connection. The detailed circuit configuration and operating principle are the same as in Section 6.1.

Since infrared light is not visible to the human eye, to indicate whether the infrared sensors are detecting objects or not with light, you can add LED circuits on the left and right sides and include the LED code. Once the hardware connection is complete, refer to the code below and add it accordingly.

```
pinMode(11, OUTPUT);
pinMode(4, OUTPUT);

if (irLeft == 0) {
  digitalWrite(11, HIGH);
} else {
  digitalWrite(11, LOW);
}
if (irRight == 0) {
  digitalWrite(4, HIGH);
} else {
  digitalWrite(4, LOW);
}
```

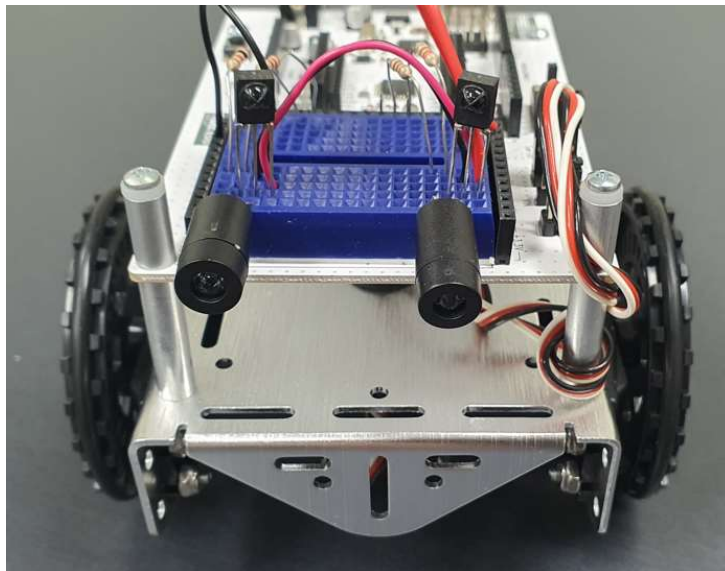


Figure 8.2: The appearance of exploration driving with infrared transceiver sensors

Ex8_4_enhanced_irNavigationdriving_abot.ino

```
#include "EnhancedSelfAbot.h"

const byte servoLeftPin = 255;
const byte servoRightPin = 255;
EnhancedSelfAbot abot(servoLeftPin, servoRightPin);

void setup() {
  Serial.begin(9600); // Start serial communication for debugging
  abot.setup();
  abot.servoAttachPins(13, 12); // Replace with actual servo pins

  abot.setDeviationFactor(-0.05);
}

void loop() {
```



```

int irLeft = abot.irDetect(9, 10, 38000);
int irRight = abot.irDetect(2, 3, 38000);

abot.irNavigationdriving(irLeft, irRight);
delay(50);
}

```

If the sensitivity of the infrared sensors installed on the left and right feels too sensitive or dull (recognizing objects from a distance or failing to recognize them even when close), you should find the appropriate conditions by changing the size of the resistance in the infrared circuit.

Another consideration is the infrared sensors' detection of light, where the ambient light conditions can cause interference. If possible, align all practice conditions in the same environmental conditions, and practice the robot's operation in the same environment. For example, it is not recommended to practice moving between bright areas where sunlight enters and dark areas where light does not enter within a classroom.

8.2.5 Expressing Robot In-Place Rotation Angle

To easily create in-place rotation actions for the Arduino robot, use the `calibrateRotation()` method and `rotateAbot()` method. First, the `calibrateRotation()` method is intended to calculate the correction factors `_setanglefactorCW`, `_setanglefactorCCW`, which can correspond to the robot's rotation angle, using the time (global variable: `duration`) it takes for the robot to make one full 360-degree rotation in place.

When the robot rotates in place, the characteristics of clockwise rotation and counterclockwise rotation can be different. These characteristic differences can be due to factors ranging from slight differences in servo motors to asymmetry in the robot's center of gravity and differences in surface friction between the two wheels.

Therefore, execute the sketch example below for in-place rotation calibration.

Ex8.5 enhanced_rotateinplace_calibration.ino

```

#include "EnhancedSelfAbot.h"

const byte servoLeftPin = 255;
const byte servoRightPin = 255;
EnhancedSelfAbot abot(servoLeftPin, servoRightPin);

int duration = 4400;

void setup() {
  Serial.begin(9600); // Start serial communication for debugging
  abot.setup();
  abot.servoAttachPins(13, 12); // Replace with actual servo pins

  bool isClockwise = false; // Set true for clockwise, false for counterclockwise
  abot.calibrateRotation(duration, isClockwise);
  // Get and display the appropriate _setanglefactor value based on the direction
  if (isClockwise) {
    float setanglefactorCW = abot.getAngleFactorCW();
    Serial.print("_setanglefactorCW clockwise value : ");
    Serial.println(setanglefactorCW);
  } else {
    float setanglefactorCCW = abot.getAngleFactorCCW();
    Serial.print("_setanglefactorCCW counterclockwise value : ");
    Serial.println(setanglefactorCCW);
  }
}
}

```

```
void loop() {  
}
```

Once the calibration for the parameters needed for one full clockwise and counterclockwise rotation through repeated practice is complete, you can easily rotate the robot by applying the desired angle value as a parameter to the rotateAbot() method. Use the rotateAbot() method conveniently when selecting the direction of robot travel or sensor operation.

For clockwise rotation, to find the correction factor (_setanglefactorCW), change the code in sketch Ex8.5 from bool isClockwise = false; to true and upload the sketch. For counterclockwise rotation, keep the value as false, enter the duration value it takes for a 360-degree rotation, and upload the sketch to find the correction factor (_setanglefactorCCW).

Let's explain again in detail how to find the correction factors _setanglefactorCW, _setanglefactorCCW for the robot's in-place rotation. For clockwise rotation, input the bool variable as true, and enter an arbitrary value for the global variable duration to find the optimal time value for a 360-degree rotation. Each time the sketch is executed, the _setanglefactorCW value is displayed on the serial output screen. Record and use the value displayed in the most optimal 360-degree rotation condition.




```
5 EnhancedSelfAbot abot(servoLeftPin, servoRightPin);  
6  
7 int duration = 3950;  
8
```

Serial Monitor Output:
_setanglefactorCW clockwise value : 10.97
_setanglefactorCW clockwise value : 10.97

Figure 8.3: Output of correction factor during the robot's clockwise rotation

For counterclockwise rotation, input the bool variable as false, and enter an arbitrary value for the global variable duration to find the optimal time value for a 360-degree rotation. Each time the sketch is executed, the _setanglefactorCCW value is displayed on the serial output screen. Record and use the value displayed in the most optimal 360-degree rotation condition.



```
5 EnhancedSelfAbot abot(servoLeftPin, servoRightPin);  
6  
7 int duration = 4400;  
8
```

Serial Monitor Output:
_setanglefactorCCW counterclockwise value : 12.22

Figure 8.4: Output of correction factor during the robot's counterclockwise rotation

After completing the calibration with the calibrateRotation() method from the results of Figures 8.3 and 8.4, you can rotate the robot in place by the desired angle using the rotation angle as a parameter. If the calibration practice was conducted correctly, there should be no significant difference between clockwise and counterclockwise rotations, and it should rotate by the desired angle.

