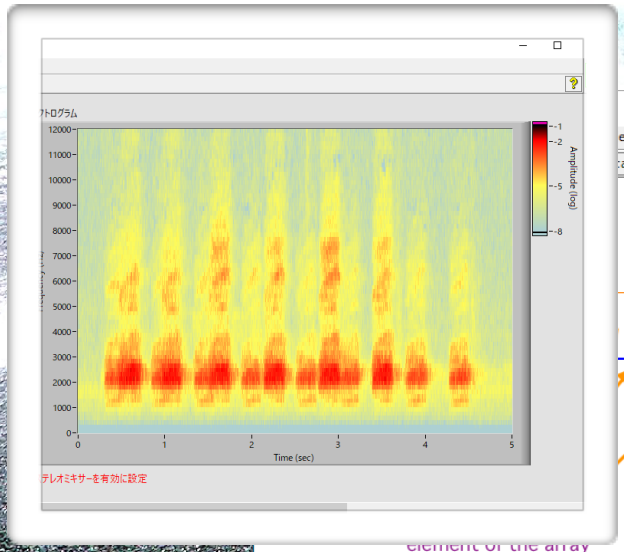
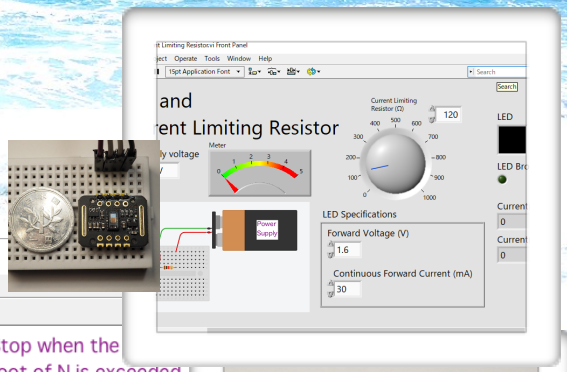
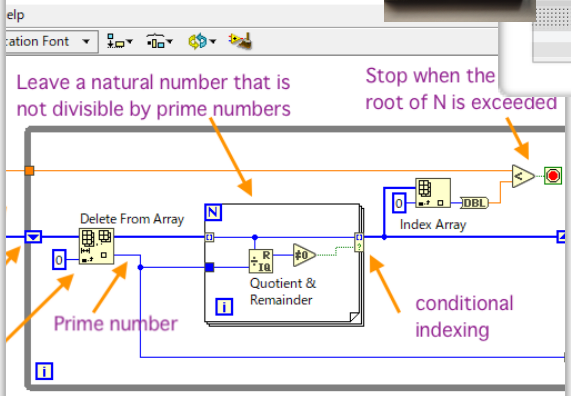
Free for hobbies

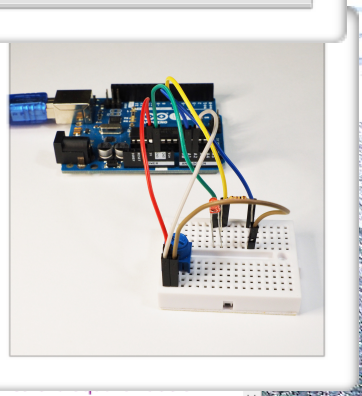
Enjoy Programming

with LabVIEW Community Edition









Volunteer members of Japan LabVIEW Users Group

Foreword

Jeff Kodosky, the father of LabVIEW,
gave us his thoughts on LabVIEW Community Edition and the following message for our readers.

We created the LabVIEW Community editions so engineers could use the software for free — to pursue their hobbies and personal projects, experiment with programming ideas and create and share IP with their peers. I am thrilled to see the Japan LabVIEW User Group create this e-book that contains everything a budding hobbyist would need to start programming with LabVIEW. The quality of this work shows we have a very passionate community of LabVIEW developers who want to share their love of LabVIEW with everyone. I personally want to thank the Japan LabVIEW User Group for their enthusiasm, dedication, and hard work. I look forward to seeing the many fantastic projects everyone will create with the LabVIEW Community edition after reading and learning with this free e-book.

JEFF KODOSKY
Inventor of LabVIEW
Cofounder and Business and Technology Fellow, NI

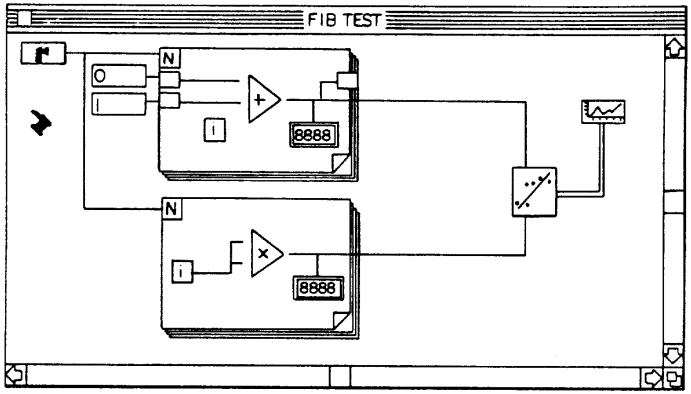
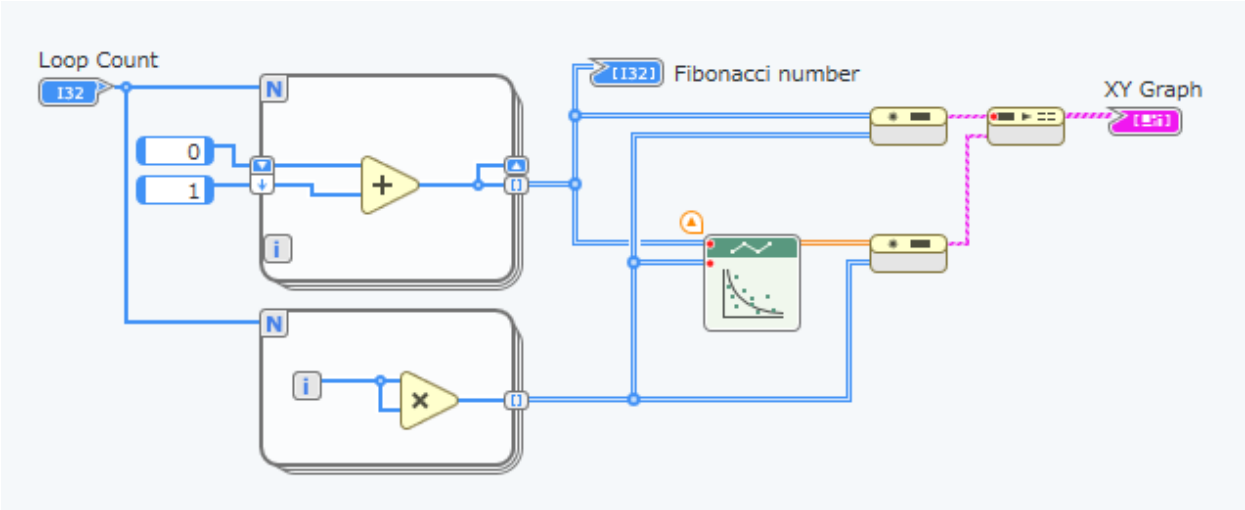


FIG. —57



LabVIEW NXG Community Edition June 24, 2020

Introduction

What comes to your mind when you hear the word "programming"? Your computer and smartphone are in fact, working thanks to programming. You can find programming everywhere. There is rarely any home appliance (TV, fridge) that does not use programming. It is hard to come to the realization, but programming has become a part of everyday life.

From 2020, programming has been included in the school curriculum in Japan (Figure 0-1). The word "study" might put you on guard, but unlike other subjects, studying involves using a computer. Now that sounds a bit better, doesn't it? When you are forced to study, learning stops being fun because it is something you have to do. But there are many perks to being able to program.

- **Automating various tasks**

The tasks that you used to do manually can be automated by programming. For example, if your homework is to measure and record the temperature every hour, you can use programming to automatically measure and record repetitively at specified intervals.

Figure 0-1 Studying???

- **More convenient life**

Old TVs could only show TV programs. However, with the progress of technology, especially in programming, TVs can now connect to the Internet. Various new features have been added, including using the Internet on TV. Another example is the cleaning robot (Figure 0-2). It remembers the cleaning course and gets every corner of the house squeaky clean thanks to programming.

- **Job opportunities**

There are many companies out there that want people with programming skills. Becoming a programmer (someone who can program) will open more career doors for you in the future.

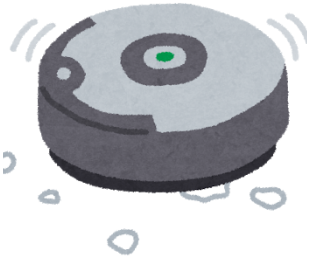


Figure 0-2 Cleaning robot

- **Make your own games**

All games, whether RPGs or fighting games, are made by programming. It is common to buy and play the games you want, but once you master programming, you can create and play your own titles.

Now that we have covered the perks, let us look at different kinds programming out there. It is said that there are several thousands types of programming in the world. They each have their own set of programming rules. Just like English and Japanese, they can be used to program the same content with different rules (Figure 0-3).

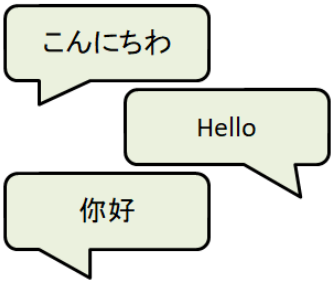


Figure 0-3 Greetings

Because of this, programming rules are collectively called "programming languages." There are vast number of programming languages, ranging from ones that have been around for decades like machine language and C, to the popular Python and LabVIEW. It is hard to say which is better than which, but it is often the case that some programming languages become obsolete while some become popular because of the trends in the world. It is quite important which language you choose to learn. The outcome of your studying is way more valuable if you choose to learn a popular language rather than an obsolete one, because there is more demand out there for programmers who can program in the popular languages. The first milestone of studying programming is choosing the programming language.

In this book, you will learn programming using LabVIEW. LabVIEW is not just a programming language for learning purpose but also commonly used in the real world. LabVIEW is used to develop new products in companies and perform state-of-the-art researches in universities (Figure 0-4, Figure 0-5). In this book, the small details of programming (such as bit arithmetic) are not covered. However, you will master how to "measure" and "control" with LabVIEW. We hope that you will simply enjoy programming and the science along this journey with us.

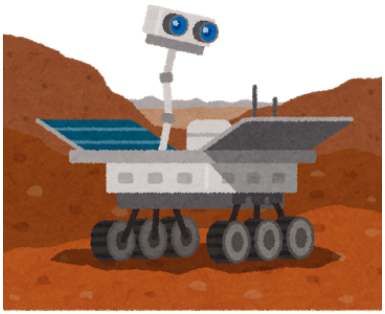


Figure 0-4 Mars rover

Figure 0-5 IoT (Internet of Things)

Contents

Part 2 Getting Started with LabVIEW Programming

Learn the Basics of LabVIEW Programming

Chapter 3 First Exposure to Graphical Programming

LabVIEW-specific programming methods such as icons, wires, and data flows

Chapter 4 Making Your Own Application

How to make a practical application that can record and playback

Part 3 Electronic Projects with LabVIEW and Arduino

Please prepare Arduino UNO, breadboard, tact switch, fan, LED, fixed resistance (100Ω), variable resistance (10kΩ), 6 wires

Chapter 5 LabVIEW and Arduino

Start physical computing with Arduino

Chapter 6 Investigating LED Properties

Examine the LED properties

Chapter 7 Using the Latest Semiconductor Sensors

Sensors used in smartphones and automobiles are sold as modules that can be easily connected to Arduino, so we will introduce an example of how to use them.

List of Example VIs

Afterword

Part 1
Getting Started with LabVIEW

Chapter 1
Studying Programming with LabVIEW

We will review computers and programming briefly and introduce key features of the LabVIEW graphical programming and the world where LabVIEW is used.

[Keywords] Programming, Text-based language, graphical language, LabVIEW, Physical computing, Arduino, Raspberry Pi, LINX, LabVIEW Community Edition

1.1 Computer and Computer Program

What is programming? A computer program is an instructions manual that tells computers how to perform a task. Let's say you were asked to "Take a memo on what I say. Write that a tomato is $1, meat is $2, and books...will be two books so $20." This memo would look similar to this (Figure 1-1). You were able to complete this task successfully.

Figure 1-1 Shopping list

Let's say we were to ask the PC to do the same. The result is that no memo has been taken (Figure 1-2). Why did this happen? This is because PC cannot act the same way as humans when told to perform a certain task. We humans can understand the instruction and break it down into smaller steps.

(1) To take a memo, prepare a pencil and a notepad
(2) I heard "Tomato is $1" so let's write down "Tomato $1."
(3) I heard "Meat is $2" so write down "Meat $2."
(4) I heard "and books..." so let's first write "Books"
(5) "will be two books so $20" should be written as "Books $20."

Figure 1-2 Result of PC's chore

Just like this. Instead, PC does not understand the action of "taking the memo." Even if it did understand memo taking, PC does not know that it needs to prepare a pencil and a notepad. We need to tell the steps very precisely to PC in order to have it assist us. The instructions that PC can understand are very few compared to humans, so we must break down the instructions more explicitly and explain in detail.

Programming is the act of creating the instruction manual to ask computers to perform a task.

Figure 1-3 Programming

Moreover, we also need to ask computers to perform a task in a specific order. The instructions "Use a pencil to write words," and "Write words using a pencil" are the same for humans, but a PC cannot understand the minute difference between the two. The syntax, or the way to ask PC to perform a task, is specifically fixed, and if we tell it the wrong way, PC cannot perform the task we feed into it.

Furthermore, even if we write the instructions precisely, a PC still cannot understand it. Our language and the language that PC can understand are different. Even a phrase "Write a word" to a PC would be gibberish. Therefore, we must translate the instruction to PC language so that it can read them. This is called "compiling," and

a PC can finally start a task when we compile the instruction manual we created with programming (Figure 1-3).

1.2 Text-Based Language and Graphical Language

In the programming world, it is a tradition to write a program that displays “Hello” on the PC monitor as a first step. Thousands of programming languages can be divided into “Text-based language” and “graphical language.”

Text-based language - C code as an example

```
#include / / Preparation to use display terminal
int main(){/ /Beginning of the program
printf(“Hello, World”);/ / Writing Hello
return 0;}/ / End of the program
```

In text-based languages we program by typing code with the keyboard in text editor such as the Notepad. In this example, “printf” is the instruction to display “Hello, World” on the PC monitor. In text-based languages, there are predetermined writing rules for every type of instructions. Programmers type in the instructions based on these rules. To run the program, we need to compile the code, and once compiled, the program will be executed in the order it is written in the notepad from top to bottom.

Graphical Language – LabVIEW as an example

In graphical languages, we program by placing many different icons on the monitor and connecting the icons with lines (Figure 1-4). In this example, we are programming by connecting the icon with “Hello, World” written on it and the “String” icon with a line. When it is run, “Hello, World” is displayed on the right window. Most graphical programming languages provide their own programming software, and programming is done within that

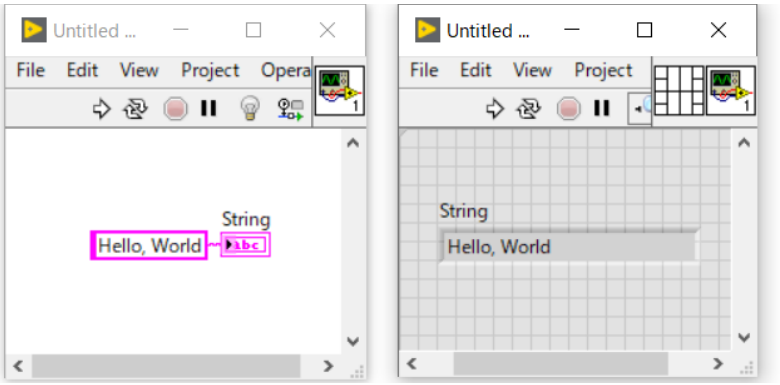


Figure 1-4 Graphical language - LabVIEW

software. This programming software is called “Development Environment.” In graphical languages, icons are provided for each task you ask to the PC, and programmers use the icons and lines to program. In the development environment, a run button is provided, and when you press the button, the program is compiled and run automatically.

Either of the text-based or graphical language is superior than one another. There are pros and cons for each. For example, website design is best handled with a text-based language such as JavaScript, not with LabVIEW. However, a program that measures room temperature every five minutes with a sensor can be best written with LabVIEW. Most programming languages used today are text-based. However, graphical language such as LabVIEW and Simulink (product of Mathworks Inc.) are used very extensively in specific fields such as engineering.

1.3 Physical Computing and LabVIEW

You may not think about programming in your daily life. I don’t believe there are many days where you think to yourself “I felt programming today.” The reason why it is difficult to be conscious about programming is because you do not use it in your daily life. To make PC and programming closer to our lives and broaden the potential of computers, the concept of “Physical Computing” was born. The key point is “bringing PC and programming more practical to our daily life.” This does not mean that you study programming every day; it means improving our daily lives with PC and programming. For example, if you often forget to turn off the lights in your room, you can craft a system that automatically turns off the lights using a PC and electronics kit. Recent gaming hardware allows you to play by shaking the remote controller. This is also an example of physical computing. The remote controller contains a microcomputer that reads the acceleration sensor and displays that data to humans through the game monitor. It is important, even from the point of view of studying programming, to incorporate computers and programming into our daily lives. Studying programming just for the sake of studying would not stick to your mind. If you define a purpose to

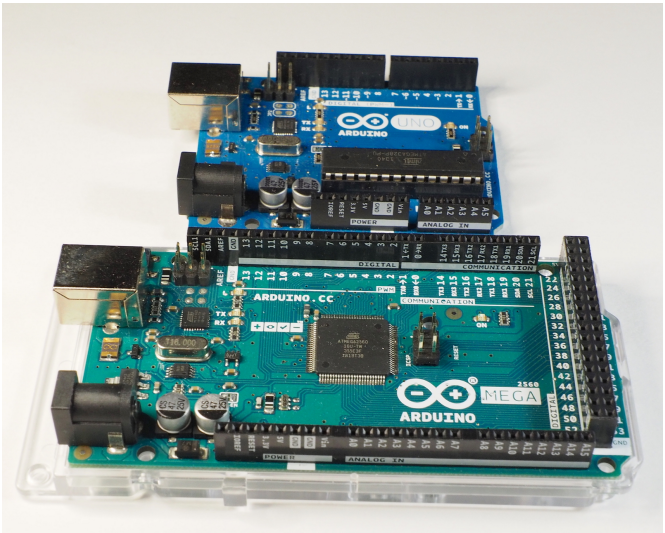


Figure 1-5 Arduino

study, it will accelerate the speed to acquire the knowledge.

LabVIEW is good at handling systems made with electronics kits. When you combine LabVIEW and electronics kits, we are able to venture outside of computers and measure the temperature or turn a light switch on and off. LabVIEW may be the programming language that is the best fit for the concept of physical computing. To connect our physical world to a PC, we need a hardware, a product made with electronics manufacturing. In the previous years, it was difficult to purchase the hardware privately since it was typically expensive, but nowadays, because the notion of "make simple things at low cost" is widespread, we are able to purchase hardware at relatively low price and play with it with programming. Let us introduce two LabVIEW-friendly hardware types.