The example sketch below repeats the action of rotating the robot 360 degrees clockwise, waiting for 2

seconds, and then rotating it 360 degrees counterclockwise. To rotate the robot clockwise, input a positive number as a parameter to the rotateAbot() method, and for counterclockwise rotation, input a negative number.

Ex8_6_enhanced_abot_rotateinplace.ino

```
#include "EnhancedSelfAbot.h"

const byte servoLeftPin = 255;
const byte servoRightPin = 255;
EnhancedSelfAbot abot(servoLeftPin, servoRightPin);

void setup() {
  Serial.begin(9600); // Start serial communication for debugging
  abot.setup();
  abot.servoAttachPins(13, 12); // Replace with actual servo pins

  abot.setAngleFactorCW(10.97);
  abot.setAngleFactorCCW(12.22);
}

void loop() {
  abot.rotateAbot(360); // Rotate robot by 90 degrees
  delay(2000); // Wait for 2 seconds

  abot.rotateAbot(-360); // Rotate robot by -90 degrees (opposite direction)
  delay(2000); // Wait for 2 seconds
}
```

The calibrateRotation() method functionality is a simplified code form intended for calibration without additional sensor parts.

To start calibration with the below code, you would need to use additional sensors, like a gyroscope or compass, to automate the task. Refer to the below example code. You would need to define a getOrientation() method to process the sensor signal that provides angle data.

```
void EnhancedSelfAbot::calibrateRotation() {
  unsigned long startTime = millis(); // Start time
  float startOrientation = this->getOrientation(); // Get starting orientation
  this->servoSpeed(100, -100); // Start rotation

  float currentOrientation;
  do {
    currentOrientation = this->getOrientation();
    delay(10); // Small delay to prevent too frequent checks
  } while (abs(currentOrientation - startOrientation) < 360.0);
  // Rotate until a full 360-degree turn is completed

  this->servoSpeed(0, 0); // Stop rotation
  unsigned long endTime = millis(); // End time

  // Calculate time taken for a 360 degree rotation
  unsigned long timeTaken = endTime - startTime;
  _setanglefactor = timeTaken / 360.0;
}
```

The introduced code enables the robot to measure its own rotation angle with a sensor and stops upon completing a 360-degree rotation. It then stores the angle conversion coefficient in _setanglefactor, allowing the robot's rotateAbot() method to use the rotation angle parameter for spot rotations. This

means that running the code once is sufficient for a successful calibration, typically performed in the setup function.

8.2.6 Aligning the Robot's Direction by Detecting Frontal Objects with Infrared Sensors

When installing and utilizing two infrared sensors on an Arduino robot, a known method involves using the sensors to detect left and right frontal obstacles independently, interpreting these as '0' or '1' digital signals for detection. Another application method employs infrared sensors to estimate the distance to frontal objects, thereby diversely controlling the robot.

Given that the infrared sensors used in the exercise are not high-end, it's more practical to use them for relative distance estimation rather than absolute distance measurement accuracy. This is because, unlike ultrasonic sensors, infrared light can be significantly affected by the color of objects (e.g., white vs. black), material of objects, and ambient light conditions, causing distance errors. Therefore, it's necessary to minimize these factors and conduct the exercise under consistent environmental conditions.

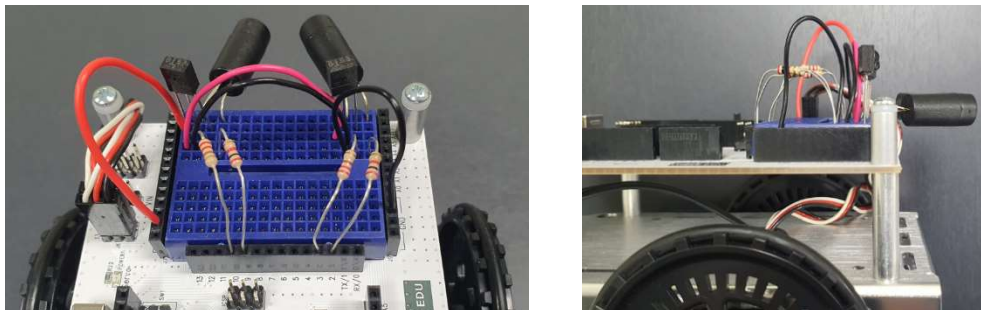
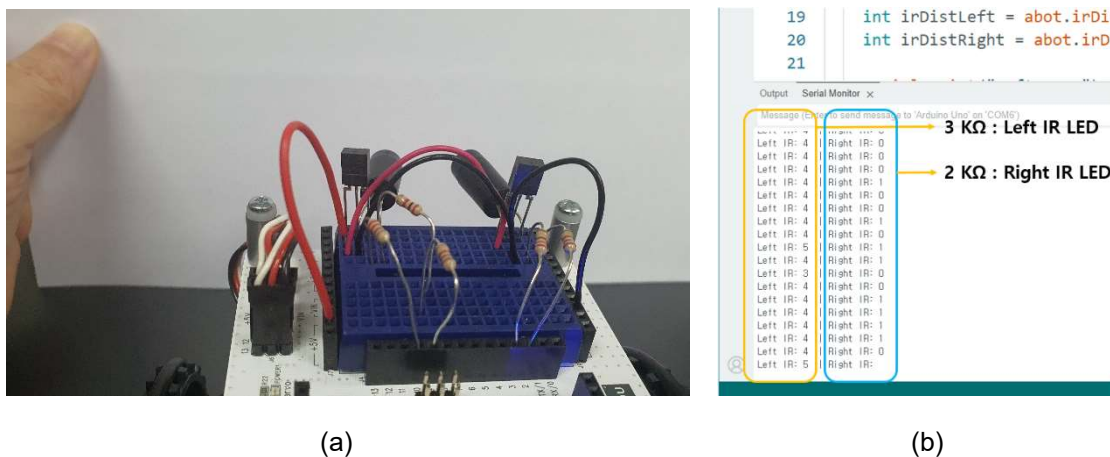


Figure 8.5: Method of Focusing Two Infrared Sensors on a Frontal Object

As explained in Table 7.2, infrared sensors, having a narrower field of view compared to ultrasonic sensors, can more accurately target objects directly in front of them. The orientation of the two infrared sensors can be adjusted inward towards the center to favorably detect frontal objects.

Now, taking into account all limitations, we'll proceed with the exercise by setting up the robot with two infrared sensors as shown in Figure 8.5 and detecting the distance to frontal objects to control the robot's autonomous driving.



(a)

(b)

Figure 8.6: Scenes of an Infrared Sensor Robot Detecting a White A4 Paper Obstacle (a) Detection scene (b) Serial output

The logic of our exercise first involves the robot slowing down and eventually stopping as it approaches an object in its path. Secondly, once stopped, the robot scans from -60 degrees to +60 degrees in place to align itself in the most open direction, opposite to the previous exercise in Chapter 6, Section 6, where the robot aligned itself towards the nearest object. This time, the aim is to align the robot in the direction of the most open space (the direction where objects are furthest away).

To vary the serial output values when detecting a white obstacle with infrared sensors, you can adjust the current size output by the LED. In previous exercises, 2k Ω resistors were used for digital pins 2 and 9. In this exercise, to create a bigger change in output values at closer distances, the resistance size in the left LED circuit was increased by connecting a 2k Ω and a 1k Ω resistor in series.

The result in Figure 8.6(b) shows the Left IR output value is larger than the Right IR, indicating the A4 paper is getting closer. Now, the right LED circuit will also be changed to a 3k Ω resistance condition.

Rather than focusing on the precision of the two infrared sensors' distance measurements, we plan to use the sum of the two output values as a variable for the distance to the frontal object, better handling relative distance changes. The following example sketch implements the logic where the robot slows down and eventually stops as it approaches an object, fulfilling the first logical construct.

Ex8.7_enhanced_irdistance_speedcontrol_infrontofObstacle.ino

```
#include "EnhancedSelfAbot.h"

const byte servoLeftPin = 255;
const byte servoRightPin = 255;
EnhancedSelfAbot abot(servoLeftPin, servoRightPin);

void setup() {
  Serial.begin(9600);
  abot.setup();
  abot.servoAttachPins(13, 12);

  abot.setDeviationFactor(-0.05);
}

void loop() {
  byte irLeftled = 9;
  byte irLeftreceiver = 10;
  byte irRightled = 2;
  byte irRightreceiver = 3;
  int irDistLeft = abot.irDistance(irLeftled, irLeftreceiver);
  int irDistRight = abot.irDistance(irRightled, irRightreceiver);

  int totalIRValue = irDistLeft + irDistRight;

  Serial.print("Left IR: ");
  Serial.print(irDistLeft);
  Serial.print(" || Right IR: ");
  Serial.println(irDistRight);
  delay(50);

  if (totalIRValue > 9) {
    // If the total IR value is more than 9, run the robot forward normally
    abot.gradualServoSpeed(100, -100);
  } else if (totalIRValue >= 1 && totalIRValue <= 9) {
    // If the total IR value is between 5 and 9, reduce the speed gradually
    // Adjust the speed reduction factor as per your requirement
    int speed = map(totalIRValue, 1, 9, 0, 100);
    abot.gradualServoSpeed(speed, -speed);
  } else {
```

```

// If the total IR value is less than 5, stop the robot
abot.servoSpeed(0, 0);
}

delay(20);
}

```

It's time to add the second logic. The essence of the second logic is to align the robot in the direction where there is no obstacle or where the obstacle is furthest away, after scanning from -60 degrees to +60 degrees in place while the robot is stopped.

This purpose of robot action will be added in the loop() function's else condition in the Ex8.7 example content. The below code will first rotate the robot to a -70 degree angle, then rotate it by 10 degrees at a time to measure the distance to frontal objects, recording the angle where the furthest object is detected simultaneously.

Ex8.8_enhanced_irdistanceScan_stoppedPosition.ino

```

#include "EnhancedSelfAbot.h"

const byte servoLeftPin = 255;
const byte servoRightPin = 255;
EnhancedSelfAbot abot(servoLeftPin, servoRightPin);

void setup() {
  Serial.begin(9600);
  abot.setup();
  abot.servoAttachPins(13, 12); // Replace with actual servo pins

  abot.setDeviationFactor(-0.05);
  abot.setAngleFactorCW(10.97);
  abot.setAngleFactorCCW(12.22);
}

void loop() {
  byte irLeftled = 9;
  byte irLeftreceiver = 10;
  byte irRightled = 2;
  byte irRightreceiver = 3;

  int irDistLeft = abot.irDistance(irLeftled, irLeftreceiver);
  int irDistRight = abot.irDistance(irRightled, irRightreceiver);

  int totalIRValue = irDistLeft + irDistRight;

  Serial.print("Left IR: ");
  Serial.print(irDistLeft);
  Serial.print(" || Right IR: ");
  Serial.println(irDistRight);

  if (totalIRValue > 9) {
    // If the total IR value is more than 9, run the robot forward normally
    abot.gradualServoSpeed(100, -100);
  } else if (totalIRValue >= 1 && totalIRValue <= 9) {
    // If the total IR value is between 5 and 9, reduce the speed gradually
    // Adjust the speed reduction factor as per your requirement
    int speed = map(totalIRValue, 1, 9, 0, 100);
    abot.gradualServoSpeed(speed, -speed);
  } else {

```

```

// If the total IR value is less than 5, stop the robot
abot.servoSpeed(0, 0);
delay(1000);

int maxdistance = 0;
int directionOfangle = 0;

int currentAngle = -70;
abot.rotateAbot(currentAngle);
delay(200);

for (int i = 0; i <= 13; i++) {
  if (i != 0) {
    abot.rotateAbot(10);
    currentAngle += 10;
    delay(200);
  }
  int irDistLeft = abot.irDistance(irLeftled, irLeftreceiver);
  int irDistRight = abot.irDistance(irRightled, irRightreceiver);
  int totalIRValue = irDistLeft + irDistRight;

  if (totalIRValue > maxdistance) {
    maxdistance = totalIRValue;
    directionOfangle = currentAngle;
  }
}
abot.rotateAbot(directionOfangle - currentAngle);
}
delay(20);
}

```

After scanning the entire planned angular range, the robot's direction is aligned to the angle where the furthest object was detected, and the for loop is terminated. Upon repeating the loop, the robot can start moving forward normally again since no frontal object is close. The completed sketch example is introduced below in Ex8.8. Upload the sketch and give it a try.