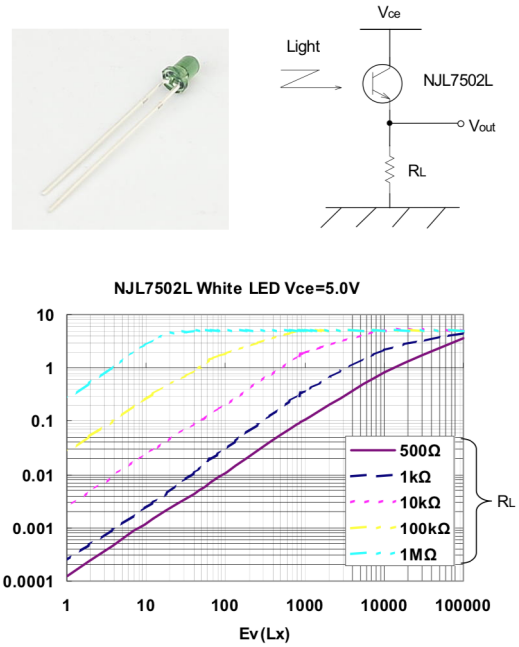


Figure 1-6 An optical sensor

One of them is the Arduino (Figure 1-5). It is a hardware that can be purchased at around 25 US dollars, and it is very popular because of its simplicity and ease of use. Since it has voltage input and output features, it can read the voltage outputted by sensors or turn on an LED. A sensor is an electronics component that changes its output voltage based on condition. For example, an optical sensor changes output voltage based on the strength of light applied to it. By monitoring this voltage, we can read the current strength of light (Figure 1-6). Another hardware is Raspberry Pi (Figure 1-7). This is around 60 US dollars, but unlike Arduino, it is able to put on an OS. An OS is a software needed to operate a PC such as "Windows" or "Mac." In other words, Raspberry Pi is a PC. You are able to purchase a PC for under $100. Of course it is not

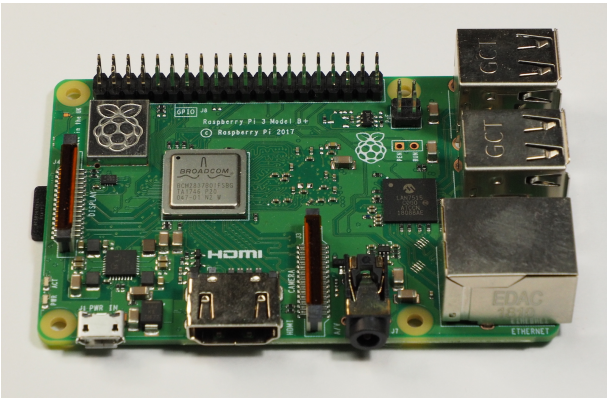


Figure 1-7 Raspberry Pi

a high spec PC, so you would not be able to play the latest computer games, but by having an OS, it is able to perform more sophisticated tasks than Arduino. The reason why these two types of hardware are easy to use from LabVIEW is because LabVIEW has the tools that allows users to use them easily. Usually when connecting a hardware to a PC and using it from LabVIEW, one must perform complicated and laborious procedures. However, LabVIEW provides a tool called LINX that allows users to skip these complicated procedures. This way, users can focus on building what we want to build. By the way, do you know how much LabVIEW costs? When you look up the price on web you may see that it is from several hundred dollars to several thousand dollars. But don't worry. It is free of charge. Previously, when using LabVIEW at work or at home, you were required to purchase the license. However, in 2020, National Instruments decided that for non-commercial use, i.e. using it for private hobby and not for business, anyone could use LabVIEW for free. This version is named "LabVIEW Community Edition," and there is no functional difference between the Community Edition and the paid edition. To repeat, you may not use this free edition when using it for business, and you must purchase a license for LabVIEW. Furthermore, if you wish to use LabVIEW's additional tools (called modules, toolkits or add-ons), you also need to purchase a license as of today.

1.4 The Best Use Cases of LabVIEW

Let us introduce again the fields that LabVIEW best works at. This is a little sophisticated and not directly related to programming study, so feel free to skip this section. LabVIEW works the best at automated test and control combined with hardware. The hardware mentioned here can be test instruments such as oscilloscopes and function generators or familiar items such as cameras and motors. For example, at a machine factory, LabVIEW is used to check if that machine is manufactured correctly without any errors. This is checked by applying voltage to the machine and measuring that voltage to check for errors. In another example, a testing system with camera is used to make sure that the printed words on a bag are printed correctly. As you can see, LabVIEW is used in many companies by combining it with different hardware. Real examples can be viewed on the website below.

http://www.ni.com/innovations-library/case-studies (Select your favorite industry from the "Industry" section on the left side of the page.

Article 1
LabVIEW Heads Towards NXG

In 2017, National Instruments announced "LabVIEW NXG," the next generation of LabVIEW. LabVIEW NXG was not developed based off of LabVIEW and made from scratch. With the similar functionality of LabVIEW remaining, LabVIEW NXG adds new functionalities such as the user-requested zoom in and out of the window, improvement on project management, and web development. Currently in 2020, not all features of LabVIEW is included in LabVIEW NXG, and it is a work in progress. National Instruments promises to invest in the continuous development of LabVIEW NXG, and it will become the mainstream in the near future. Is LabVIEW free now because National Instruments plans to migrate to NXG? Seems not. LabVIEW NXG Community Edition can also be used for free. More on this will be explained in Chapter 2.

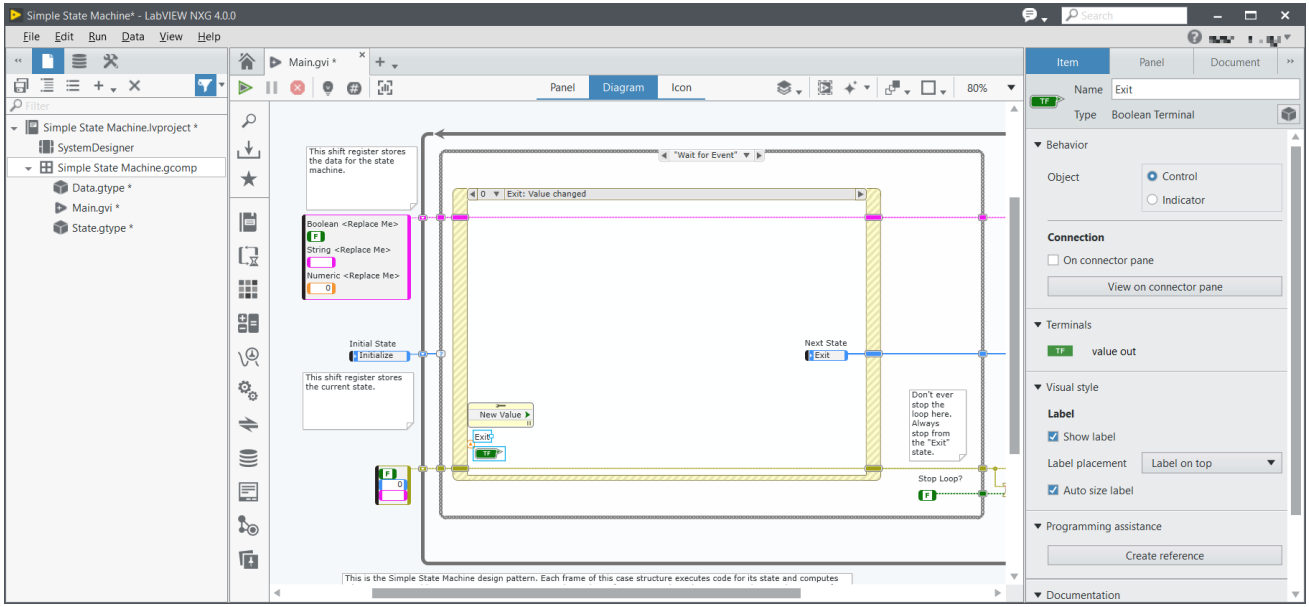


Figure C1-1 LabVIEW NXG programming window

Chapter 2
Using LabVIEW Community Edition

Let us introduce LabVIEW Community Edition and the structure of this document. We will also explain how to download and install LabVIEW Community Edition.
[Keywords] LabVIEW Community Edition, LabVIEW NXG Community Edition, Arduino, Creating User Profile, Downloading LabVIEW Community Edition

2.1 Features of LabVIEW and LabVIEW NXG

LabVIEW was first introduced to the world in 1986. In those days it was customary to develop with text-based language, so engineers first studied the language and after a while they began their actual job. However, the learning process takes too much time, and engineers cannot get their actual job done quickly enough. The first version of LabVIEW was innovated at this time in the United States. LabVIEW was not popular at the time where the text-based language was the mainstream, but as the graphical programming concept became gradually accepted, LabVIEW became the standard for the graphical programming language. This was because for engineers who were not used to text-based language, graphical programming was faster to learn, and they were able to focus on their actual jobs. After around 35 years, LabVIEW has stacked various layers of improvements and feature additions. This includes features suggested by users as well. LabVIEW became a must-have tool for many engineers.

A feature of LabVIEW includes “icons” and “wires.” In LabVIEW, we create a program by placing the icons called “function node” on the window (Figure 2-1).

Each function node has its own roles, and 1 Button Dialog function node shows a message on a window. We can compare using a function node with asking someone a chore such as “shopping” or “writing.” In LabVIEW, data is passed through lines called wires. This “data” is the information needed to ask the PC to

conduct a task. In the comparison with the chore, data is “what” task to conduct. In Figure 2-2, we ask the PC to display (the task) the word, “Hello” (data).

In LabVIEW we create a program by placing the function node and connecting them with the wires. More will be explained in Chapter 3.

In LabVIEW Community Edition, there are two versions, LabVIEW and LabVIEW NXG. At this time, LabVIEW Community Edition is available only in English. LabVIEW NXG, however, already has a feature to switch between languages, so you are able to use NXG with your preferred language. In this text we will use LabVIEW for programming. As we discussed in the previous article, LabVIEW NXG is currently a work in progress. When it becomes more developed, NXG will become the mainstream, so we would like to update this text to use NXG as well.

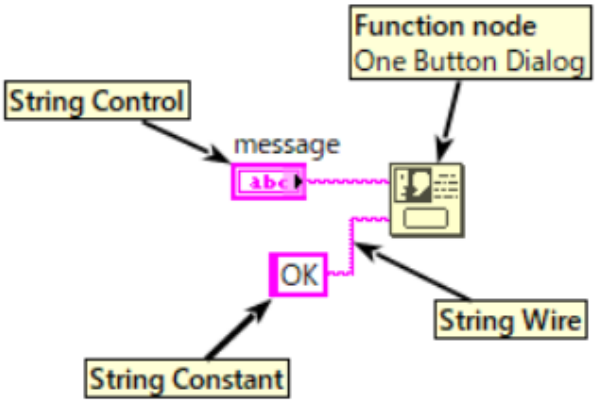


Figure 2-1 Function node

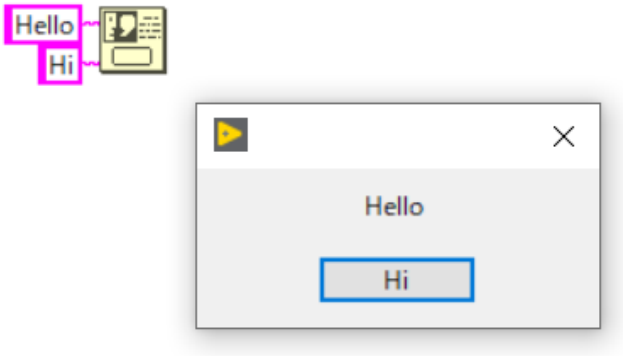


Figure 2-2 Displaying “Hello”

2.2 Structure of this Document

This document is designed for the following audience:

(1) Persons in Junior High School or above
(2) Those interested in doing programming or wants to feel programming closer
(3) Those who want to work with electronics kits

There are three parts overall and are divided into chapters.

(1) In Part 1, we introduce LabVIEW and conduct some preparation to use LabVIEW.
(2) In Part 2, we provided you with a subject to learn LabVIEW programming with just a PC. In Chapter 3, we will study the ways to control and program in LabVIEW using a simple application named "LED and Current Limiting Resistor.vi." We create a current limit resistance calculation program that model real LED properties while studying ways to implement arithmetic operations and numeric comparisons and design an LED simulator app.
(3) In Chapter 4, we will use the program called "SoundVIEW" that saves the sound taken from PC microphone or PC application and perform frequency analysis called spectrogram and observe how this program is made. We will then create a program that display sound data to a graph or play it in reverse to study how to input and process data.
(4) In Part 3, we will use Arduino UNO and some electronics parts to learn programming.
(5) At the beginning of Chapter 5, we will learn how to use Arduino. You will experiment with the push switch and DC fan to observe digital input and PWM output.
(6) In Chapter 6, we will create a program that measure the Voltage-Current property of an LED using an LED, a resistor, and a variable resistor. As an example of data analysis, we will use linear regression to calculate the coefficient of the PN junction of an LED.
(7) In Chapter 7, we will create a program that use the Arduino as an interface for MAX30102 module to acquire the blood flow data and calculate the heartbeat. I2C connection will be mentioned. We will also need to solder the pin headers.

2.3 Installing LabVIEW Community Edition

Let us introduce the steps to install LabVIEW Community Edition. First create a National Instruments user profile. This includes steps to input your personal information, so do this step with your parents or guardian if you are underage.

1. Access the National Instruments web page.
   https://www.ni.com
2. Click “Log in” at the top right corner (Figure 2-3).
3. Click “Create Account >” (Figure 2-4).
4. Enter the required credentials and click “CREATE ACCOUNT” (Figure 2-5).

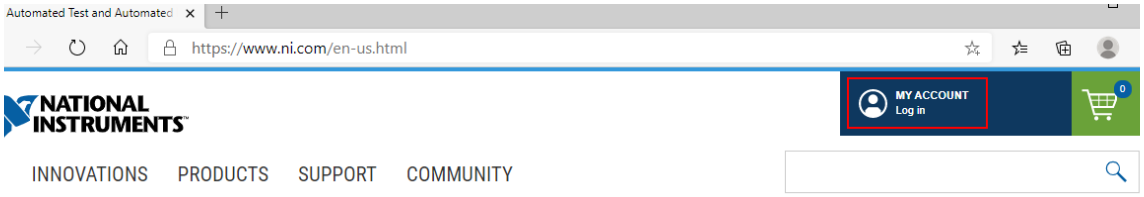


Figure 2-3 Login

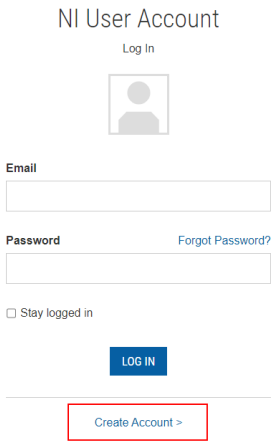


Figure 2-4 Create user profile

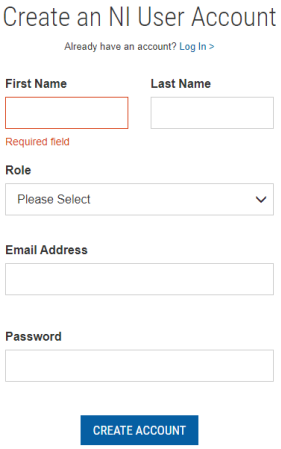


Figure 2-5 Create user profile (input)

After you created your user profile, let's install LabVIEW Community Edition.

1. Access the website below and click on LabVIEW Community Edition link (Figure 2-6).
   http://www.quatsys.com/labview/1109/lvproraku.jp.html
2. Click "DOWNLOAD NOW" button and select "LabVIEW 2020 Community Edition (Figure 2-7).

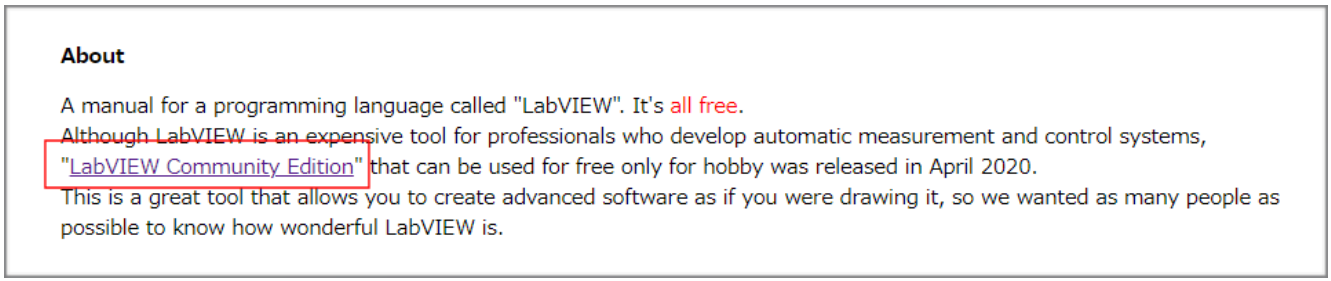


Figure 2-6 Link to LabVIEW Community Edition

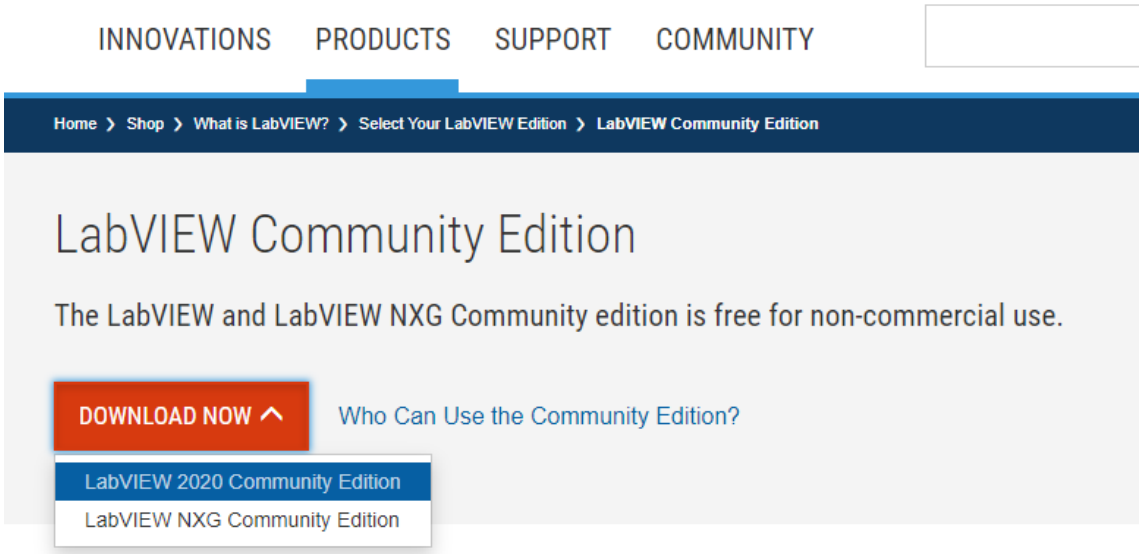


Figure 2-7 Downloading window

3. You will move to the download page. Click “DOWNLOAD” button at right of the page. You need to be logged in with the user profile you created (Figure 2-8).
4. You will be asked about how to treat the file, so select “Save” and save the file to a folder of your choice (Figure 2-9). (This will look different on browsers other than Microsoft Edge, but please save the file to the PC one way or another.)

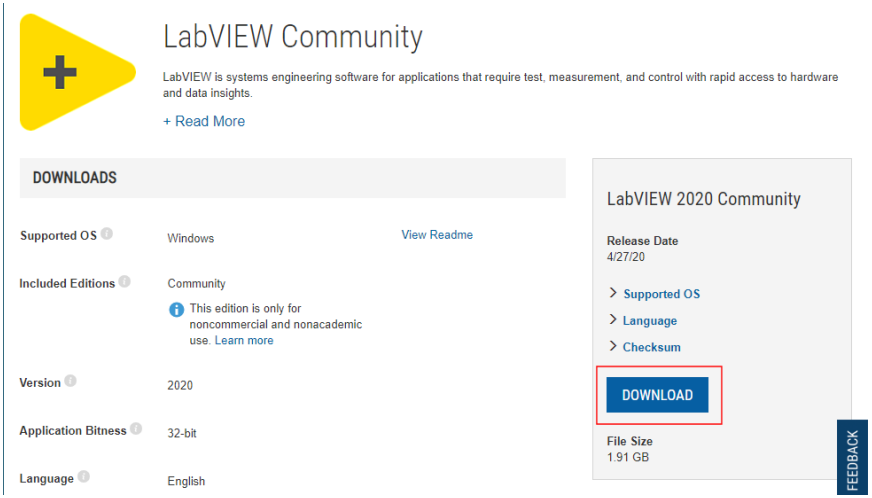


Figure 2-8 Download website

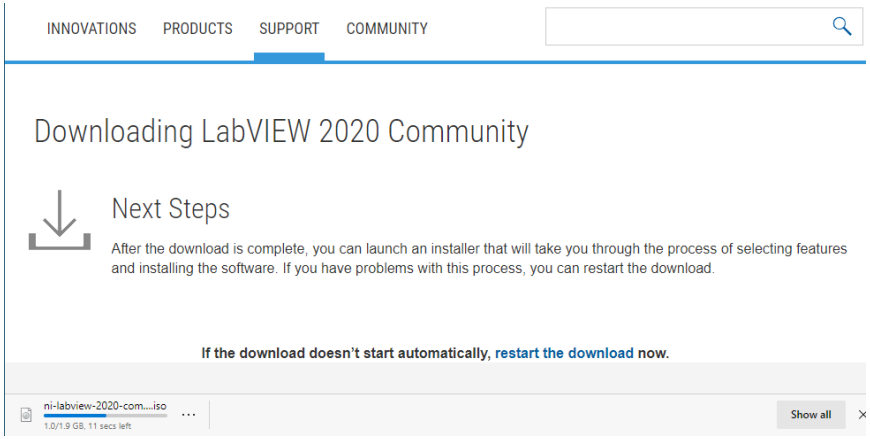


Figure 2-9 Begin downloading

5. When finished, “ni-labview-2020-community-x86-_xxx.iso” will be downloaded. Right-click and select “Mount” (Figure 2-10).
6. When mounting is finished, a virtual DVD drive will be made and all the files in the ISO file will be displayed. Double click on “Install.exe” and run it (Figure 2-11).

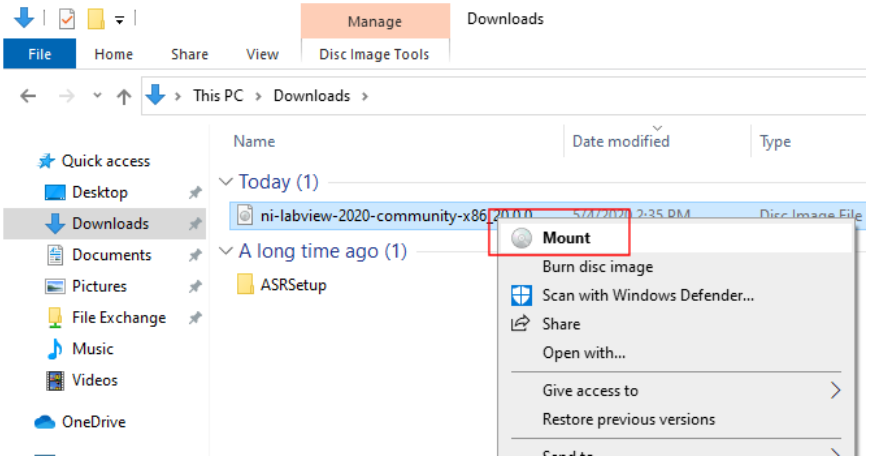


Figure 2-10 Mounting the ISO file

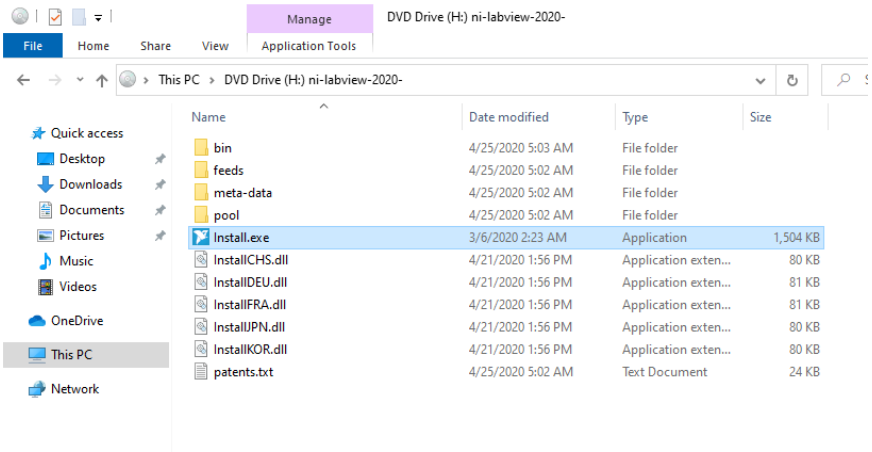


Figure 2-11 List of files in Install folder

7. When “Install.exe” is run, “NI Package Manager,” the software necessary to manage the National Instruments software, will be installed. Read the license agreement and if you agree, select “I accept the above license agreement” and click Next (Figure 2-12). Click Next a few times to begin the installation (Figure 2-13).

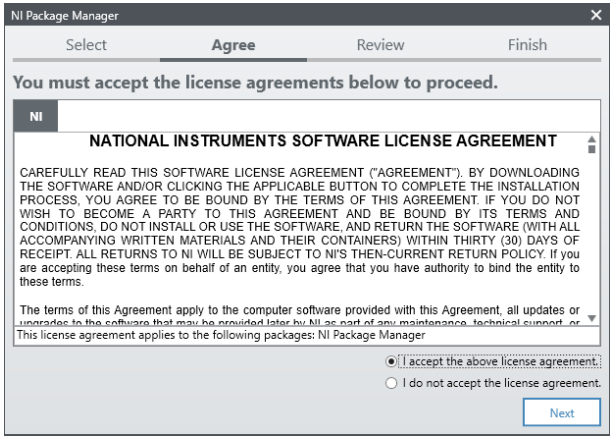


Figure 2-12 NI Package Manager License Agreement

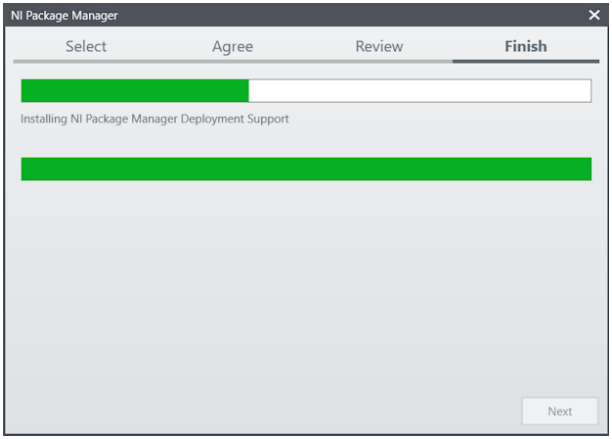


Figure 2-13 Begin installing NI Package Manager

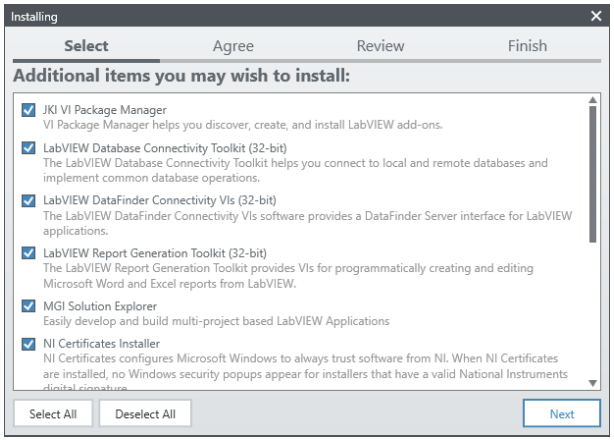


Figure 2-14 Selection window for LabVIEW components

8. When the installation of NI Package Manager is complete, it will move onto LabVIEW installation window. Here we select add-on packages to LabVIEW. Leave everything selected, and click Next (Figure 2-14).
9. As in NI Package Manager, the license agreement will be displayed. Read the license agreement and if you agree, click Next (Figure 2-15, Figure 2-16).
10. You will be asked once more if all the installation components are correct. Click Next to begin the installation (Figure 2-17).

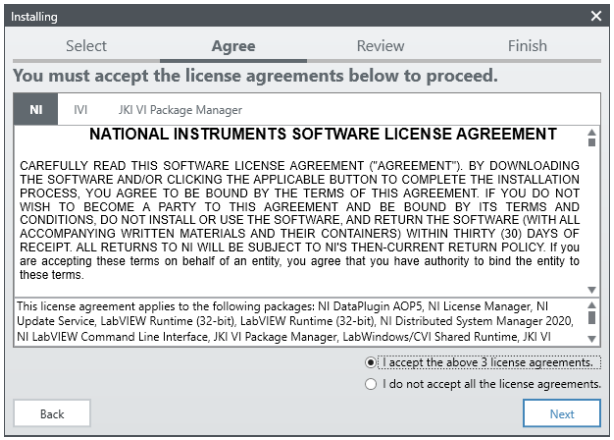


Figure 2-15 LabVIEW license agreement

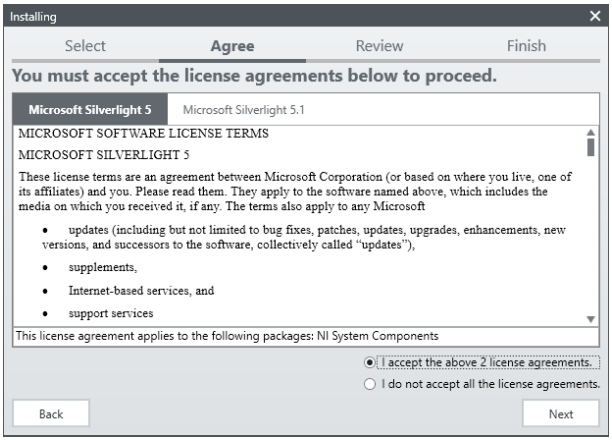


Figure 2-16 Microsoft license agreement

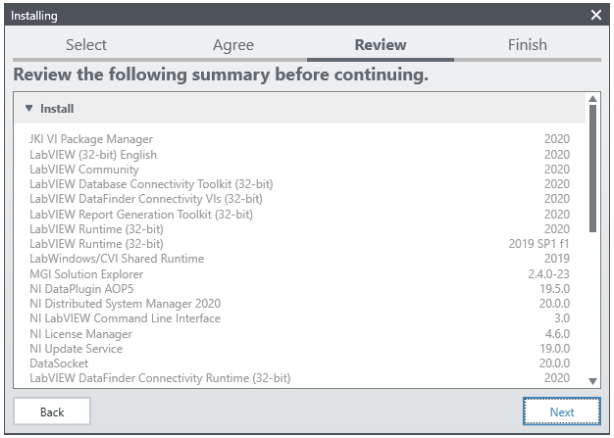


Figure 2-17 Final check for installation components

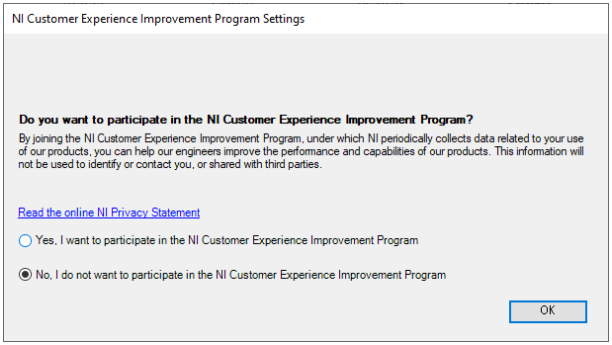


Figure 2-18 NI Customer Experience Improvement Program

11. When installation is complete, you will be asked if you would like to join “NI Customer Experience Improvement Program.” You can either join or not join, so select either option and click the OK button (Figure 2-18).
12. Next we will need to “activate” the software (Figure 2-19). In order to use LabVIEW, you will need to login with the user profile you created and perform activation. Click “LOG IN TO ACTIVATE” and a new window will be opened (Figure 2-20). Login with your credentials here.
13. Once logged in, we will activate LabVIEW. Check to make sure “Check my account for licenses” is selected and click “ACTIVATE” (Figure 2-21). In your user accounts, you already have the license for LabVIEW Community Edition, so there are no other special steps required for activation (Figure 2-22).
14. Lastly, restart your PC. Make sure to save the files if you have any other software open on your desktop and reboot (Figure 2-23).
15. This completes the installation steps for LabVIEW. Go to Windows start menu to make sure that LabVIEW is properly installed (Figure 2-24).

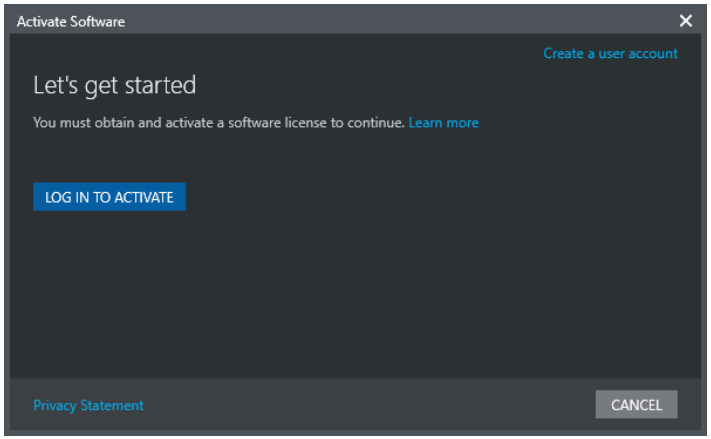


Figure 2-19 Activation window

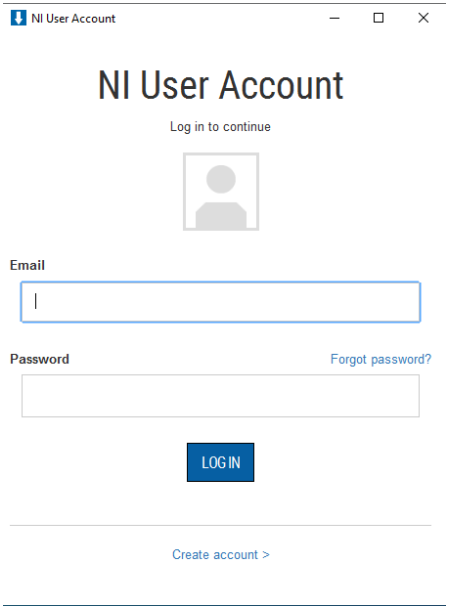


Figure 2-20 Input user profile

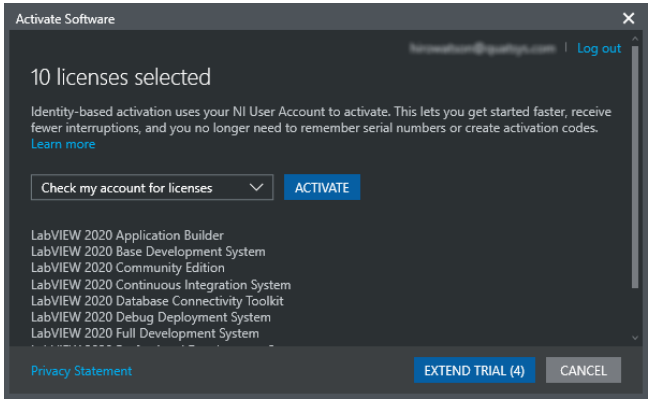


Figure 2-21 Activating license

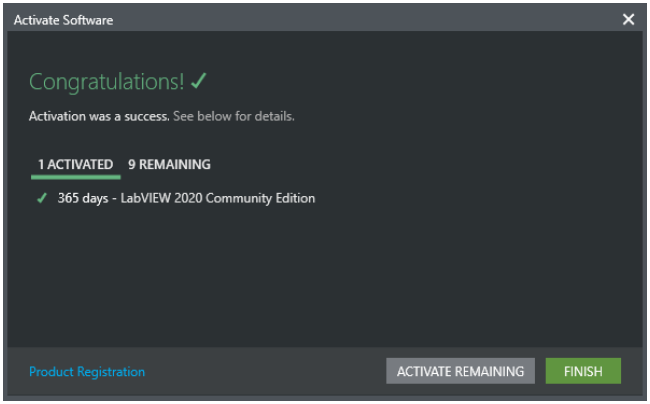


Figure 2-22 Activation complete

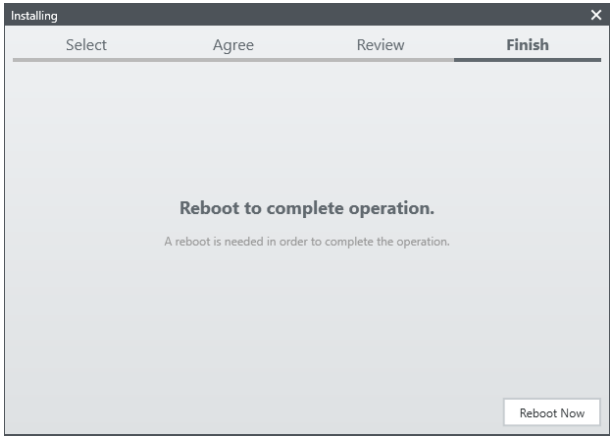


Figure 2-23 Reboot

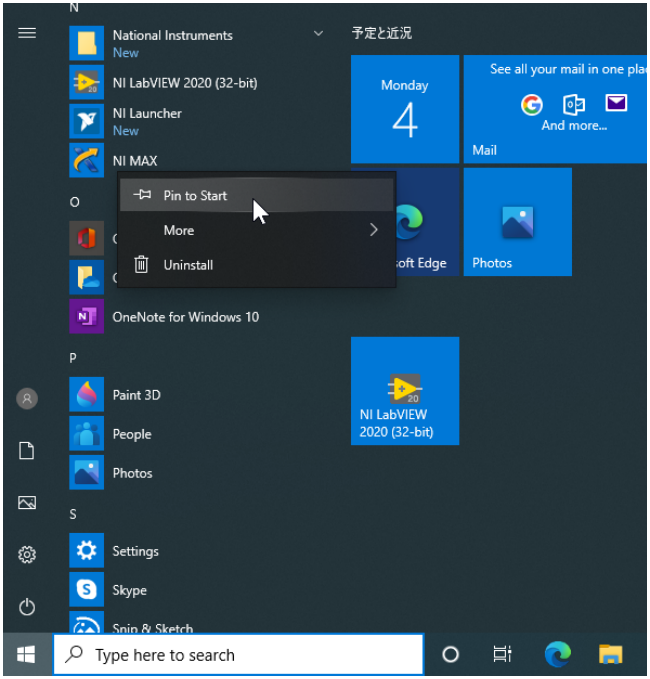


Figure 2-24 Windows Start menu

Article 2
Installing LabVIEW NXG Community Edition

The steps to install LabVIEW NXG Community Edition is almost identical to installing LabVIEW. Go to the link for LabVIEW Community Edition and select "LabVIEW NXG Community Edition" to receive the installer (Figure C2-1).

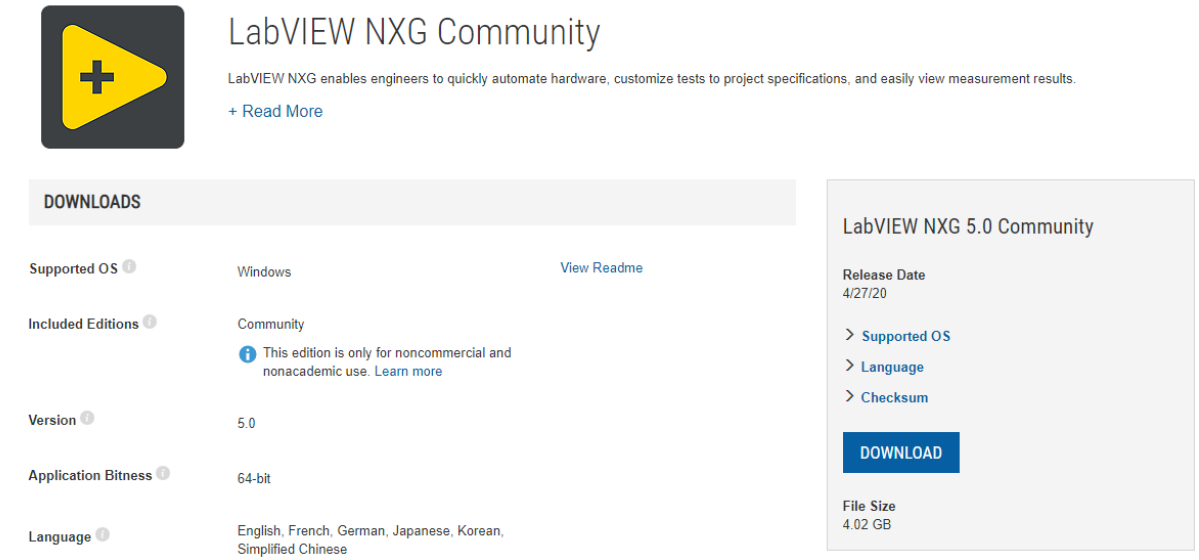


Figure C2-1 Download Link to LabVIEW NXG Community Edition

Part 2
Getting Started with LabVIEW Programming

Chapter 3
First Exposure to Graphical Programming

In this chapter we will look at the basics of LabVIEW programming and elements specific to LabVIEW programming.