If the code executes correctly, the robot will align itself in the direction where the object is placed furthest away after detecting a frontal object. Combining signals from two infrared sensors helps consistently detect frontal objects, even if one sensor slightly malfunctions or there is a variance between the two sensors. Try creating a robot that can navigate and escape complex mazes.

If there is more than one direction within the search area where the sensor's detection range is exceeded, will it head towards the first or the last direction? What other potential issues might arise? Your imagination can make the robot more intelligent and sophisticated. Implement the methods more intricately while envisioning a robot that monitors frontal objects!

8.2.7 Displaying Radar Screen with Ultrasonic Sensor

Connect the ultrasonic sensor to pin 7, and you can display the radar screen on the monitor connected with the Processing program, similar to the exercise in Section 7.2. Upload the example below to experience the same exercise outcome.

Ex8.9_enhanced_ultrasonic_radar_scan.ino

```

#include "EnhancedSelfAbot.h"

EnhancedSelfAbot abot;
void setup() {
  Serial.begin(9600);
}

```

```

abot.setup();
abot.servoAttachAngle(10);
abot.setUltrasonicSensorPin(7, 7);
}
void loop() {
  abot.ultrasonicRadarData();
  delay(2000);
}

```

The `abot.setUltrasonicSensorPin(7, 7);` code is a setter function to set the `triggerPin` and `echoPin` numbers for the ultrasonic sensor. The parameters used in the current code are for an example using a 3-pin ultrasonic sensor from Parallax, hence the same pin number is used. Additionally, the `abot.ultrasonicRadarData();` code implements rotating the ultrasonic sensor to detect distance and transmit the data via the serial port.

8.2.8 Driving a Robot with Ultrasonic Sensor for Object Detection

Chapter 7 introduced Example Ex7.5, a robot example that drives while detecting objects in front with an ultrasonic sensor. Here, we practice with a slightly more sophisticated robot by utilizing the same code from Chapter 7 but with the `EnhancedSelfAbot` library.

Ex8.10 `enhanced_ultrasonic_navigating_driveRobot.ino`

```

#include "EnhancedSelfAbot.h"

#define SAFE_DISTANCE 20 // safe distance (cm)
#define INVALID_PIN 255
const byte servoLeftPin = INVALID_PIN;
const byte servoRightPin = INVALID_PIN;

EnhancedSelfAbot abot(200, 20, 40);

void setup() {
  Serial.begin(9600);
  abot.setup();
  abot.servoAttachPins(13, 12); // Replace with actual servo pins
  abot.servoAttachAngle(10);

  abot.setUltrasonicSensorPin(7, 7);

  abot.setDeviationFactor(-0.05);
  abot.setAngleFactorCW(10.97);
  abot.setAngleFactorCCW(12.22);
}

void loop() {
  abot.gradualServoSpeed(80, -80); // forward robot driving

  scanInDirection(-30, 30); // scan from -30 to +30
  delay(100);
  scanInDirection(30, -30); // scan from +30 to -30

  delay(100);
}

void scanInDirection(int startOffset, int endOffset) {
  int step = startOffset < endOffset ? 15 : -15; // Determines the scan direction

  for(int offset = startOffset; step > 0 ? offset <= endOffset : offset >= endOffset; offset += step) {

```



```

abot.servoAngle(90 + offset); // Adjusts the servo motor to the specified angle
delay(30); // Time for the servo motor to move and stabilize
int distance = abot.calculateDistance(); // Measures distance at the current angle

if(distance <= SAFE_DISTANCE) {
    abot.gradualServoSpeed(20, -20); // Stops the robot if an obstacle is detected
    abot.detectAndProcessSafeArea(); // Performs detailed search and avoidance
    abot.gradualServoSpeed(80, -80); // Advances again after avoidance action
    break;
}
}
}

```

This practice example demonstrates a robot that slows down for scanning when a front object appears and resumes normal speed after finding a direction with no front objects.

To successfully perform Example Ex8.10, you must first complete all the examples from the beginning of Chapter 8. These examples include finding correction factors to offset mechanical movement deviations in the Arduino robot, allowing it to drive straight regardless of speed, implement gradual robot speed, and perform proportional clockwise or counterclockwise rotation in place.

For Example Ex8.10, the `setup()` function must apply unique correction factors for your robot.

```

abot.setDeviationFactor(-0.05);
abot.setAngleFactorCW(10.97);
abot.setAngleFactorCCW(12.22);

```

The values applied in the code must be unique to the experimenter. Also, the code represents using a 3-pin ultrasonic sensor with pin 7.

```

abot.setUltrasonicSensorPin(7, 7);

```

If using a 4-pin ultrasonic sensor, the first parameter should be the trigger pin and the second parameter the echo pin.

The constructor can specify several variables for exploring with the ultrasonic sensor. The following code details these unique variables for this practice.

```

EnhancedSelfAbot abot(200, 20, 40);

```

This object declaration with three parameters uses the constructor. The first parameter sets the maximum distance the ultrasonic sensor can measure to 200 cm, a provisional value that experimenters can test and adjust to more accurately reflect reality.

The second parameter is the distance under which the ultrasonic sensor deems it unsafe when less than 20 cm. Experimenters can adjust this value based on robot speed, sensor scanning speed, and the complexity of surrounding obstacles.

✧ *Note that the second parameter value has the same meaning as the #define SAFE_DISTANCE 20 code used in the .ino code for safety distance conditions.*

The third parameter represents conditions where the ultrasonic sensor detects an angular range outside the safety distance to be greater than 40 degrees, implying the robot has enough room to pass in the selected direction. This value can also be adjusted according to the user's discretion.

Explore various practices based on the above code design concepts.