[Keywords] Front panel, Run button, Run Continuously button, Abort Execution button, Pause button, Control, Indicator, Block diagram, Control, Indicator terminal, terminal and wire color, Highlight Execution button, data flow, VI icon, Help, Sub VI, New VI, Ctrl+T, Control Palette, Functions Palette, Drag and drop, Wiring tool, Properties, Numeric Control, Numeric Indicator, Boolean Control, Bool Indicator

3.1 Investigating the LED Properties in LabVIEW

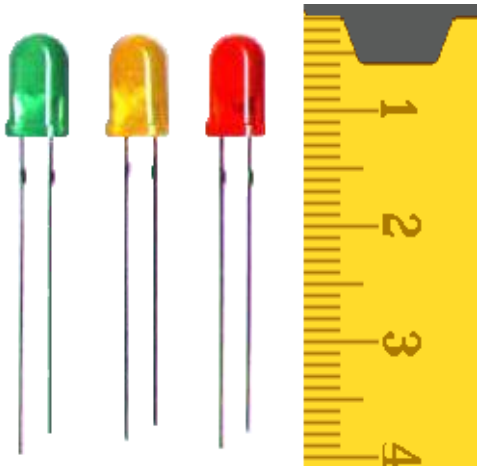


Figure 3-1 Green LED, Yellow LED, Red LED

LED (**Figure 3-1**) is short for **Light Emitting Diode**, and it is a device that emits light when current flows through. LED has advantages such as durable life and energy efficiency compared to traditional light bulbs. In 2014 Nobel Prize for Physics was awarded for the invention of the blue LED. Today LED is used in everyday life, even around you. We will explain the principles of the LED with the help of LabVIEW. LED has a direction, and current flows only from positive side to negative side. The longer side is positive and is called **anode** and the shorter side is negative and is called **cathode**. When the voltage of the anode is greater than cathode, current flows from anode to cathode. Inside of the LED, positive ion and negative electrons are separated, and when current flows through, these ions collide and the energy emitted by the collision is show to our eyes as light (**Figure 3-2**). **Figure 3-3** is the circuit diagram including the LED. When we use LabVIEW, we can simulate the circuit with programming instead of building the circuit in real life. That way, there would be no need to change resistor parts to adjust the resistance or prepare multiple batteries for replacement. This process of recreating real-life with programming is called **simulation**. Simulation is used to develop cars and airplanes as well.

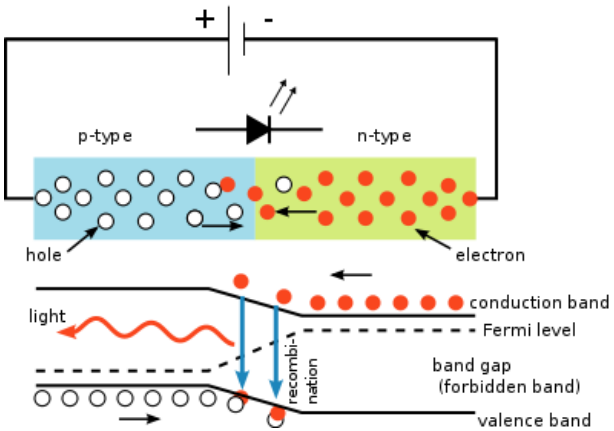


Figure 3-2 Principle of LED luminance

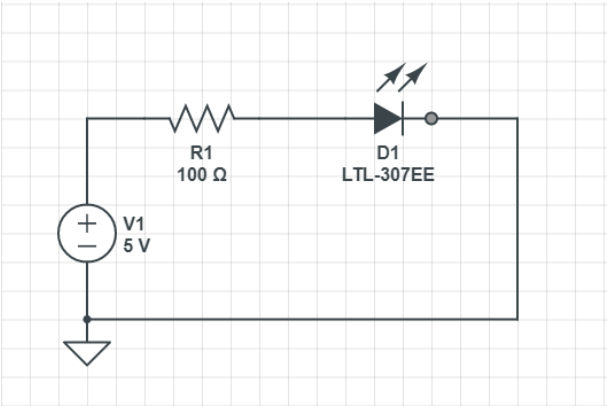


Figure 3-3 Circuit diagram

Let's boot LabVIEW. From the Windows Start menu, select **NI LabVIEW 2020 (32bit)** to boot LabVIEW (**Figure 3-4**). Go to the example folder for Chapter 3 and open **LED Simulator.vi**. The window displayed should resemble **Figure 3-5**. Let's run the program to observe the LED behavior. In LabVIEW we call a program a "**VI**" ('vi 'aɪ). Click the Run button (white arrow) at the top left of the window to execute the VI. When executed, the LED turns in red color (**Figure 3-6**). Next, turn the "**Current Limiting Resistor (Ohm)**" dial to 500 and run the VI. Notice that the LED light is a little dimmer. Clicking the Run button every time we try a different dial is cumbersome, so click the Run Continuously button next to the Run button to let the VI run repeatedly automatically. To stop the VI, click the Run Continuously button again.

We will explain each objects on the window. "**Power-supply voltage**" determines the voltage of the battery. Typically either of 5/3.3/1.8 V is used, so the VI is designed for you to choose from these three options. "**Current Limiting Resistor (Ohm)**" limits the amount of current that flows to LED. When more current flows into LED, the brighter that LED will light, but too much current flow cause electrons

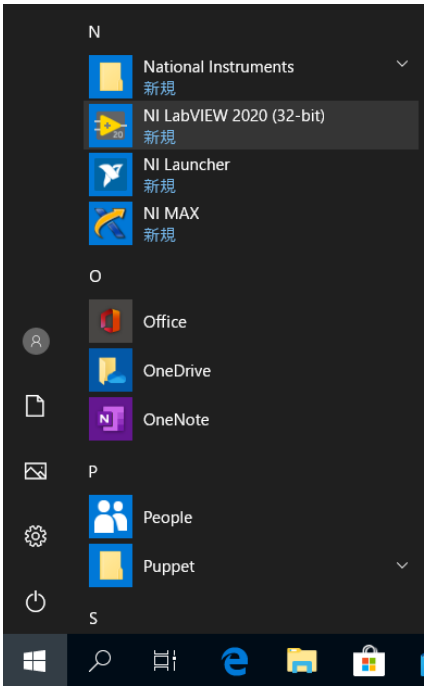


Figure 3-4 Booting from Windows menu

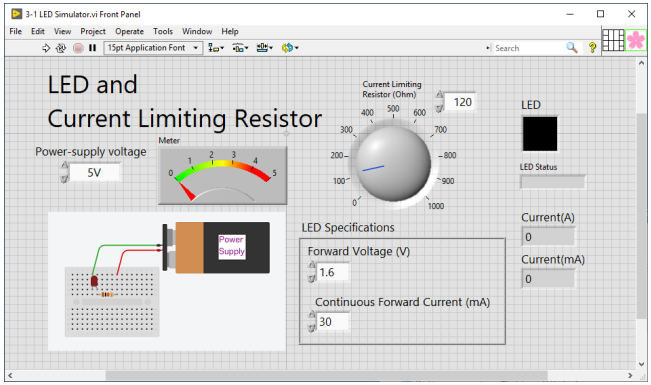


Figure 3-5 Front panel for LED and Current Limiting Resistor

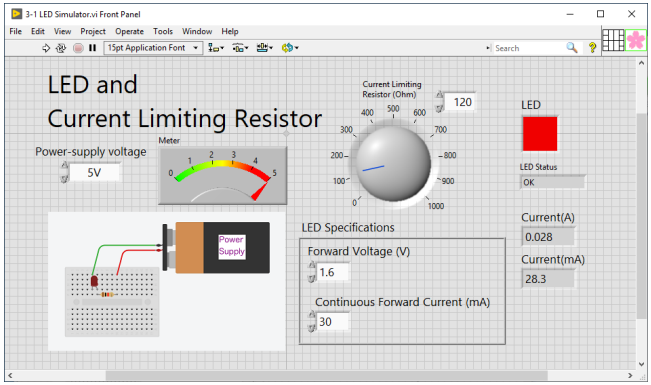


Figure 3-6 Result of execution for initial state

within LED to bounce into each other too strongly and will break LED. To avoid this we use a resistor to control the current flow. The more resistance it has the less current flows into an LED. "**Forward Voltage**" is the minimum voltage needed to light an LED. If voltage is too small, electrons bounce too slowly and will not release enough energy to light an LED. Finally, "**Continuous Forward Current (mA)**" is the amount of current flow that breaks the LED. As an example, settings shown in **Figure 3-7** displays "LED Broken", indicating that LED has broken. If you create this circuit in real-life, you may have broken your LED. However, you can use whatever settings in simulation and a real LED will not break. This is a reason why simulation is used when companies develop new products such as electronics, automobiles, and airplanes.

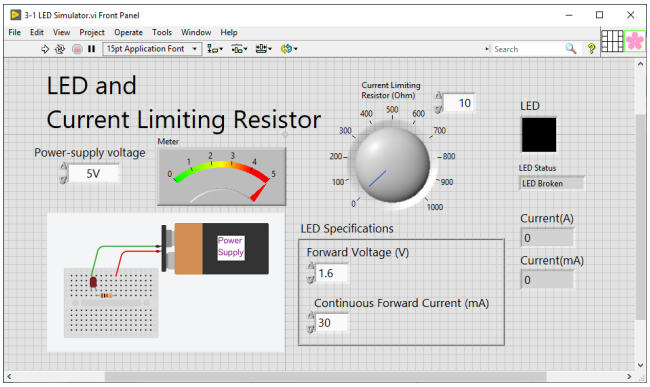


Figure 3-7 LED broken

Try out different settings with LED Simulator.vi to get used to using LabVIEW.

(1) At what value of Continuous Current (mA) will LED break?

(2) When "**Power-supply voltage**" is changed from 5V to 1.8V, how should you modify other settings so that LED turns in the same color as when the voltage was 5V? (There are multiple solutions)

(3) How could you have avoided the breaking of LED? What can be modified to avoid the damage?

3.2 Observing the LabVIEW Program

Now let's look more closely at how this LabVIEW program is made. The window you have been seeing is called **Front Panel**, and it is a window used to modify settings or view the return value of the program. In programming we call such window a **User Interface (UI)**. An interface is a connection between something and something else, and in case of programming, UI is a connection between a user and a program (**Figure 3-8**). Select from the menu bar of LabVIEW "**Window > Show Block**

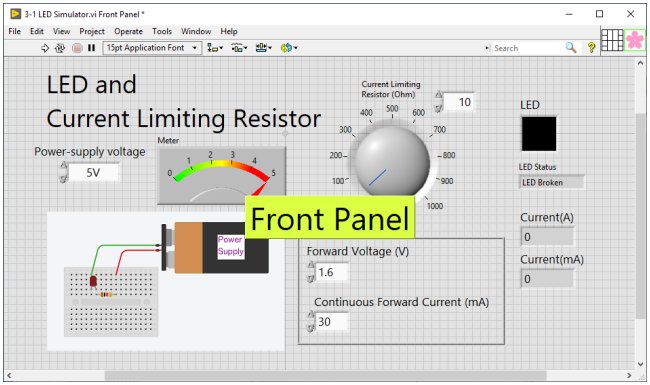


Figure 3-8 Front panel

Diagram," and you will see a window with a white background. This is called **Block Diagram**. In LabVIEW, we write the program in the block diagram and use the program from the front panel (**Figure 3-9**). You can see from the block diagram how colorful it is. In LabVIEW a color is assigned to each type of data, i.e. orange or blue from numbers, pink for text, etc. You can also see that objects with a same name is present both in front panel and block diagram. We call them on the block diagram **terminals**, and almost every objects in the front panel are coupled with these terminals in the block diagram. Objects with yellow background are called **functions**, and they perform actions such as division or comparison between two values. Let's observe how

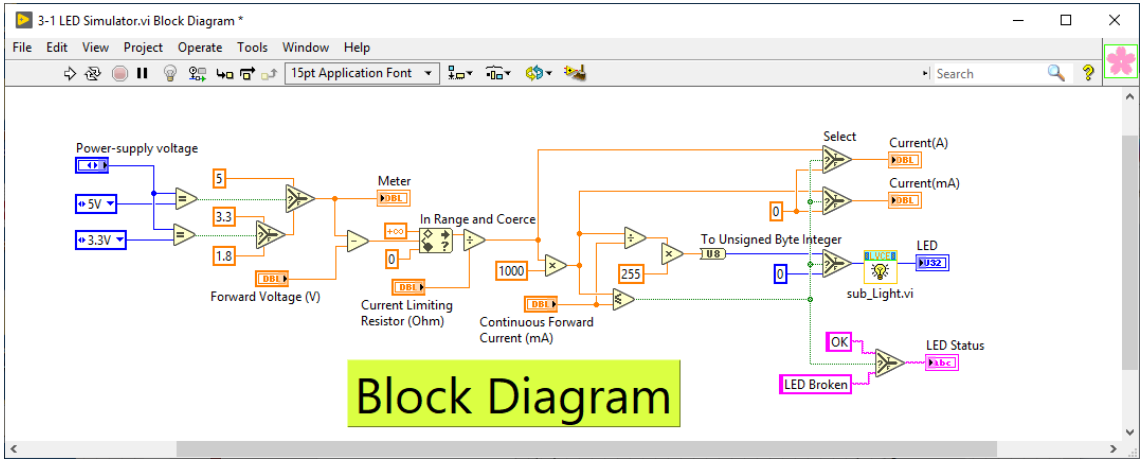


Figure 3-9 Block diagram

this program works. Click on the **Highlight execution** (**Figure 3-10**) and the lightbulb will turn on. Run the VI, and you will see the program data flowing in animation. Observe how the value set on the front panel is displayed on the terminal and that the front panel objects and the corresponding terminals are paired. In LabVIEW, we program by connecting the terminals and functions with wires. In this example, the program is executed from left to right, but in LabVIEW we can control the execution flow of the program (ex: top to bottom, right to left,

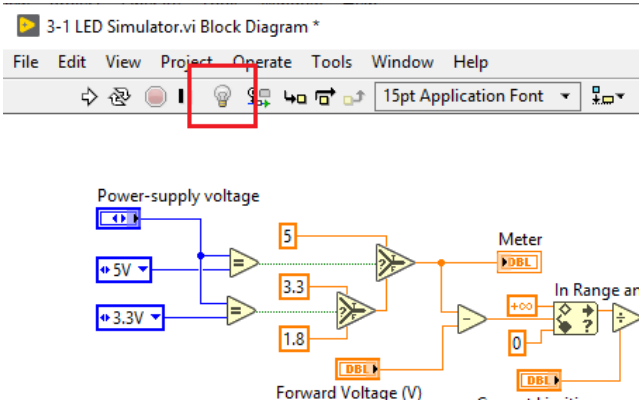


Figure 3-10 Highlight execution

etc.) by modifying the wiring. However, it is customary to program LabVIEW from left to right. When you are finished observing the program behavior, click on the x button on the top right of the front panel to finish the VI. Note that clicking the x button on the block diagram just closes the block diagram only and does not finish the VI. If you see the popup similar to **Figure 3-11**, select "**Don't Save**."

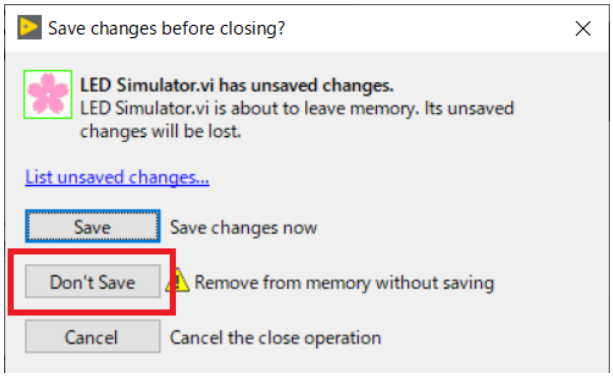


Figure 3-11 Save dialog

3.3 Creating a Simple LabVIEW Program

Let's now create a VI from scratch. We will take the LED program as an example. Fist, close all VIs and open the LabVIEW startup window (**Figure 3-13**). To create a new VI, select "**File > New VI**" as shown in **Figure 3-13** or press **CTRL** and **N** keys on the keyboard at the same time. You can use your mouse along with the keyboard to enable you to control LabVIEW quickly. Now that you made a new VI, let's take a closer look (**Figure 3-14**). To the left of the screen, the window with gray background is the Front Panel, and we use this to control the program with buttons or observe the measurement data with a graph. It is empty now, but we can add buttons and graphs to create a control panel. To the right of the screen with white background is the Block Diagram, and we use this to create the program. To begin your LabVIEW journey, let's first create a program that performs mathematical operations such as addition and multiplication. We will place four necessary parts on the front panel. On the front panel, right-

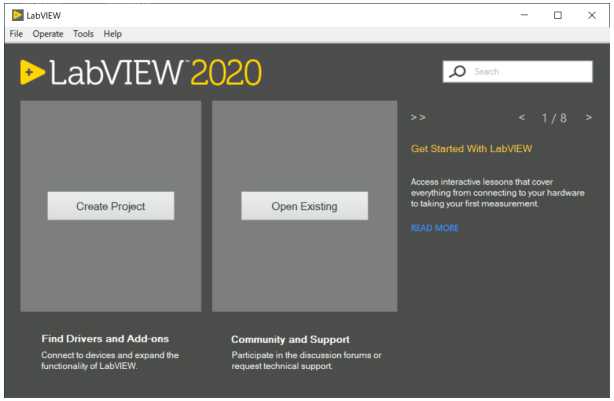


Figure 3-12 Boot menu of LabVIEW

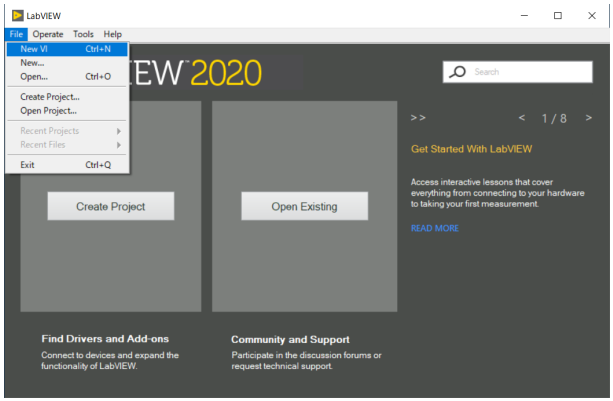


Figure 3-13 Creating a new VI

click on the mouse. The **Controls Palette** will be displayed (**Figure 3-15**). We will choose the necessary parts from this palette and place them onto the front panel. You may already have the palette displayed without right-clicking the front panel. You can close this palette and have it reappear by right-clicking on the front panel.