The `scanInDirection()` function used in the .ino example is examined.

```

scanInDirection(-30, 30);
delay(100); // Wait before changing scan direction

```

```
scanInDirection(30, -30);  
delay(100);
```

In the code, the scan range is from -30 to 30 or from 30 to -30, implementing the code within the function using a step of 15 and a ternary operator to determine whether to add or subtract 15.

```
int step = startOffset < endOffset ? 15 : -15; // Determines scan direction  
for(int offset = startOffset; step > 0 ? offset <= endOffset : offset >= endOffset; offset += step) {  
    abot.servoAngle(90 + offset); // Adjusts the servo motor to the specified angle  
    delay(30); // Time for the servo motor to move and stabilize  
}
```

This operation contrasts with the following example code, offering more flexibility.

```
int angles[] = {-30, -15, 0, 15, 30}; // Defines angle offsets in an array  
for(int i = 0; i < 5; i++) { // Iterates over each array element  
    int angleOffset = angles[i];  
    abot.servoAngle(90 + angleOffset); // Adjusts the servo motor to the specified angle  
    delay(100); // Time for the servo motor to move and stabilize  
}
```

The ultrasonic sensor measures and records the distance to frontal objects.

```
abot.calculateDistance()
```

Unlike the conditions in Chapter 7's example, the if condition does not immediately stop when an obstacle is detected within the front safety distance but slows down to 20 speed. Then, it scans the surroundings with the sensor and resumes normal speed of 80 once a safe driving direction is determined.

```
if(distance <= SAFE_DISTANCE) {  
    abot.gradualServoSpeed(20, -20); // Robot stops if an obstacle is detected  
    abot.detectAndProcessSafeArea(); // Performs detailed search and avoidance action  
    abot.gradualServoSpeed(80, -80); // Advances again after avoidance action  
    break; // Exits the loop if any condition is met as further detection is unnecessary  
}
```

The detailed explanation of methods used in the library is omitted. If you need a deeper understanding, feel free to ask ChatGPT about the code's operating principles. Engaging with artificial intelligence in conversation might reveal new possibilities for using the code.

We wish you continued profound understanding of the code.

Appendix: 'EnhancedSelfAbot.zip' library header and source file
(please download EnhancedSelfAbot.cpp file for review).

```
#ifndef ENHANCEDSELFABOT_H
#define ENHANCEDSELFABOT_H

#include "SelfAbot.h" // Include the base class header
#include "Arduino.h"
#include <Servo.h>

struct Detection {
  int angle;
  unsigned int distance;
};

class EnhancedSelfAbot : public SelfAbot {
public:
  EnhancedSelfAbot();
  EnhancedSelfAbot(byte servoLeftPin, byte servoRightPin);
  EnhancedSelfAbot(unsigned int maxDistance, unsigned int safeDistance, int angularRange);

  static const int MAX_DETECTIONS = 40;

  void resetDeviationFactor();
  float getDeviationFactor() const;
  void setDeviationFactor(float deviationFactor);
  void calibratedeviationFactor(float deviatedistance);

  void driveStraight(int leftSpeed, int rightSpeed);
  void gradualServoSpeed(int targetLeftSpeed, int targetRightSpeed);
  void lightFollowing(unsigned long leftTime, unsigned long rightTime);
  void irNavigationdriving(int irLeft, int irRight);

  void rotateAbot(int angle);
  void calibrateRotation(int duration, bool clockwise);
  // Setters
  void setAngleFactorCW(float angleFactorCW);
  void setAngleFactorCCW(float angleFactorCCW);

  // Getters
  float getAngleFactorCW() const;
  float getAngleFactorCCW() const;
  void setUltrasonicSensorPin(byte triggerPin, byte echoPin);
  int calculateDistance();
  void ultrasonicRadarData();
  void detectAndProcessSafeArea();
  bool scanAndProcessAngle(int angle, bool isDescending);
  int findSafeAreaCenterAngle(bool isDescending);
  void setTravelDirection(int angle);
private:
  float _deviationFactor;
  float _setanglefactorCW, _setanglefactorCCW;
  int _currentLeftSpeed; // Current speed for left servo as a class member
  int _currentRightSpeed; // Current speed for right servo as a class member
  Detection _detections[MAX_DETECTIONS];
  int _detectionCount;
  int _notDetectedAngularRange;
  byte _triggerPin, _echoPin;
  unsigned int _maxDistance;
  unsigned int _safeDistance;
};

#endif // ENHANCEDSELFABOT_H
```

Chapter 9 Advanced Movement and Control of the Robot

9.1 Implementing Advanced Functions of Wireless Robots

9.2 Managing Unanticipated Situations During Robot Operation



Adding wireless capabilities to an Arduino robot significantly enhances its functionality and applications. Let's examine why wireless capabilities are beneficial and the advantages they offer.

(1) Increased Mobility and Operational Range:

Unlimited Movement: Wireless communication frees the robot from the constraints of physical wires, enabling unrestricted movement. This is particularly important for tasks related to exploration, navigation, or interaction in various environments.

Extended Operating Range: Robots can operate at distances beyond the reach of wires, expanding the scope of projects and applications.

(2) Remote Monitoring and Control:

Real-time Data: Wireless communication allows for the transmission of real-time data. This means sensor readings, robot status, or environmental conditions can be monitored from remote locations.

Remote Control: Commands can be sent to the robot via remote workstations, smartphones, or the internet, allowing for remote control and intervention when needed.

(3) Scalability and Network Integration:

Easy Expansion: Wireless systems are easier to expand than wired systems. Adding more robots or sensors to the network doesn't require complex wiring. They just need to be configured to communicate wirelessly with existing systems.

Integration into Large Systems: Wireless robots can be integrated into large networks, including the Internet of Things (IoT), interacting with other smart devices, cloud-based services, or data analysis

platforms. Arduino supports various wireless communication protocols such as Wi-Fi, Bluetooth, Zigbee (XBee), LoRa, and NFC, each offering unique features suitable for different applications.

(4) Enhanced Capabilities through Collaboration:

Swarm Robotics: Wireless communication allows multiple robots to work together in harmonious groups or swarms. Such collaboration can lead to complex group behaviors and efficiencies in tasks like area mapping, search and rescue, or collective object transportation.

Data Sharing: Robots can share sensor data or computational results with each other, making decisions based on more information and enabling adaptive behaviors. Wireless systems are essential for real-time data collection and control in applications such as environmental monitoring, smart homes, or industrial automation.

(5) Flexibility and Adaptability:

Rapid Deployment: Wireless robots can be quickly deployed in various environments without the need for extensive infrastructure.

Adaptation to the Environment: Wireless robots can adjust their operations based on environmental or central control system feedback, making them suitable for dynamic or unpredictable conditions.

(6) Enhanced User Interaction:

User-friendly Interfaces: Wireless communication allows for the use of user-friendly interfaces on devices like smartphones, tablets, and computers to control the robot.

Educational and Entertainment Value: Wireless robots have great potential in education and entertainment, enabling interactive and participatory activities such as robot competitions, interactive art installations, or educational demonstrations.

Wireless communication opens up various possibilities but requires consideration of factors such as communication range, battery life, wireless protocol security, and potential interference from other wireless devices.

9.1 Implementing Advanced Features in Wireless Robots

The 'neo-Arduino robot car' product used in this book is designed with hardware options for wireless communication. To adopt wireless functionality, components for cellular, Bluetooth, XBee/Zigbee, Wi-Fi communications can be easily mounted on the Arduino robot's slot, and hardware toggle switches can be used to easily create wireless communication conditions. The types of wireless antennas compatible with the 'neo-Arduino robot car' product, detailed code writing methods, and the use of toggle switches can be referred to in the provided materials.



Figure 9.1: Types of XBee pin antennas to attach to the Arduino robot (abot)

Especially, XBee communication is well-known for short-range 1:N communications. Although the examples use XBee Series 1 antennas, XBee Series 3 has been released, offering improved communication performance. Future discussions may cover short-range wireless communication methods.

Using the toggle switch on the Arduino robot 'Fribee EDU' board allows for the selection between USB communication and wireless communication for serial communication via Tx/Rx pins (0, 1), though they cannot be used simultaneously.

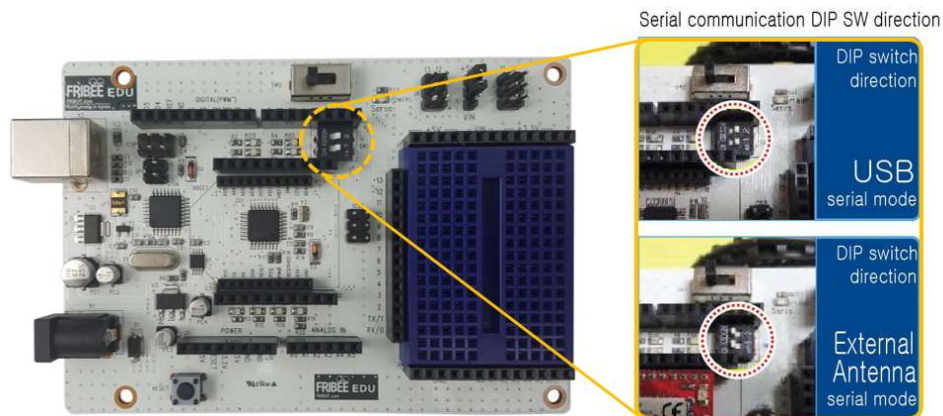


Figure 9.2: Toggle (DIP) switches for H/W serial communication using the Fribee board

The easiest method is to upload the code via USB communication, check the serial data values on the screen, and then switch the toggle to wireless communication, ensuring the same serial data values are transmitted wirelessly (*initial settings and conditions for the antenna must be prepared in advance*).

The choice between communicating via USB or wirelessly depends on the position of the toggle switch, as shown in figure 9.2.

There are two methods for serial communication with wireless on an Arduino (or Arduino-compatible Fribee) board: using Fribee's hardware serial and software serial. Let's examine the differences between the two.

Hardware Serial (Using Tx/Rx pins 0/1):

Advantages: Hardware serial communication is generally more stable and can handle higher transmission speeds than software serial, as the hardware is specifically designed for serial communication. With the hardware handling serial communication, CPU overhead is reduced, benefiting other processing tasks. Hardware serial also manages interrupts more effectively, allowing for smoother multitasking.

Disadvantages: Using hardware serial (pins 0 and 1 of Arduino) means they cannot be used for general I/O, and especially during code upload or serial monitoring, it's necessary to switch to USB communication. Additionally, there's a potential waste of cycles as the CPU waits for the transmission to complete if not managed properly.

Software Serial (Using pins other than 0/1 for Tx/Rx):

Advantages: Provides flexibility to use all digital pins for serial communication, leaving hardware serial pins free for other tasks or USB communication. This is useful for creating multiple serial connections on boards with only one hardware serial port.

Disadvantages: Software serial may not handle high transmission speeds as reliably as hardware serial, especially on less powerful microcontrollers. It introduces more CPU overhead as the CPU emulates serial communication, which can disrupt important timing or processing tasks. Additionally, without dedicated hardware buffering, it's more prone to buffer overrun issues.

Considerations for Wireless Communication:

(1) **Interference and Noise:** Wireless communication is susceptible to interference and noise, which can be more problematic in software serial due to lower tolerance for timing disruptions.

(2) **Power Consumption:** Depending on the method of wireless communication, power consumption can be a critical factor, affecting the choice between hardware and software serial. Software serial might require more CPU activity during communication, increasing power consumption.

(3) **Complexity:** Implementing wireless communication adds complexity, especially in ensuring reliable data transmission over a potentially unreliable medium. This can influence the choice between hardware and software serial based on specific application requirements such as error correction or

buffer management.

Additional considerations for wireless communication include setting up communication capabilities on the counterpart device (PC, smartphone, or another Arduino robot), preparing antennas with initial and condition settings, and possibly developing apps for smartphone communication.

In situations where many students begin wireless communication experiments in the same space, managing unique wireless antenna IDs might be necessary.

The following is an example code for **'hardware serial' communication on an Arduino**, allowing for data transmission via keyboard input to the Arduino or data transmission from the Arduino to the computer screen while connected via USB cable.

```
void setup() {
  Serial.begin(9600);
}

void loop() {
  if (Serial.available() > 0) {
    // Read the incoming byte:
    char incomingByte = Serial.read();

    // Say what you got:
    Serial.print("I received: ");
    Serial.println(incomingByte, DEC);
  }
}
```

The example code for **'Software Serial' communication for the Arduino** robot is as follows.

```
#include <SoftwareSerial.h>
// RX and TX pins for the software serial
int rxPin = 10;
int txPin = 11;
// Initialize the software serial port
SoftwareSerial mySerial(rxPin, txPin);

void setup() {
  // Open the software serial port at 9600 bps
  mySerial.begin(9600);
  // Start the hardware serial port for debugging
  Serial.begin(9600);
}

void loop() {
  // Check if data is available to read
  if (mySerial.available()) {
    char inChar = (char)mySerial.read();
    Serial.print("Received: ");
    Serial.println(inChar);
  }
  // Send a message every second
  static unsigned long lastSendTime = millis();
  if (millis() - lastSendTime > 1000) {
    mySerial.println("Hello from Arduino!");
    lastSendTime = millis();
  }
}
```

9.1.1 Displaying Objects Detected with Infrared Sensor on Radar Output Screen

This practice screen reproduces the "8.2.7 Ultrasonic Sensor Radar Screen Output" practice wirelessly. We used the "Ex8.9_enhanced_ultrasonic_radar_scan.ino" code without modification and switched the toggle (DIP) switch in the opposite direction to utilize the hardware serial port.