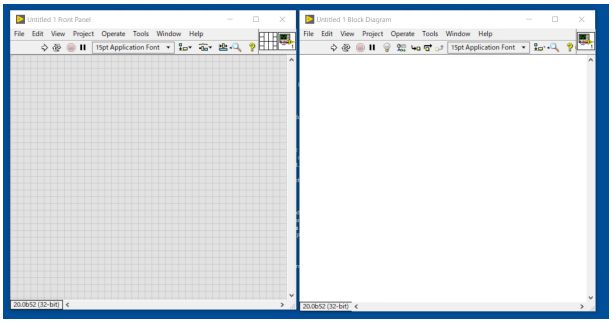


Figure 3-14 Front panel and block diagram

Place the mouse over **Numeric** icon, and you will see many parts related to numbers. Left-click on the **Numeric Control** (**Figure 3-16**). As you can see on **Figure 3-17**, you will be able to grab the Numeric Control and move it. Left-click on the front panel again, and you are able to place it on the front panel (**Figure 3-18**). In LabVIEW, we create the UI window by selecting the parts from the palette and placing them on the front panel. Similarly, place the **Numeric Indicator** (**Figure 3-19**). Notice that they look different. Numeric Control is belongs to the **Controls** parts group, and the Numeric Indicator belongs to the **Indicators** parts group. The Controls allows users to input data into the program by pressing buttons or inputting numbers. We use indicators to get feedback from the program such as viewing the calculation result or graphs. Keep this momentum and add two more numeric controls to set up the total of three numeric controls and one

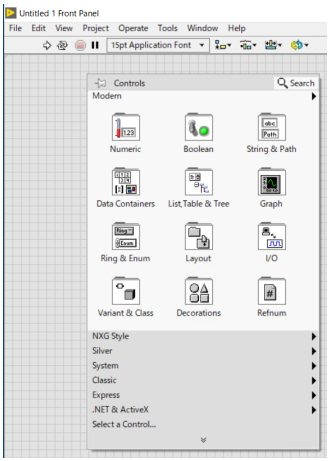


Figure 3-15 Control palette

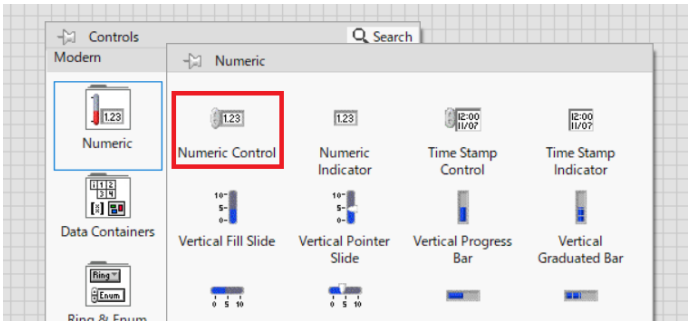


Figure 3-16 Numeric Control

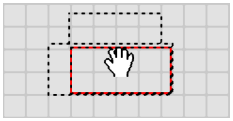


Figure 3-17 Selecting the numeric control

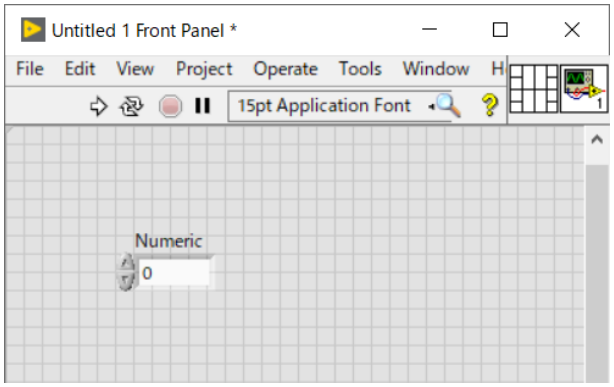


Figure 3-18 Placing the numeric control

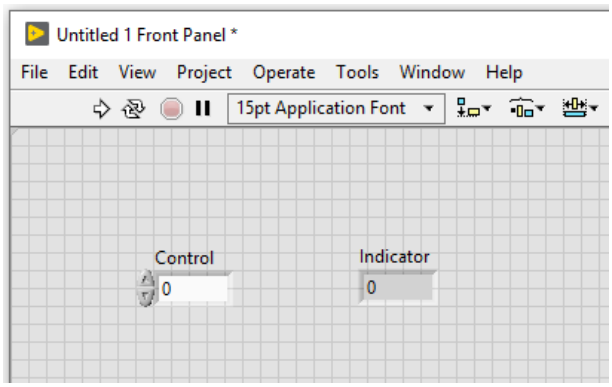


Figure 3-19 Control and indicator

numeric indicator (**Figure 3-20**). You can change the name of the controls and indicators by double-clicking on them. Since we cannot decipher what each control does when they all have the name **Numeric Control**, so name them as shown in **Figure 3-21**. In the programming world, it is important to name everything correctly. We cannot understand the purpose of each program modules if they are not named, and we cannot use them if we cannot understand them. By including the necessary information in the name, we are able to create a

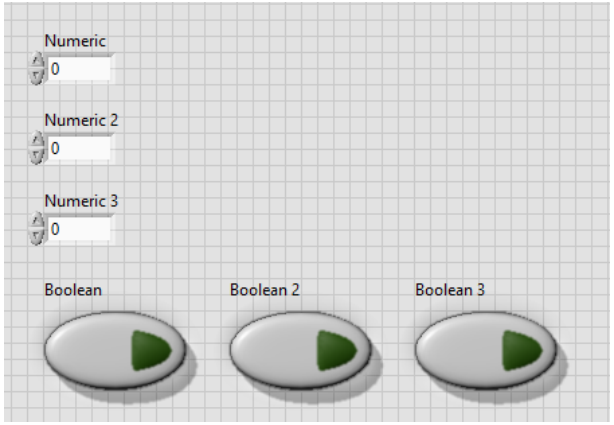


Figure 3-20 We cannot understand if unnamed

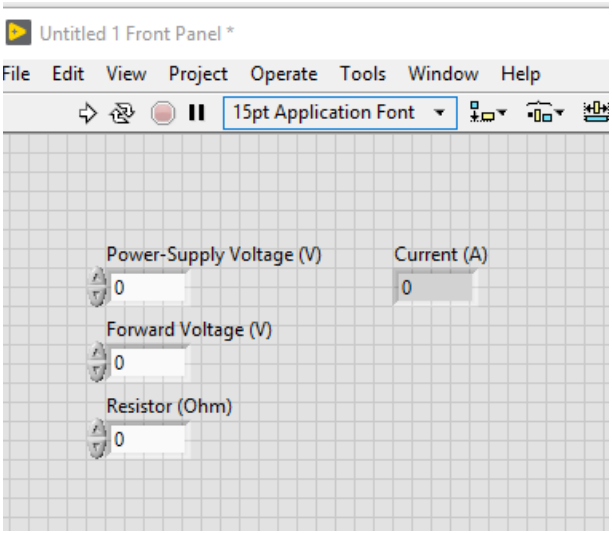


Figure 3-21 Front panel in current state

usable program. We have now completed the steps to create the front panel. Next let's look at the block diagram, We can switch between the front panel and the block diagram by pressing **CTRL** and **E** keys (**Figure 3-22**). This is the first time we open the block diagram, but you notice that some objects are already put on it. When we placed controls and indicators on the front panel, these objects were automatically created. We call these objects **Terminals**. A control or an indicator and a terminal are in pairs. The Numeric Control "**Power-Supply Voltage(V)**" is paired with the "**Power-Supply Voltage(V)**" terminal on the block diagram. The value you input on the front panel will be outputted from this terminal. Let's take a look.

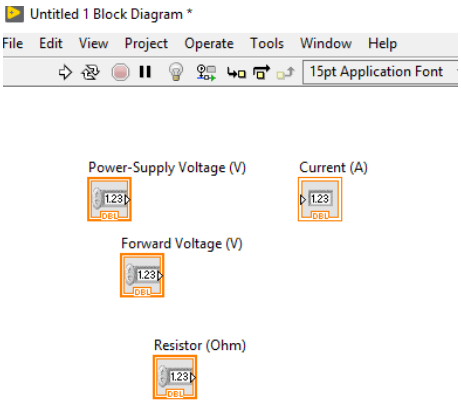


Figure 3-22 Block diagram in current state

Looking at the terminals, you may notice that the terminal you just made look different from the ones in the example VI (**Figure 3-23**). Don't worry, they are the same, yet they look different. Right-click on the terminal and uncheck **View As Icon** to make the terminal thinner. If you want the terminal to always look this way, go to the menu bar and select "**Tools > Options**" to display the options window (**Figure 3-24**) and uncheck "**Place front panel terminals as icons**" in the Block Diagram category. Let's set a value in the control and display it on an indicator. To do this we need to connect "**Power-Supply Voltage(V)**" control and "**Current (A)**"

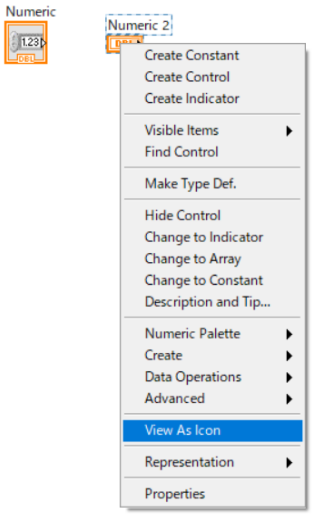


Figure 3-23 Terminals

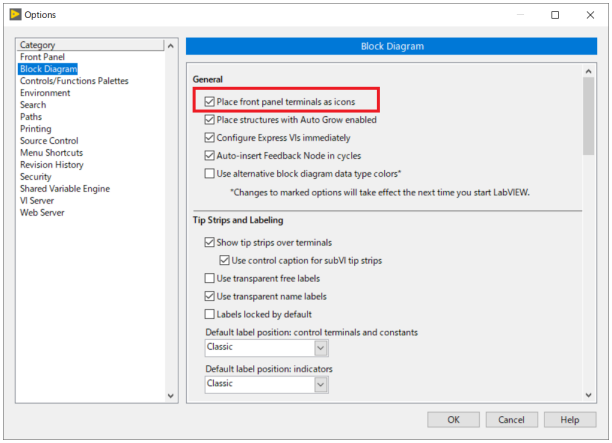


Figure 3-24 Options window

indicator. In LabVIEW we connect these with the wire. Hover the mouse cursor over the "▷" at the right of the control terminal. The cursor changes to the bobbin icon (**Figure 3-25**). This indicates that we can connect the wire. Left-click here and move the mouse. Then you will see the dotted line follows the cursor (**Figure 3-26**). As you can see, we connect the terminals with wires to transfer the data (**Figure 3-27**). After you connect the wire, put a value in the "**Power-Supply Voltage(V)**" control and run the VI. You can also run the VI by pushing **CTRL** and **R** keys together. Did the value you put on the control get displayed on the indicator? Then delete the wire you just created. To delete, click on the wire and once it is surrounded by the dotted lines, press the **Backspace** or **Delete** key. Now, let's use functions to modify the program. Right-click on the block diagram. You can see that just like on the front panel, a palette is displayed but it has a different look (**Figure 3-28**). This is called **Functions Palette** and we can use various functions that are included here. To use functions in functions palette, we drag and drop just like we did on the front panel. This time we will create a program that calculates the current flow into LED based on the power voltage, forward voltage and resistance. Let's consider the calculation formula before we program. From **Ohm's Law**, current is voltage/resistance. Voltage is the amount of power voltage, but we must not forget about forward voltage. The actual amount of voltage applied to the resistor is forward voltage subtracted from power voltage. Since forward voltage is stable even when the current changes, we will assume that forward voltage remains the same and use only the resistance to calculate. Therefore, the formula will be Current flow into resistor and LED = (power voltage – forward voltage) / resistance. Let's create the program. First we will conduct subtraction. Display the functions palette and select **Subtract** from **Numeric** palette. Place it onto the block diagram (**Figure 3-29**).

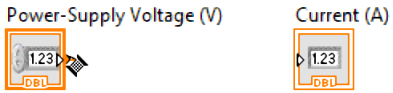


Figure 3-25 Bobbin icon

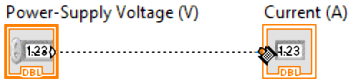


Figure 3-26 Wiring

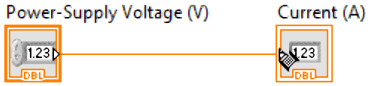


Figure 3-27 Wiring complete

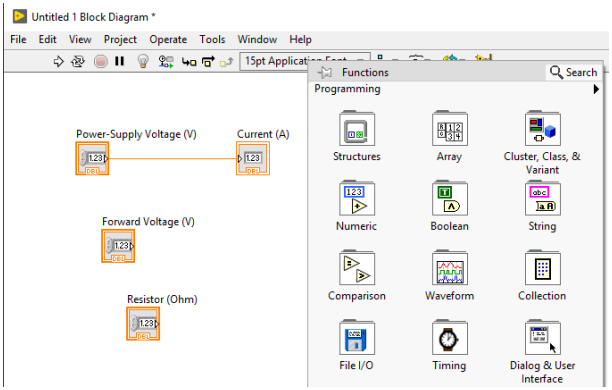


Figure 3-28 Functions palette

Let's look at how to use this function. Press the **CTRL** and **H** keys. **Context Help**, or the window that

tells you how to use the function, will be displayed (**Figure 3-30**). Press the same shortcut keys again and the window will disappear. As you can see, if we input **x** and **y** on the left, the subtract function outputs the value **x-y** from the right. Context Help shows the overview on how to use a function. For more information, click on **Detailed Help** to display the help document with additional content. Now, let's wire to the **Subtract** function the "**Power-Supply Voltage(V)**" control to top left and "**Forward Voltage(V)**" control to bottom left. Wire them the same way as we did for wiring the control and the indicator. Move the mouse cursor on **Subtract** function and you will see orange circles. Place the cursor over that orange circle, and the cursor changes to the bobbin icon and you will be able to create wires (**Figure 3-31**). Left-click on it and move the mouse to the "▷" on the right side of the "**Power-Supply Voltage(V)**" control and left-click again to create the wire. Do the same for "**Forward Voltage(V)**" control and wire it. Your block diagram should look like **Figure 3-32**. Finally, let's place the division. We will use **Divide** function. You will find this on the right of **Subtract** function in the

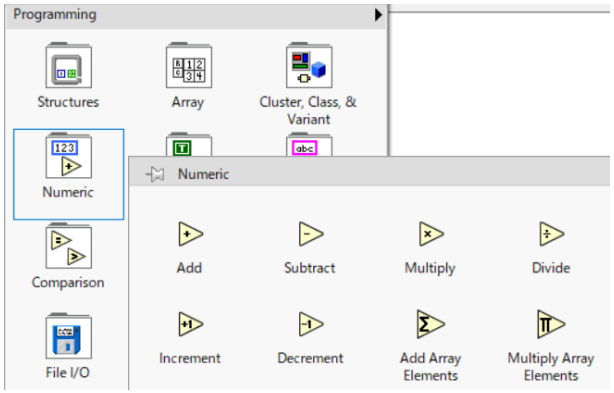


Figure 3-29 Subtract function

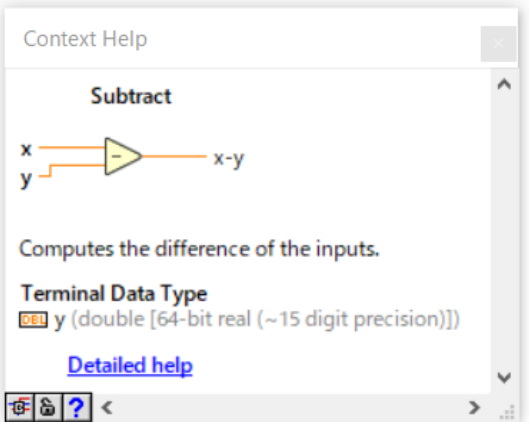


Figure 3-30 Context Help for Subtract function

Figure 3-31 Wiring to Subtract function

functions palette. Wire the function and the control and indicator the same way we did for subtraction. It should now look like **Figure 3-33**. Input values on the front panel (**Figure 3-34**) and run, and you will be able to check how much current flow through the LED.

By the way, we are currently able to input negative values into forward voltage, but this is a little strange. Because forward voltage is the minimum voltage needed to light the LED, it will never be negative. In LabVIEW, we can limit the value to be inputted into a control. Right-click on "**Forward Voltage (V)**" control and select **Properties** at the very bottom of the right-click menu. When the properties window is displayed, select **Data Entry**. Click **Use Default Limits** checkbox to uncheck it. Change **Minimum** to "0" and **Response to value outside limits** to **Coerce**. With this setting, the minimum value of forward voltage is now 0 and if any value lower than 0 is inputted, it will automatically become 0. For convenience, make the value of Increment "0.1." Press the OK button to close the Properties window and check the control behavior. Notice that value will never become lower than 0 and when you click the up or down arrows on the left of the control, it will change by 0.1 (**Figure 3-35**). This is often forgotten since it is not directly shown on the block diagram, but it is very important to "**set the value allowed to be inputted**." Making the program easier to use will raise the reputation of your program. By limiting the value that users can input, the program will not

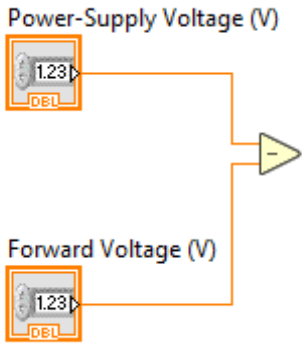


Figure 3-32 Finished wiring to Subtract function

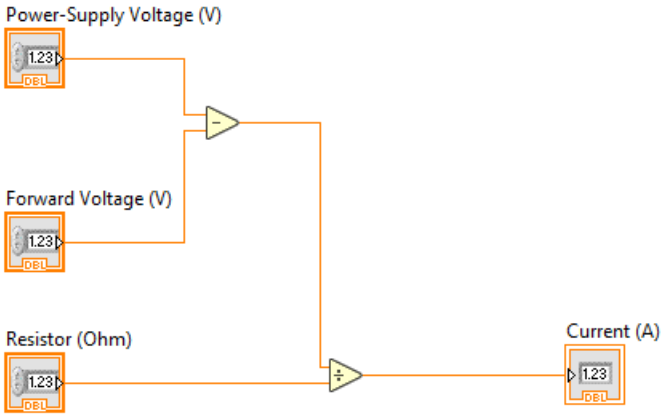


Figure 3-33 Block diagram completed

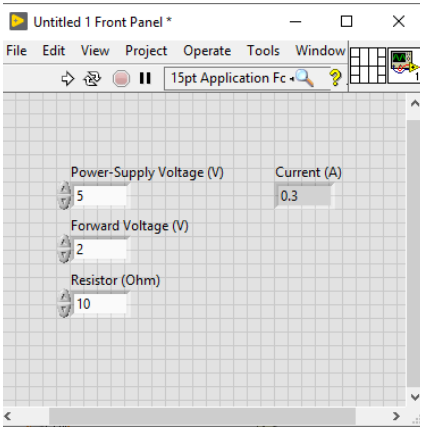


Figure 3-34 Front panel completed

process incorrect data. It is very important to think about the **usability**, how easy to operate the program. Now let's save the VI. Go to the top menu bar and select "**File > Save**" to save the VI. You can also save by pressing **CTRL** and **S** keys. Name the VI as you like.

Let's make this VI evolve a bit more. This time, let's modify it so that we are notified when current becomes greater than the value specified and LED is damaged. Add a numeric control and name it "**Continuous Forward Current (mA)**." Similarly, from the **Boolean** palette (**Figure 3-36**) add **Round LED** to the front panel. Name it "**LED Broken?**" (**Figure 3-37**). We will compare the calculated value of current and the value of "**Continuous Forward Current (mA)**" control. From **Numeric** palette in function palette, select **Multiply** function, and from **Comparison** palette (**Figure 3-38**), select **Greater Or Equal?** function and place them on the block diagram. Wire them as in **Figure 3-39**. Do you know why we are multiplying current by 1000? The

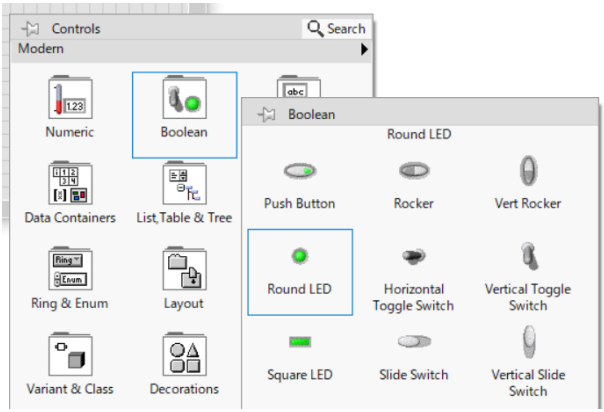


Figure 3-36 LED Indicator

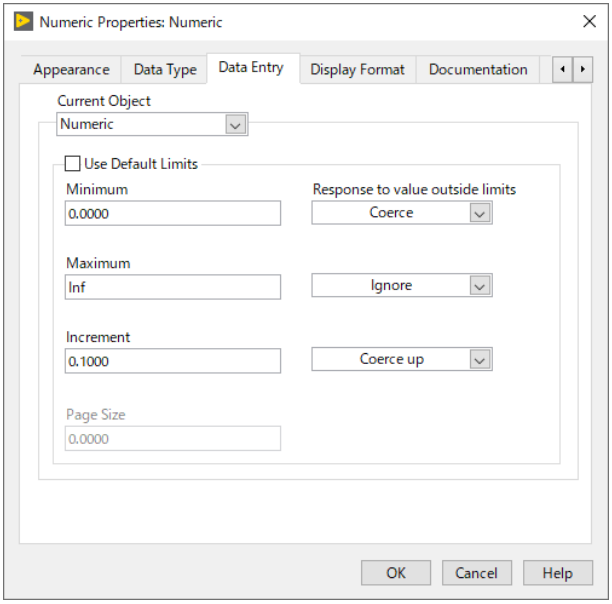


Figure 3-35 Properties for Numeric Control

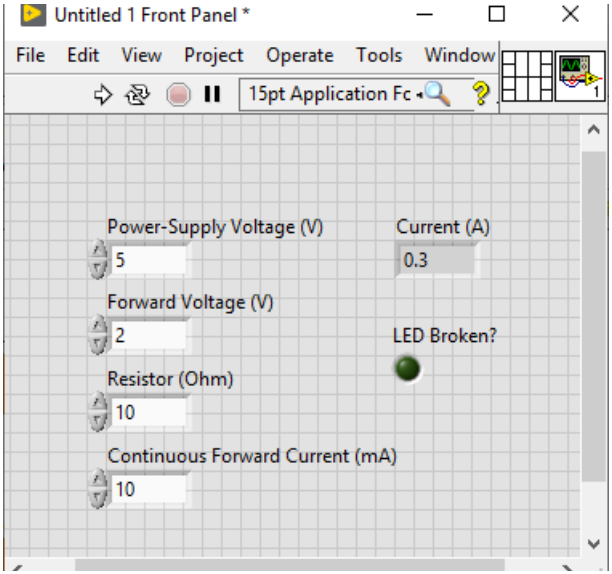


Figure 3-37 LED indicator placed

calculation result should be in amperes but the input to the maximum current control is in milliamperes, so we include this to align the units. You should take a note on the color of "**LED Broken?**" control and the wire connected to it. In programming, knowing what type of data we handle is important (**Figure 3-40**). The orange and blue terminals and wires that we have been seeing deal with numbers. Orange can represent decimal data such as "1.2345." Blue only represents integers. Green is data called **Boolean**, and it only handles two values, F (False) or T (True). Pink terminals and wires handle letters or **strings** such as "Hello, World." In LabVIEW, each datatype is represented by colors, so we can distinguish them by memorizing the corresponding data and color. Run the VI and observe its behavior. If current exceeding the maximum current flows, "**LED Broken?**" indicator turns on and we can understand that it is damaged (**Figure 3-41**). In reality, if LED is damaged, the current no longer flows so the current indicator should become zero. In LED Simulator.vi these details are also implemented so check it out.

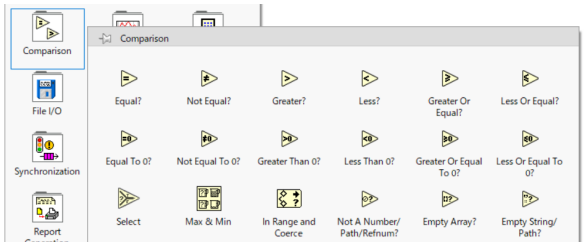


Figure 3-38 Comparison Palette

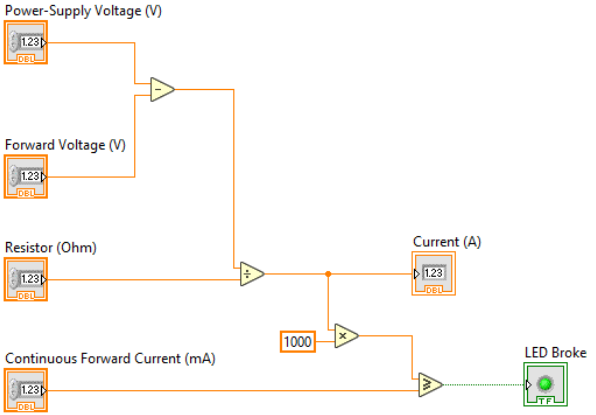


Figure 3-39 Completed block diagram

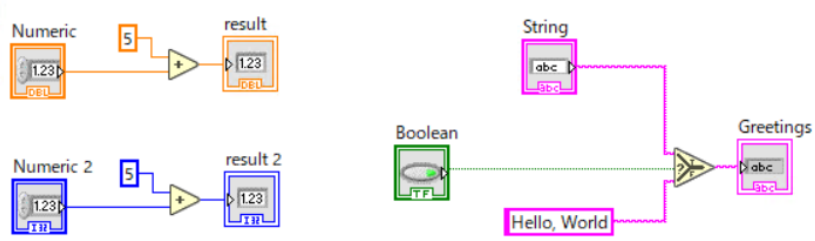


Figure 3-40 Examples of data types

3.4 Making the Program Run Repeatedly

Let's customize the program further. Currently, if we

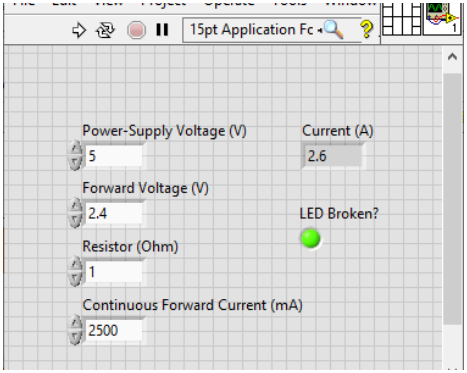


Figure 3-41 Result for LED damaged

change control values to observe the result, we needed to run the VI every time. This is cumbersome. Let's modify the VI so that the VI is always running and changing a value changes the result on the fly. In the world of programming, we call the act of repeatedly running a program **looping**. When looping, the program repeats a specific task until it is ordered to stop looping. In LabVIEW, loop is contained in the **Structures** palette (**Figure 3-42**). Now, let's make the calculation task we made so far run repeatedly. Select **While Loop** from the palette. The mouse cursor changes as in **Figure 3-43**. With this icon as the cursor, surround the code we want to run repeatedly. Move the mouse from top left to bottom right while holding onto the click (**Figure 3-44**). Let go of the mouse once everything is surrounded and a While Loop will be created (**Figure 3-45**). Now the code surrounded by the loop will be run repeatedly until ordered to stop.

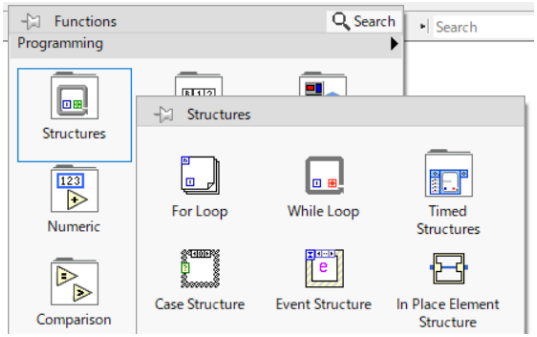


Figure 3-42 Structures Palette

Figure 3-43 Cursor after selecting a structure

Let's run the VI!

Wait, you can see that the run button at top left is not the usual white arrow (**Figure 3-46**). In LabVIEW, when there is a

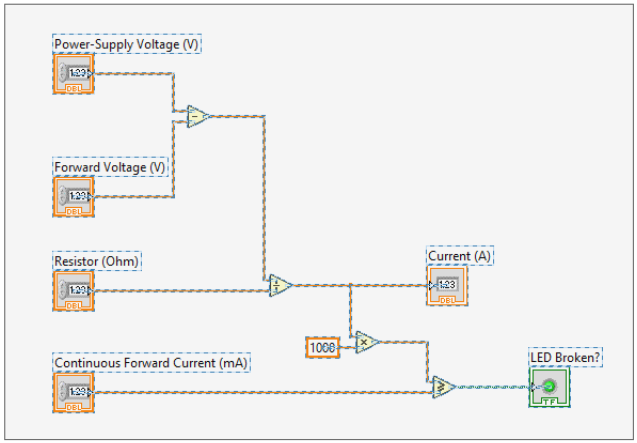


Figure 3-44 Loop in creation

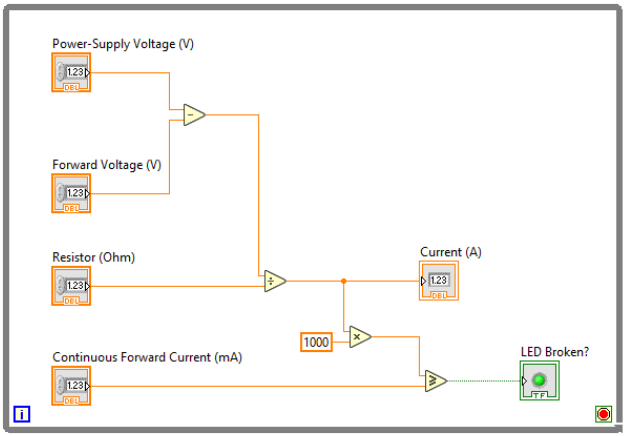


Figure 3-45 Loop created

problem with the program and cannot be run, the run button becomes broken. What could be the cause of this issue? Rest assured. In LabVIEW, there is a mechanism that tells you where the problem is when the run button is broken. Let's try to run the VI even though the arrow is broken. When you click on the run button, a list of problems will be displayed (**Figure 3-47**). In this case, it shows that "While Loop: Conditional terminal is not wired," so the root cause is that there is nothing connected to the so-called conditional terminal. Let us fix this right away. You can press the Show Error button in the Error List to jump to the cause of the issue. Conditional terminal is the red icon located at the bottom right of the While Loop. Right-click on this icon and select **Create Control** (**Figure 3-48**). Now you can see that the run button is fixed. In While Loop, in order to stop the repeated task, we must pass a Boolean data such as a button to this terminal. By following the step instructed previously, LabVIEW automatically creates a button Boolean control with "**STOP**" written on it. Run the VI to make sure that the change of value is applied immediately and that pressing the Stop button stops the VI. We have now created a repeated process, but there is one more thing to do to run repeated task smoothly. When we use the While Loop to repeat a process, PC will process it at its maximum speed possible. This could possibly cause some lag in the mouse cursor movement or in the worst case scenario, cause other software to stop. To avoid this from occurring, in programming we create a wait time between processes to allow the PC to perform other tasks during that time (**Figure 3-49**).

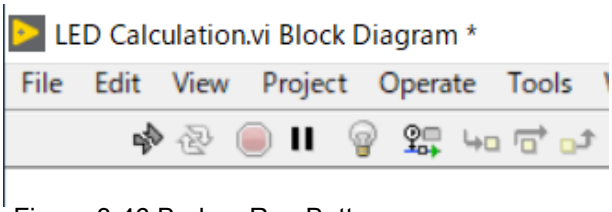


Figure 3-46 Broken Run Button

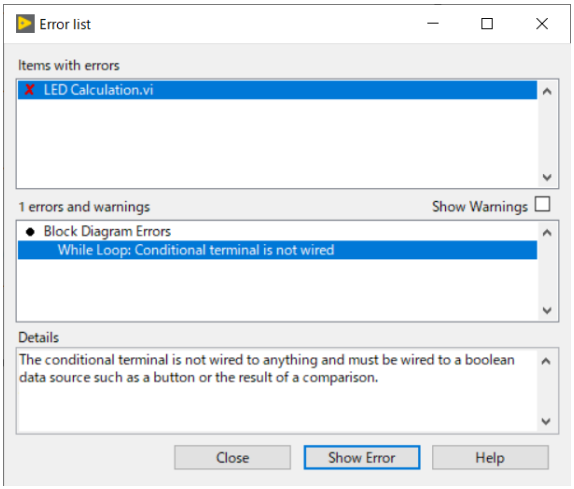


Figure 3-47 Error List

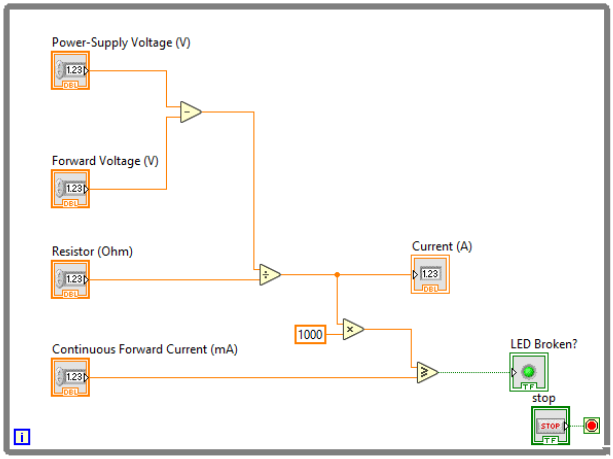


Figure 3-48 Block diagram so far

Place **Wait Until Next ms Multiple** function in the **Timing** palette in the While Loop. This function inserts a wait time between processes and lighten the workload for a PC. We can also use this function when we want to measure something in xx second intervals. We will connect a constant to it. A constant is a value that will not change when a program runs. Values defined in society such as Pi or the speed of light is also an example of a constant. To create a constant, place the cursor to the left of the function, and when the cursor changes to the bobbin icon, right-click and select **Create Constant**. The unit is **milliseconds** so if we want to process every 1 second, we will input "1000" (**Figure 3-50**). Now we have finished creating a program that performs a calculation every second. Next let's change the power voltage to be selected from a set of predefined values. Currently, any value can be used for power voltage, but in reality, either of 5/3.3/1.8V are often used. We will modify the VI so that users can select from these three values. Delete "**Power-Supply Voltage(V)**" control. Click on the control or the terminal on the front panel or block diagram respectively, and use **Backspace** or **Delete** key to delete. When the terminal is deleted, an X is marked on the wire connected to Subtract function. This means the wire is broken, and this must be fixed to run the VI. We can either delete the broken wire or use **CTRL** and **B** keys to mass delete all broken wires. Place an **Enum** control in the **Ring&Enum** palette (**Figure 3-51**) on the front panel and rename it as "**Power-Supply Voltage(V)**." Enum is **enumerated constant** and it is a dataset of a string and an integer pair. You will see when you create

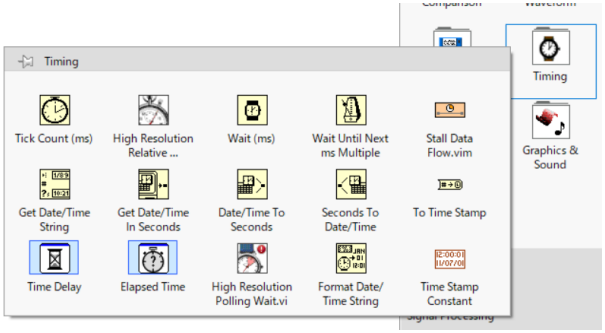


Figure 3-49 Timing Palette

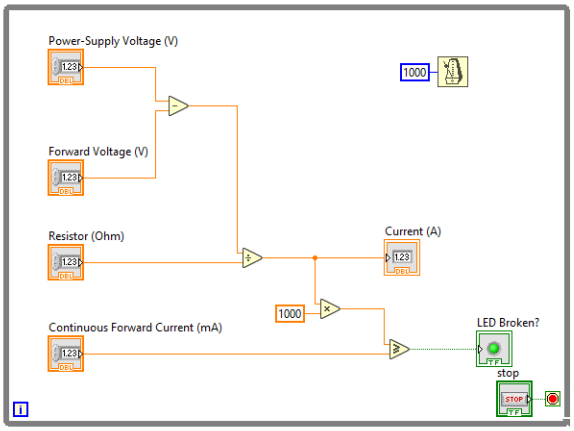


Figure 3-50 Wait until Next ms Multiple function

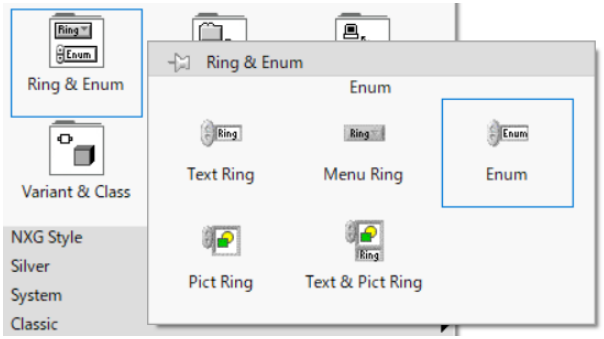


Figure 3-51 Ring & Enum Palette

one. Right-click on the Enum control and display Properties. Select **Edit Items** tab in the Properties. We will define the Enum data here. Double-click on the white space under **Items** column next to "0" and enter "5V." Press the Enter key to move onto the next row and add 3.3V and 1.8V. Each value corresponds to an integer, as in 5V is to 0, 3.3V is to 1, and 1.8V is to 2. This means that when user selects "5V," the value "0" will be used in the program. Once you complete this, click the OK button (**Figure 3-52**). Now we must make the value of the power voltage equal to the selected voltage level. In Enum, even if 5V was selected, the actual value outputted will be "0," so this will yield calculation such as "0V – Forward Voltage(V)." We will modify the VI and use a Case Structure here to make the program use the same voltage level as selected in the program. A Case Structure is used to process different cases based on condition. We change the process with way such as if "oo" then "xx" or if "xx" then "oo", i.e. If button is pressed, process A will be conducted, and if not pressed, process B will be conducted. For this VI, we will change the value we pass to **Subtract** function based on which voltage is selected. **Case Structure** is located in the **Structures** palette as a Case Structure. We use it the same way as a While Loop; select it in palette and drag and drop from top

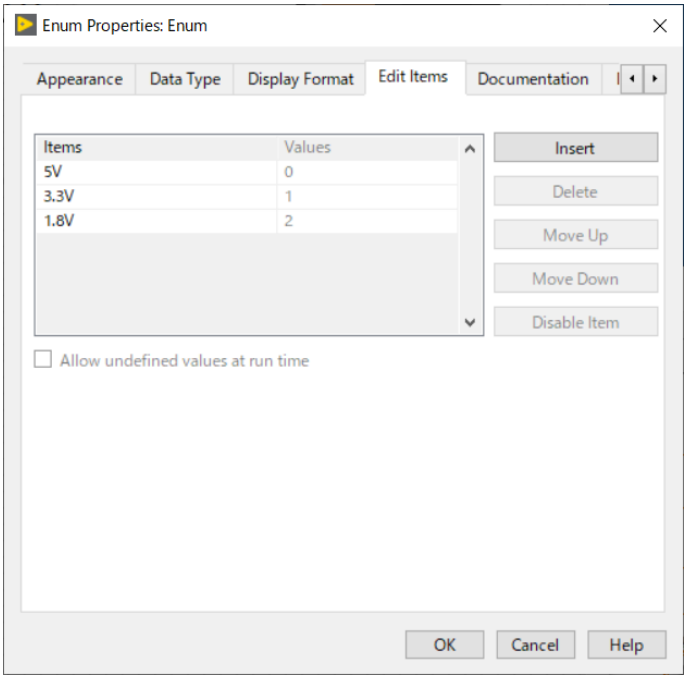


Figure 3-52 Editing an Enum

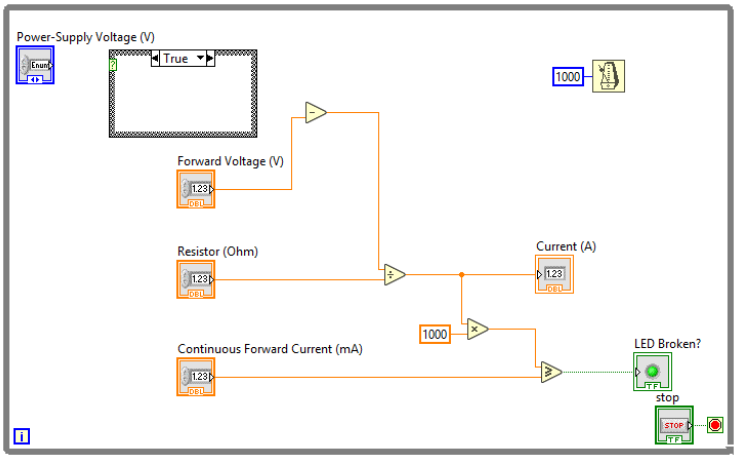


Figure 3-53 Creating a Case Structure

left to bottom right where you want to place it. For this example, create it as in **Figure 3-53**.

On the left side of the Case Structure, there is a "?" terminal, and the data passed to this terminal determines the case execution condition. When Case Structure is first placed, it thinks that a Boolean data will be inputted, so there are two cases, True or False preset. By selecting the ◀▶ on the top, you can change the page. Now, wire the Enum we created to this "?" terminal. Notice that the page was automatically renamed. Press the ▼ mark, and you will see two cases, 5V and 3.3V displayed. Notice that there is no page for 1.8V (**Figure 3-54**). Right-click on the top of the Case Structure where the page name is displayed, and select **Add Case for Every Value**. Now all the power voltage patterns are displayed (**Figure 3-55**). Select the page with ◀▶ or ▼ marks to move to the page you want to display. By creating a program inside every page, we can determine the process to be executed for each selection of the power voltage value. Display the 5V page, and place a DBL Numeric Constant in the Numeric palette. Input "5" in this numeric constant. Connect the numeric constant and the Subtract function with a wire (**Figure 3-56**). There is a Case Structure edge in between, but feel free to connect them as usual as LabVIEW will process it. Once wiring is complete, you will see a white square created on the edge of the Case Structure. This is called a **tunnel**, and it is automatically created when we let out a wire from inside of structures such as a Case

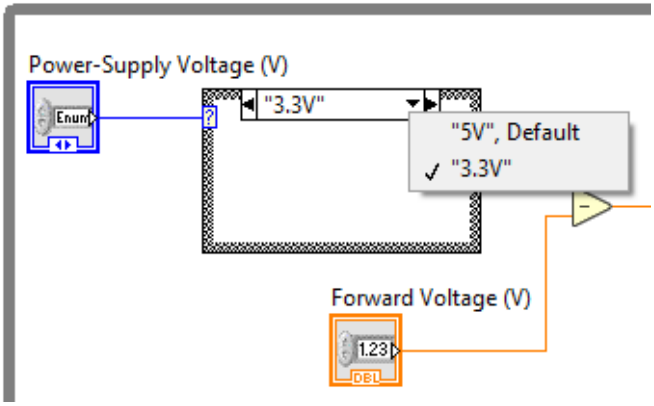


Figure 3-54 Not enough cases are created

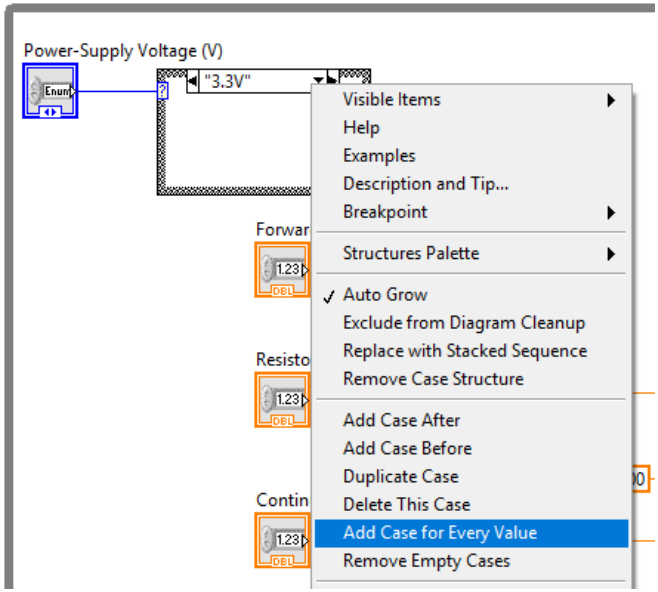


Figure 3-55 Add Case for Every Value

Structure or a While Loop. Previously, we directly wired from the numeric constant to Subtract function, but a tunnel is also automatically created if we click on the edge of the Case Structure in the middle of creating a wire. The reason why the inside of the square is white is because this tunnel has not been defined a value to output in other pages. We have not yet defined a value for 3.3V and 1.8V pages. Let us place a numeric constant on each page and create 3.3V and 1.8V values respectively (**Figure 3-57**). When the value is defined for all pages, the tunnel will be filled (**Figure 3-58**). Lastly, define the default value of the front panel objects. Save the VI and close it. When we reopen the VI, notice that values you assigned on the controls are reset. In LabVIEW, there is a setting known as default value, and we can predefine the value of the controls and indicators at the time VI is first opened. For instance, set "**Forward Voltage(V)**" as 1.2V, "**Current Limiting Resistor (Ohm)**" as 300Ω, and "**Continuous Forward Current**" as 30mA and select from LabVIEW menu bar "**Edit > Make Current Values Default**." Save the VI and reopen it, and you will see that the previous setting has been preserved.