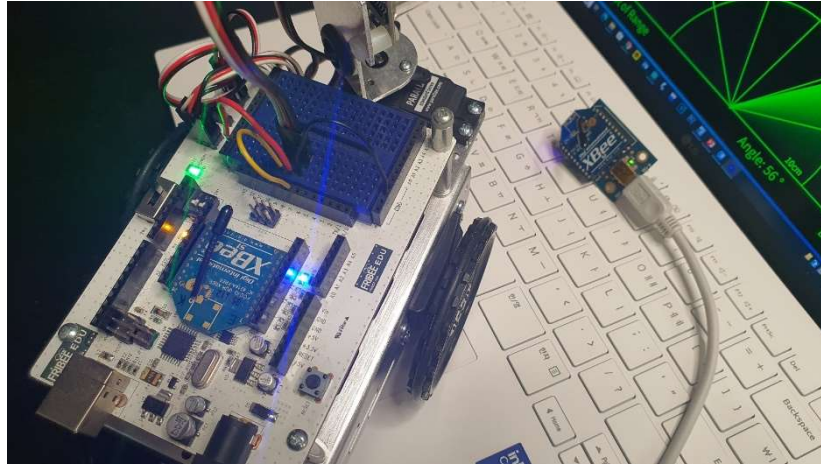


Figure 9.3: Outputting Ultrasonic Sensor Data to Radar Screen Wirelessly

In most microcontroller environments, including Arduino, we write codes connected through wired USB serial, check data on the output screen, and often think we need to add or change codes to use software serial for wireless environment conversion. Then, there's a need to test the changed code again.

Using hardware serial means the output screen check results in the USB serial environment become the final results.

To configure Arduino's software serial, you connect Tx/Rx pins for wireless data transmission to other numbers excluding the existing hardware serial numbers 0 and 1 with jumper wires to configure software serial. To configure Arduino's hardware serial, even if not equipped with a toggle (DIP) switch like the Fribee board, when you choose the pin numbers for Tx/Rx for wireless communication with jumper wires to use 0 and 1, it means you are using hardware serial.

Using hardware serial, as introduced in this practice, means you can easily set up a wireless communication environment with little to no modification of existing codes.

The antenna used in Figure 9.3 for wireless communication is the original XBee Series 1 antenna. Series 1 antennas have been discontinued, and Series 3 products have been released, offering superior performance for IOT construction.

We won't delve deeper here, but note that setting up wireless communication involves more than just attaching an antenna to the robot. Especially, tasks like setting up a mesh network require configuring antennas for end nodes, routers, and coordinators, among others.

More examples of controlling robots wirelessly will be introduced later.

9.2 Managing Unexpected Situations During Robot Operation

Managing unexpected situations during Arduino robot operation is especially important for beginners. Robots often interact with dynamic environments and can behave unpredictably. Here's a summary of managing unexpected situations for beginners in eight points:

(1) Understanding the Robot and Its Environment

Understand the Robot: Know how the robot should behave under normal conditions. Be aware of the robot's capabilities and limitations.

Be Aware of the Environment: Know the environment where the robot operates. Various floors, obstacles, lighting conditions, and even weather can affect performance.

(2) Programming for Safety and Error Handling

Safe Defaults: Program the robot to stop or enter a safe state when encountering unexpected situations.
Error Handling: Implement error handling in your code. For example, ensure the robot reacts safely if a sensor fails to read.

Timeouts: Use timeouts in your code so that if expected events don't occur within a specified time, the robot can perform default actions.

(3) Validating Sensor Data

Check Sensor Readings: Continuously monitor sensor readings to detect anomalies. For example, treat sudden '0' or maximum values from a distance sensor as potential errors.

Duplicate Sensors: Use redundant sensors for critical measurements. This way, if one sensor fails, you can rely on another.

(4) Power Management

Monitor Battery Levels: Ensure the robot can detect low battery and react appropriately, such as returning to a home base or shutting down non-essential functions.

Voltage Regulation: Ensure the power supply is stable. Voltage fluctuations can cause unpredictable behavior.

(5) Mechanical Safety Measures

Physical Limits: Implement physical limits, like bumpers, to prevent damage during collisions.

Emergency Stop: Have a way to immediately stop the robot manually to halt its operation when necessary.

(6) Testing and Debugging

Thorough Testing: Test the robot in a controlled environment first, then gradually increase the complexity of tasks.

Debugging: Learn how to use tools that help debug Arduino code. The serial monitor is a great tool for obtaining real-time feedback from the robot.

(7) Handling External Interference

Manage Interference: Be aware of electromagnetic interference (EMI) or interference from other wireless devices. Shielding or changing frequencies can help resolve issues.

(8) Learning and Iterative Development

Continuous Learning: Use unexpected events as learning opportunities. Use these experiences to improve the robot's design and code.

Iterative Development: Develop the robot iteratively. Start with simple tasks and add complexity as you gain experience and confidence.

Suggestions for Exception Handling in Arduino Robots

Exception handling is a programming concept used to manage errors and abnormal conditions that occur during program execution. However, it's important to note that in the context of Arduino and other embedded systems, traditional exception handling seen in higher-level programming languages (such as try-catch blocks in C++ or Java) is often not used or not available. This is mainly due to the limited memory and processing resources of microcontrollers.

The Arduino platform typically incorporates a simplified version of C++ programming. While standard C++ supports exception handling, Arduino programming generally does not use this feature. Instead, Arduino programmers employ alternative methods to deal with errors and unexpected situations. Below is a beginner-friendly explanation of how to handle exceptions and errors in Arduino-based robots.

(1) Checking for Error Conditions

Instead of relying on exceptions, you can explicitly check for potential error conditions. For example, if a sensor fails or provides incorrect readings, you can write code to verify that the sensor values are within a reasonable range.

```

int sensorValue = analogRead(sensorPin);
if (sensorValue < MIN_VALUE || sensorValue > MAX_VALUE) {
    // Handle error: maybe set an error flag, try to read sensor again, etc.
}

```

(2) Using Return Values and Error Codes

Functions can return special values or error codes to indicate that a problem has occurred. The main program must check these return values and decide what action to take.

```

int result = performTask();
if (result == ERROR_CODE) {
    // Handle the error
}

```

(3) Failsafe States

Design your robot to enter a 'safe' state if a problem occurs. For example, if there's an error with the motor driver, the robot should stop all movements to prevent damage or unsafe behavior.

(4) Timeout Mechanisms

Implement timeout mechanisms in your code, especially when dealing with communication protocols or waiting for events. If the expected event doesn't occur within a specific time frame, assume an error has occurred and handle accordingly.

```

unsigned long startTime = millis();
while (!eventOccurred()) {
    if (millis() - startTime > TIMEOUT_DURATION) {
        // Handle timeout
        break;
    }
}

```

(5) Watchdog Timer

Some microcontrollers, including Arduino boards, have a feature called a watchdog timer. It's a hardware timer that resets the system if the program hangs or enters an infinite loop.