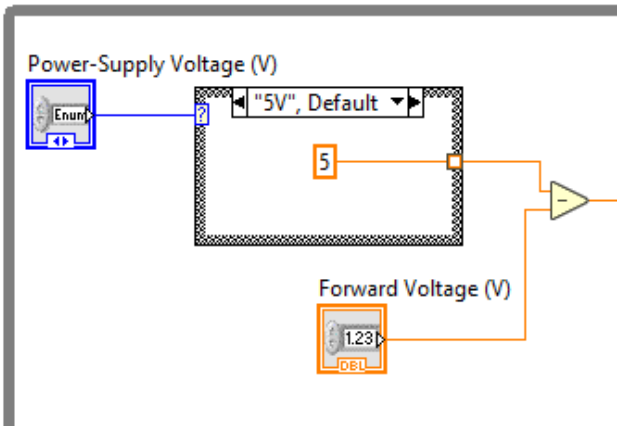


Figure 3-56 5V case wiring complete

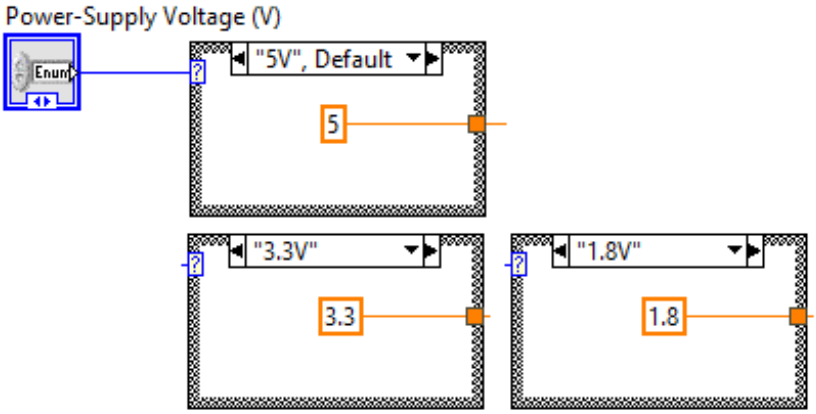


Figure 3-57 Processing for each case

This concludes the exercises for creating a VI, but feel free to work on the advanced topics below.

- Since current does not flow when LED is broken, we want to display "0" on current indicator. Use the **Select** function in **Comparison** palette instead of a case structure to implement this.
- A broken LED is a serious condition. How can we notify the user that it is broken with a more obvious visual effect other than "**LED Broken?**" indicator? Use **One Btn Dialog** function in **Dialog & User Interface** palette and a case structure to display a popup message box when the LED is broken.

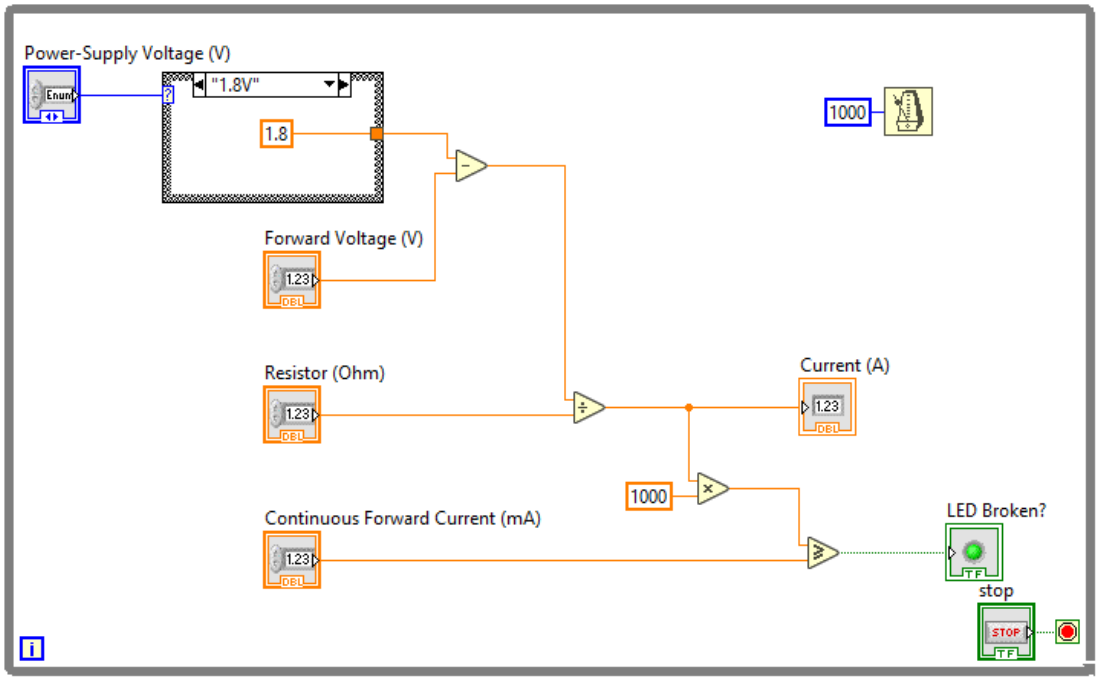


Figure 3-58 Finished VI

Article 3: Programming with LabVIEW NXG

In LabVIEW NXG, there is no change in basics such as Control and Indicator, functions and wires. The biggest difference is that in LabVIEW, front panel and block diagram were separate windows but in LabVIEW NXG, they are combined in a single window. Their names have been changed to Panel (**Figure C3-1**) and Diagram (**Figure C3-1**) respectively with new and improved user interface. LabVIEW NXG allows you to switch languages within their UI so you could program in the language of your choice. You are also able to create a website with LabVIEW NXG, so the future of programming may lie here in NXG.

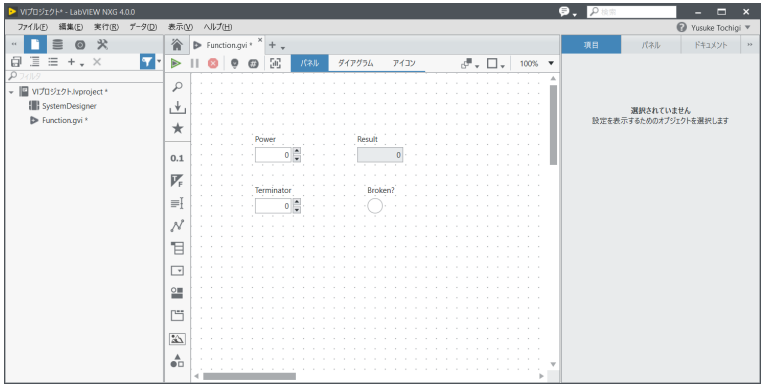


Figure C3-1 LabVIEW NXG Panel Window

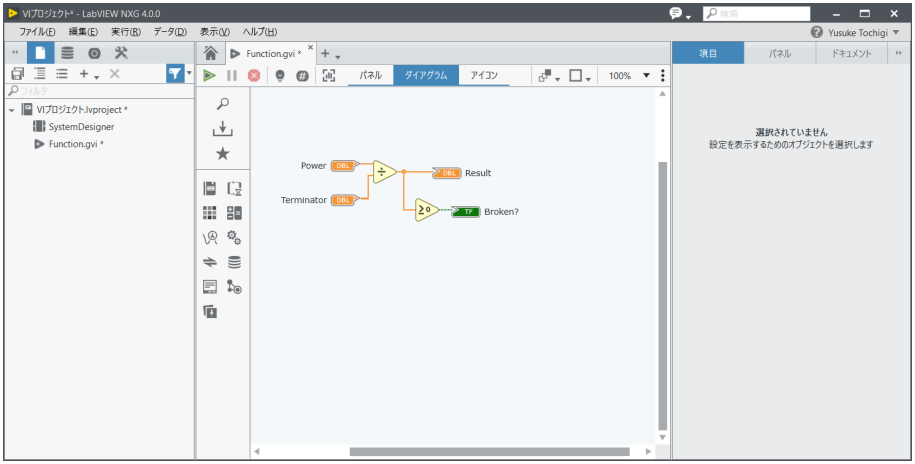


Figure C3-2 LabVIEW NXG Diagram Window

Chapter 4

Making Your Own Application

Get started making applications that you can enjoy and use in your hobby.
In this chapter, we will introduce the application of spectrogram that uses the recording and playback function of PC, and show you how to create your application.

[Keywords] Spectrogram, Sound Control panel, Stereo Mixer, State Machine, State Transition Diagram, LabVIEW Example VI, Finite Sound Input.vi, Graphics & Sound Palette, Flat Sequence Structure, Wait (ms) Function, Array, Waveform Data Type, Reverse 1D Array Function, For Loop, Shift Register, 1D Array, Index, 2D Array, Intensity Graph, Type Definition, Measurement File

4.1 LabVIEW Programming for Your Hobby

LabVIEW was born over 30 years ago for engineers and scientists to use at work. Suddenly, if you were told to use LabVIEW as a hobby rather than at work, you wouldn't immediately know what to use it for.

Home automation and control of railway models are extensions of LabVIEW used in the workplace, so you will be able to program immediately. For other hobbies, using the LabVIEW graphical programming language, you can do things you didn't realize before and create something more amazing than you might imagine.

With the power of measurement and analysis that LabVIEW has improved over the years, we have come up with a program that will attract a lot of people. The **SoundVIEW** (**Figure 4-1**) created for this eBook can be enjoyed without purchasing any other device if there is a PC and LabVIEW. It uses a method called a spectrogram that expresses changes in the frequency components of a sound with time. This graph has time on the horizontal axis and frequency on the vertical axis, and the intensity of the frequency component of the sound is represented by the gradation from yellow to red and black.

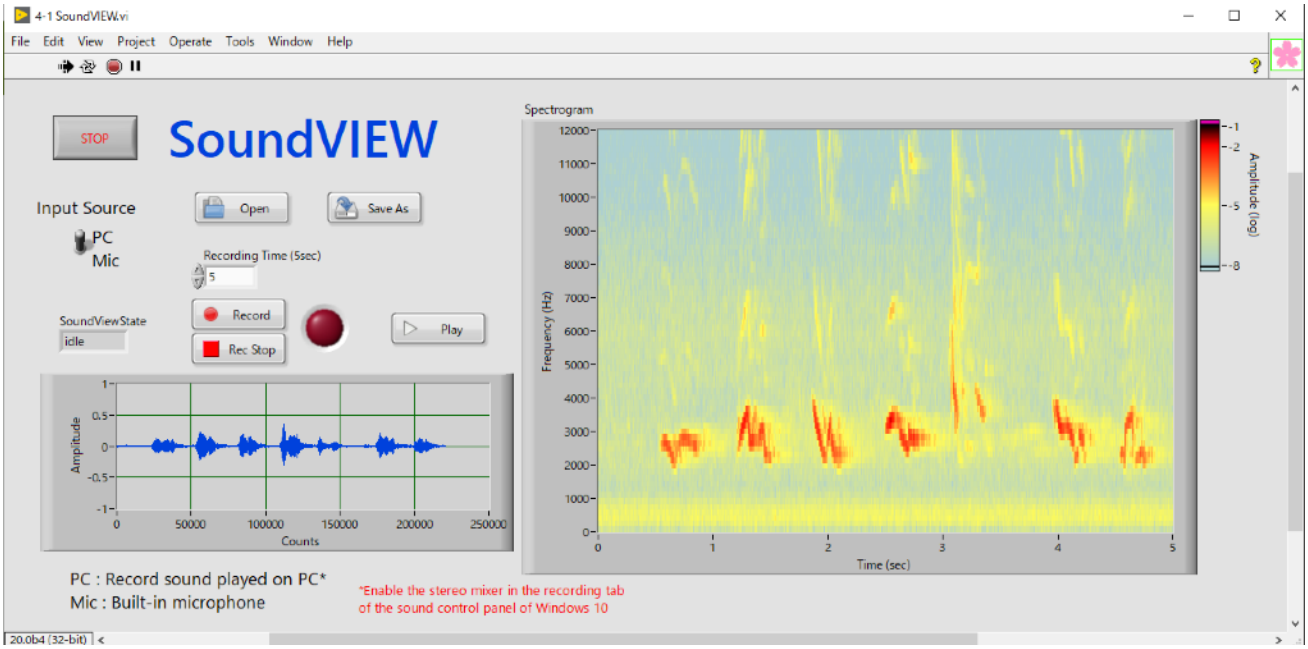


Figure 4-1 Front panel of SoundVIEW.vi

If you like hiking, you may hear birds chirping while walking along the mountain trails. Wouldn't it be great to know the name only from the birds' voices? People who like wild birds have recorded the song of birds and posted it on the website "Bird-sounds.net" (**Figure 4-2**). **Figure 4-1** shows the frequency analysis using SoundVIEW while listening to the birdsong of "American Robin" on a PC. In this way, you can see the sound of many wild birds using SoundVIEW.

For those interested in speeches, for example, you can watch the spectrogram in SoundVIEW while listening to a speech by high school environmental activist Greta Thunberg at the UN Climate Change Summit on YouTube. Song lovers can see Lady Gaga's song in the spectrogram, jazz lovers can see Miles Davis's trumpet sound, and poetry lovers can see Peter Dixon's recitation Robert Frost's poem. You will surely find something fun using SoundVIEW (**Figure 4-3**).

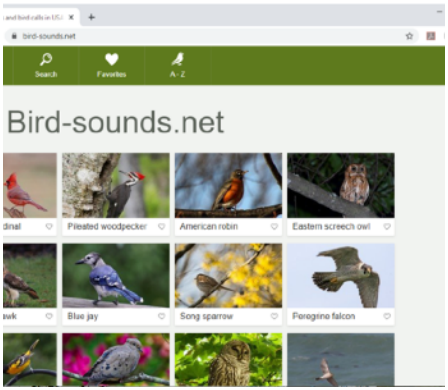


Figure 4-2 Bird-sounds.net

This chapter describes how to create an application in LabVIEW.

First, let's operate SoundVIEW. This program is designed to operate according to the button operation. After explaining the operation, we briefly explain the SoundVIEW block diagram. This program was made with an architecture called a **state machine**. In this chapter we will create a record and playback program step by step using this architecture.

4.2 Operate the SoundVIEW Program

Open **SoundVIEW.vi** from the program folder in Chapter 4 (**Figure 4-4**). SoundVIEW has a recording function, and you can select **PC** or **Mic** as the sound source. If you select **PC**, you can record the audio output from the web browser or music player with the function of **Stereo Mix** (**Figure 4-5**) of Windows PC.

Depending on your computer, **Stereo Mix** may not be enabled, or **Stereo Mix** may not be displayed in the recording tab of the **Sound** control panel. In that case, search for "Windows 10" "Stereo Mix" as a keyword to find a solution. **Stereo**

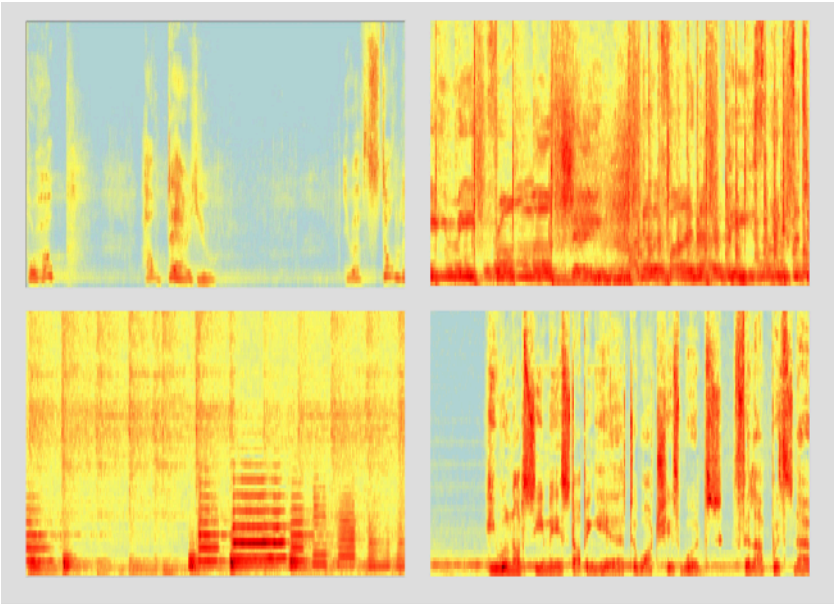


Figure 4-3 Samples of spectrogram

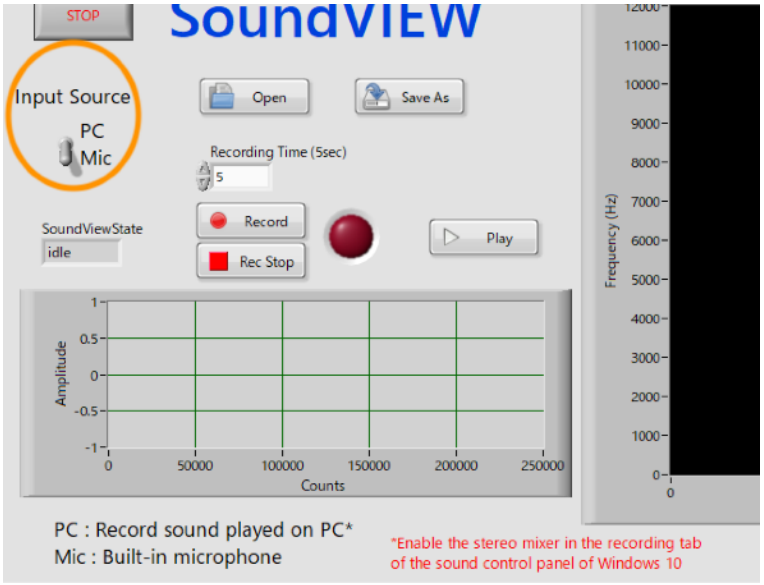


Figure 4-4 Select input source

Mix may appear as **Playback Redirect** or **What U Hear**.

In LabVIEW, sound devices are identified by an ID number. The **default device** has ID = 0. If there is a **default communication device**, it will be "ID = 1." The remaining devices will be numbered incrementally.

Three devices are shown in Figure 4-5, but the external microphone is disabled because it is not connected. The microphone is "0" and the **Stereo Mix** is "1." If an external microphone is connected and enabled, the external microphone is "0", the microphone is "1", and the **Stereo Mix** is "2".

As you can see, the device ID changes depending on the PC state, so we must change **recStart** subdiagram (Figure 4-6) to the appropriate ID number (Figure 4-7).