(6) Debugging Output

Use serial communication to send debugging information from the Arduino to a computer. This helps you understand what's happening in the program and identify where and when problems occur.

```

Serial.println("Starting task...");
// Perform some task
Serial.println("Task completed.");

```

In summary, traditional exception handling using Try-Catch blocks is generally not used in Arduino programming, but there are several other strategies you can employ to handle errors and unexpected situations. These include checking for error conditions, using return values and error codes, designing failsafe states, implementing timeouts, using watchdog timers, and relying on debugging output.

Appendix

Processing source code for ultrasonic radar imaging.

```
/*
Radar Screen Visualisation for HC-SR04
Maps out an area of what the HC-SR04 sees from a top down view.
Takes and displays 2 readings, one left to right and one right to left.
Displays an average of the 2 readings
Displays motion alert if there is a large difference between the 2 values.
*/
import processing.serial.*; // import serial library
Serial arduinoport; // declare a serial port
float x, y; // variable to store x and y co-ordinates for vertices
int radius = 350; // set the radius of objects
int w = 300; // set an arbitrary width value
int degree = 0; // servo position in degrees
int value = 0; // value from sensor
int motion = 0; // value to store which way the servo is panning
int[] newValue = new int[181]; // create an array to store each new sensor value for each servo position
int[] oldValue = new int[181]; // create an array to store the previous values.
PFont myFont; // setup fonts in Processing
int radarDist = 0; // set value to configure Radar distance labels
int firstRun = 0; // value to ignore triggering motion on the first 2 servo sweeps
/* create background and serial buffer */
void setup(){
// setup the background size, colour and font.
size(1204, 650);
background(0); // 0 = black
myFont = createFont("verdana", 12);
textFont(myFont);
// setup the serial port and buffer
arduinoport = new Serial(this, "COM4", 9600);
}

/* draw the screen */
void draw(){
fill(0); // set the following shapes to be black
noStroke(); // set the following shapes to have no outline
ellipse(radius, radius, 750, 750); // draw a circle with a width/ height = 750 with its center position (x and y) set by the radius
rectMode(CENTER); // set the following rectangle to be drawn around its center
rect(350,402,800,100); // draw rectangle (x, y, width, height)
if (degree >= 179) { // if at the far right then set motion = 1/ true we're about to go right to left
motion = 1; // this changes the animation to run right to left
}
if (degree <= 1) { // if servo at 0 degrees then we're about to go left to right
motion = 0; // this sets the animation to run left to right
}
/* setup the radar sweep */
/*
We use trigonometry to create points around a circle.
So the radius plus the cosine of the servo position converted to radians
Since radians 0 start at 90 degrees we add 180 to make it start from the left
Adding +1 (i) each time through the loops to move 1 degree matching the one degree of servo movement
cos is for the x left to right value and sin calculates the y value
since its a circle we plot our lines and vertices around the start point for everything will always be the center.
*/
strokeWeight(7); // set the thickness of the lines
if (motion == 0) { // if going left to right
for (int i = 0; i = 0; i--) { // draw 20 lines with fading colour
stroke(0,200-(10*i), 0); // using standard RGB values, each between 0 and 255
line(radius, radius, radius + cos(radians(degree+(180+i)))*w, radius + sin(radians(degree+(180+i)))*w);
}
}
}
```

```

/* Setup the shapes made from the sensor values */
noStroke(); // no outline
/* first sweep */
fill(0,50,0); // set the fill colour of the shape (Red, Green, Blue)
beginShape(); // start drawing shape
for (int i = 0; i < 180; i++) { // for each degree in the array
x = radius + cos(radians((180+i)))*(oldValue[i]); // create x coordinate
y = radius + sin(radians((180+i)))*(oldValue[i]); // create y coordinate
vertex(x, y); // plot vertices
}
endShape(); // end shape
/* second sweep */
fill(0,110,0);
beginShape();
for (int i = 0; i < 180; i++) {
x = radius + cos(radians((180+i)))*(newValue[i]);
y = radius + sin(radians((180+i)))*(newValue[i]);
vertex(x, y);
}
endShape();
/* average */
fill(0,170,0);
beginShape();
for (int i = 0; i = 360) {
stroke(150,0,0);
strokeWeight(1);
noFill();
for (int i = 0; i 35 || newValue[i] - oldValue[i] > 35) {
x = radius + cos(radians((180+i)))*(newValue[i]);
y = radius + sin(radians((180+i)))*(newValue[i]);
ellipse(x, y, 10, 10);
}
}
}
/* set the radar distance rings and out put their values, 50, 100, 150 etc.. */
for (int i = 0; i <=6; i++){
noFill();
strokeWeight(1);
stroke(0, 255-(30*i), 0);
ellipse(radius, radius, (100*i), (100*i));
fill(0, 100, 0);
noStroke();
text(Integer.toString(radarDist+50), 380, (305-radarDist), 50, 50);
radarDist+=50;
}
radarDist = 0;
/* draw the grid lines on the radar every 30 degrees and write their values 180, 210, 240 etc..
*/
for (int i = 0; i = 300) {
text(Integer.toString(180+(30*i)), (radius+10) + cos(radians(180+(30*i)))*(w+10), (radius+10)
+ sin(radians(180+(30*i)))*(w+10), 25,50);
} else {
text(Integer.toString(180+(30*i)), radius + cos(radians(180+(30*i)))*w, radius +
sin(radians(180+(30*i)))*w, 60,40);
}
}
}

```


*** Copyright and Ownership Notice and Permission for Use ***

© 2024, Chung Wookjin of Fribot co. Ltd. All rights reserved.

This book and its contents are protected by copyright law. This book is provided online for free for educational purposes, and Fribot grants permission for the non-commercial use of the contents of this book by educational institutions, teachers, students, and individuals for self-development.

However, the contents of this book, including the images, code, and other materials contained within, may not be reproduced, distributed, transmitted, modified, displayed, published without explicit written permission. The Dall-E images included in this book are used in accordance with OpenAI's policy, and the copyright of these images belongs to OpenAI. These images are used only for the purpose of supplementing the contents of this book.

The code examples provided in this book may be used for educational purposes, and the author or publisher is not responsible for any outcomes resulting from their use.

The content of this book is based on the author's knowledge and experience, and efforts have been made to provide accurate information. However, the author and publisher are not liable for any direct, indirect, incidental, special, or consequential damages arising from the use of the contents of this book.

This book contains information available at the time of publication, and the author or publisher is not responsible for any changes or updates to the information contained in this book.

All users utilizing this book for educational purposes must comply with these notice requirements. For commercial use, translation, modification, or other utilization of the contents of this book, explicit written permission from the author or publisher is required.