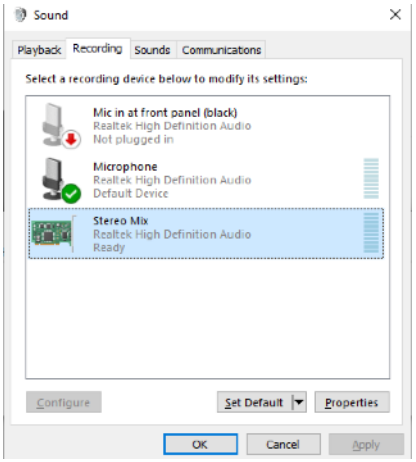


Figure 4-5 Sound control panel

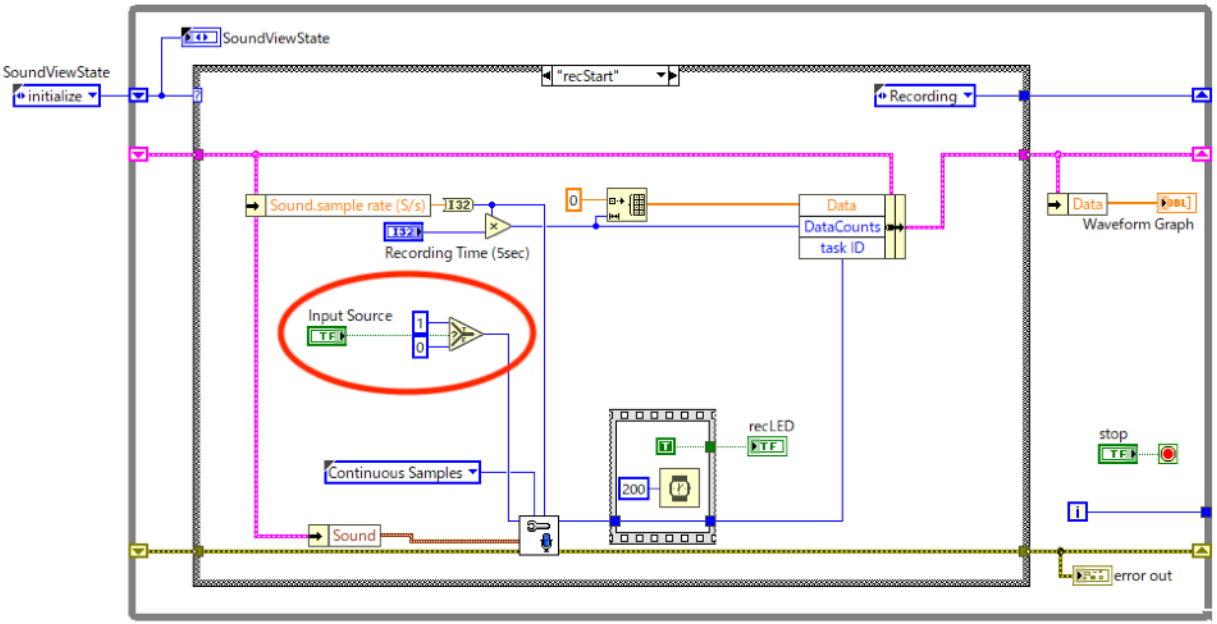


Figure 4-6 recStart subdiagram

If you open and execute **AvailableDevice.vi** in the program folder of Chapter 4, the sound source recognized by LabVIEW is displayed (**Figure 4-8**). The upper block is the input device and the lower block is the output device. In this figure, the input devices used for the recording function are a "microphone" (ID = 0) and a stereo mixer (ID = 1). If you change the settings of **Sound** control panel of Windows 10 with **AvailableDevice.vi** open, close **AvailableDevice.vi** and then open it again.

Press **Record** button to start recording. The default recording time is 5 seconds, but if you have a powerful PC, changing it to a higher value may work. Press **Rec Stop** button to stop recording. The waveform being recorded is displayed in the graph below **Rec Stop**

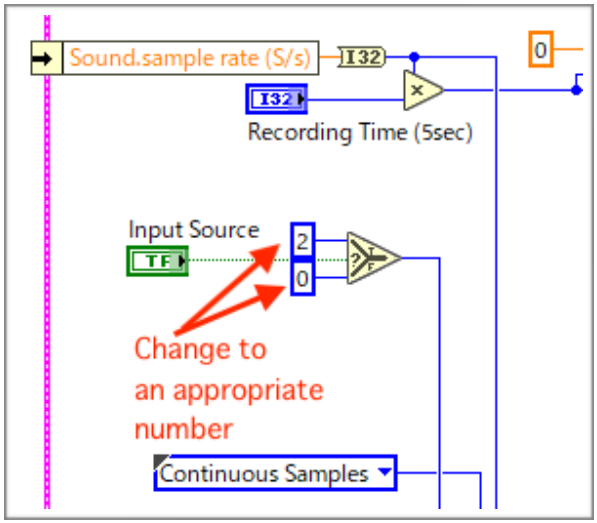


Figure 4-7 ID number of sound source

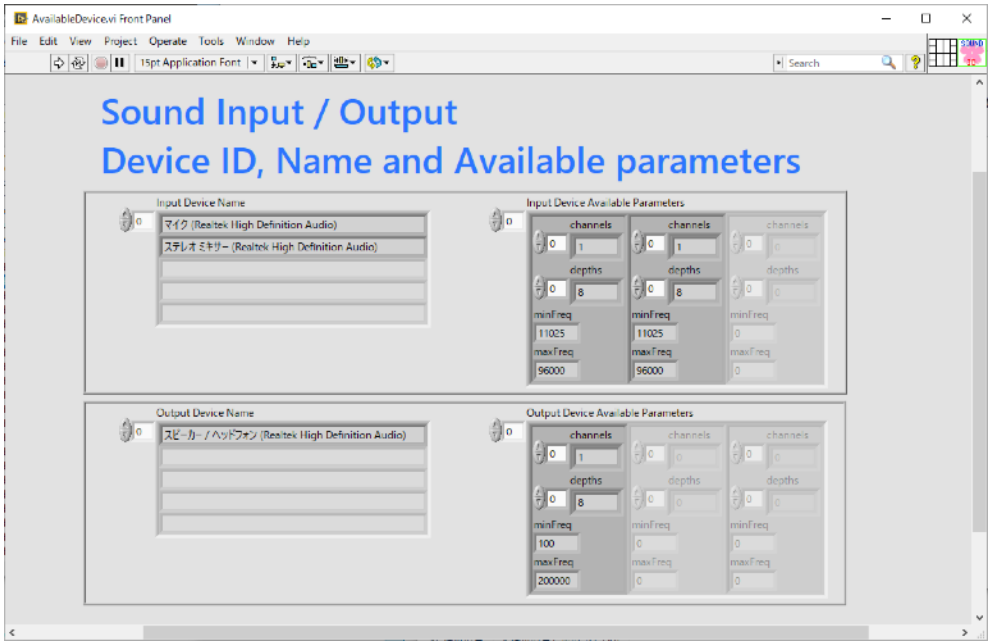


Figure 4-8 Confirmation of sound source using AvailableDevice.vi

button. Press **Play** button to play the recorded sound and display the spectrogram. When you click **Save As** button, the file dialog is displayed. You can enter the file name and save it. You can load the recorded file with **Open** button. Press **STOP** button to stop the program. The current status is displayed in **SoundViewState**.

Then, operate each button while checking the operation and reaction.

Looking at the block diagram, there is a case structure inside a large while loop (**Figure 4-9**). The case structure has eight subdiagrams. The value of **Enum "SoundViewState"** connected to the case selector changes according to the user's operation or program flow, and the subdiagram is executed selectively. Such a program structure (architecture) is called a **state machine**.

Figure 4-10 shows a state diagram that outlines the operation of the program.

When **Run** button is pressed, the VI executes **Initialize** and waits for input in **Idle**. **Idle** waits for **Record**, **Play**, **Open**, and **Save As** buttons.

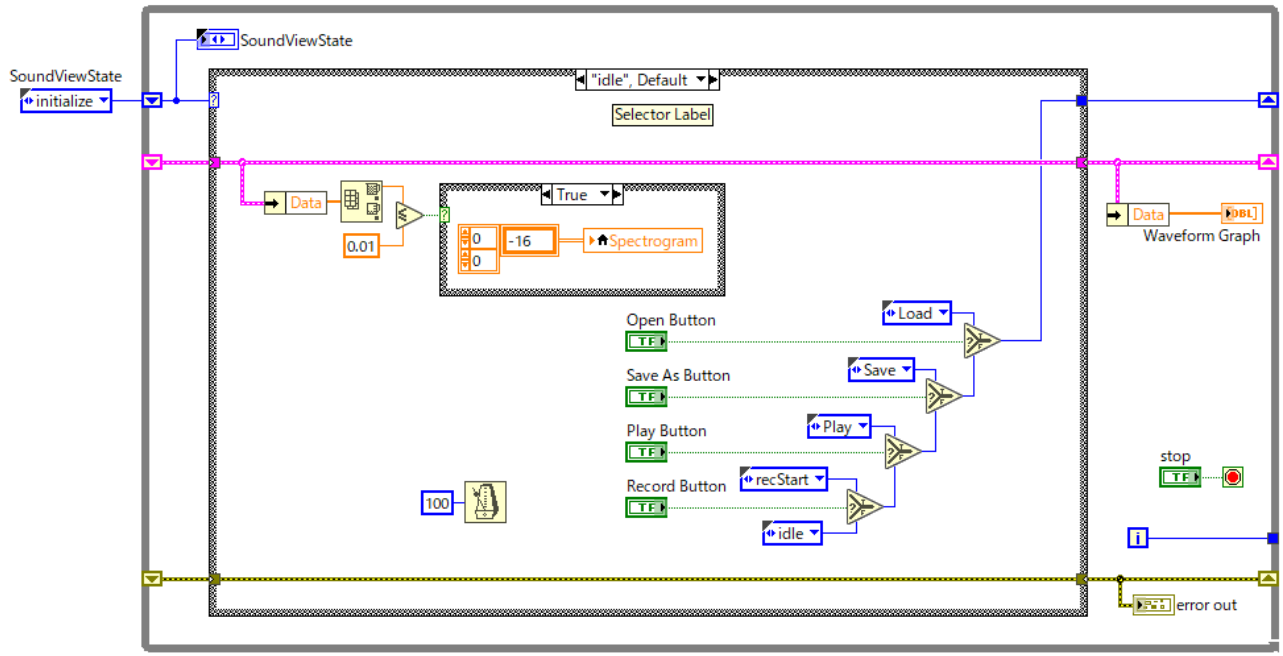


Figure 4-9 Block diagram of SoundVIEW

When **Record** button is pressed, **Start Recording** (recStart) process is executed and **Write to Ring Buffer** (Recording) subdiagram continues until **Rec Stop** button is pressed.

When **Rec Stop** button is pressed, **Stop Recording** (recStop) is performed and **Idle** (idle) is performed.

Play, save and load are handled by pressing the corresponding button and return to **Idle**.

Even complex programs can be created by dividing the desired function into subdiagrams and making state transitions that allow natural operations.

The spectrogram is created by **sub_Spectrogram.vi** in **Play** subdiagram (**Figure 4-11**). Double-click the icon to open the VI and display the block diagram (**Figure 4-12**). **sub_Spectrogram.vi** takes data as the

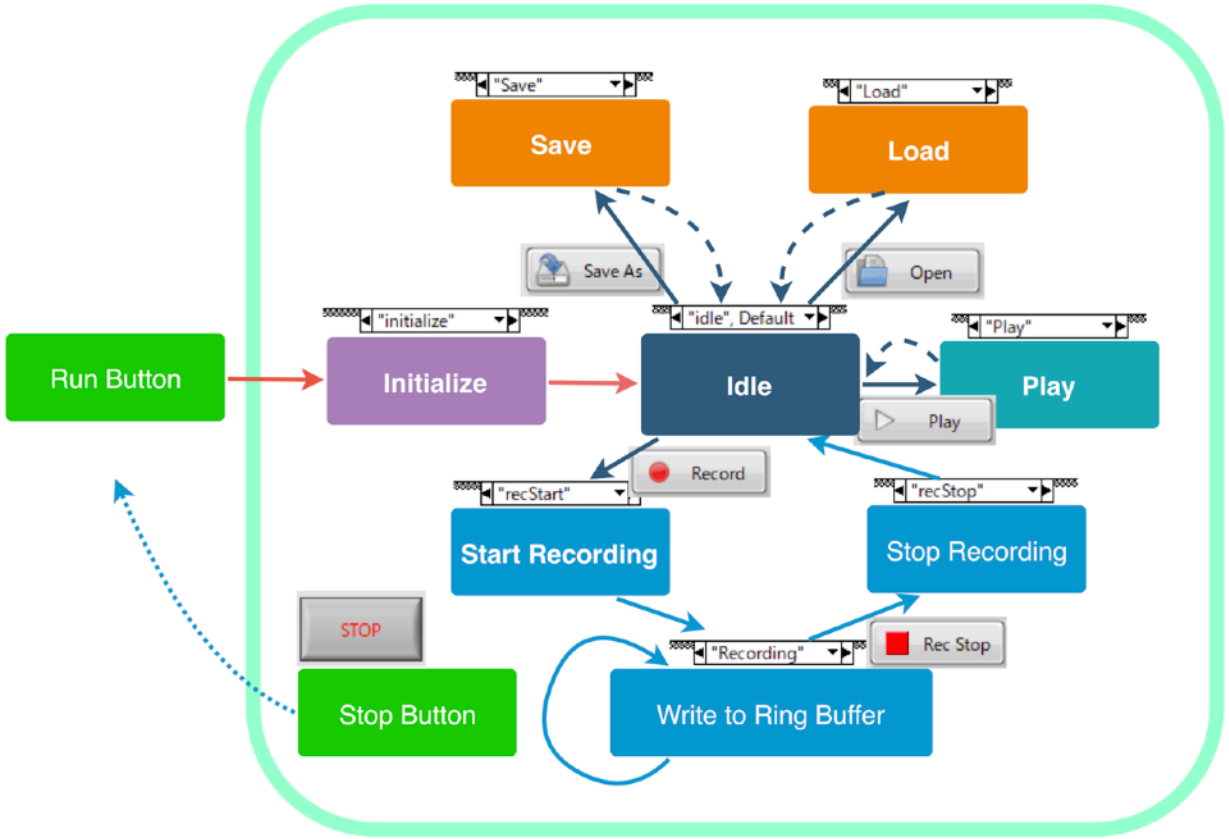


Figure 4-10 State diagram of SoundVIEW

input, processes it using the function in the **Spectral Analysis Palette**, and outputs a two-dimensional array **STFT spectrogram** and some graph scales.

sub_Spectrogram.vi is called **subVI**, and the input, internal processing, and output are defined. Proper use of subVIs can improve block diagram visibility and helps identify program errors. If you don't need to pay attention to the details of the subVI block diagram, you can treat the subVI as a black box where you can connect the inputs and get the appropriate output. When creating a large program, subVI can be executed independently to check the function and improve the completeness.

A subVI corresponds to a **subroutine** in other programming languages, but many programming languages do not allow subroutines to work independently. One of the benefits of LabVIEW is that you can see the behavior of each subVI individually.

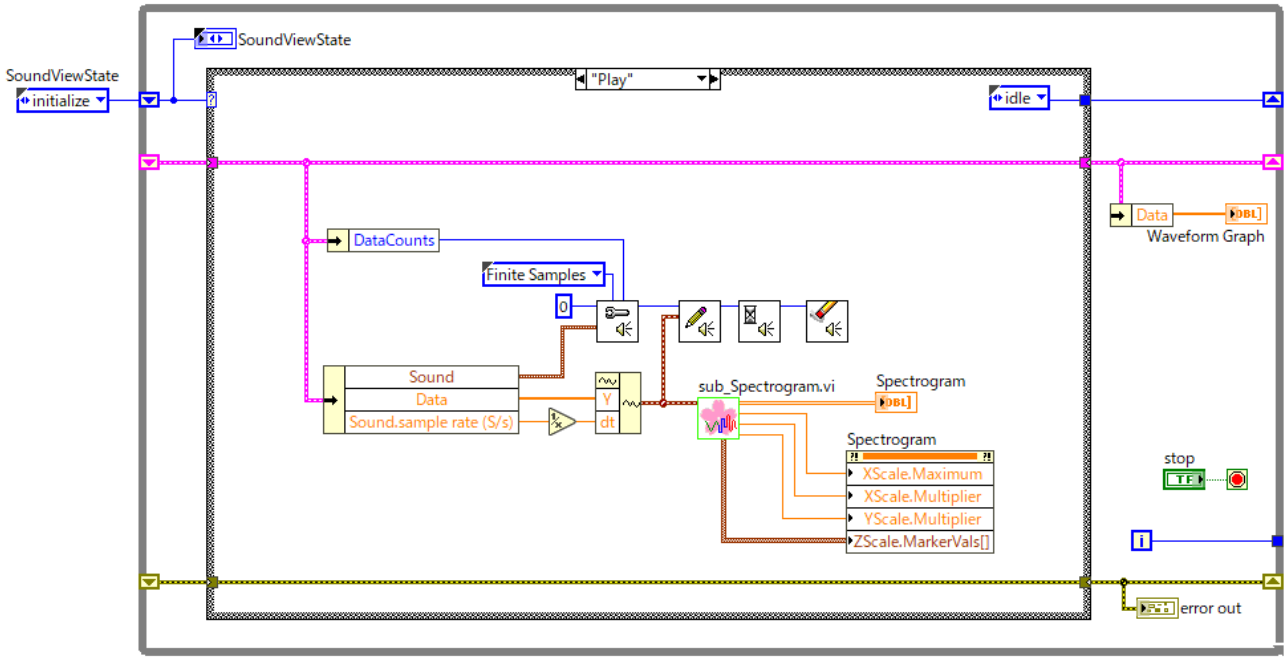


Figure 4-11 “Play” subdiagram

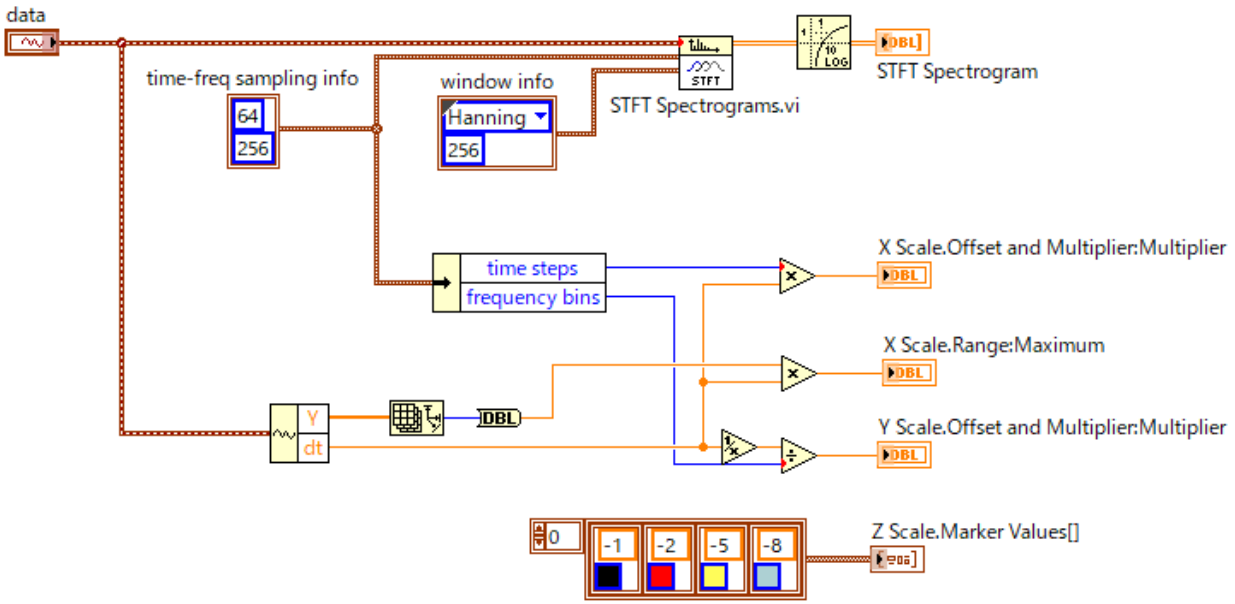


Figure 4-12 Block diagram of sub_Spectrogram.vi

4.3 Record and Playback Example Program

Look at the examples when you start a new project in LabVIEW. It usually gives a good starting point.

Open **NI Example Finder** by selecting **Find Examples ...** from the Help menu, as shown in **Figure 4-13**. Select **Directory Structure** button. Open **Sounds** folder in **Graphics & Sounds** folder and select Finite Sound Input.vi, as shown in **Figure 4-14**.

Then, **Finite Sound Input.vi** will open. Save it to your working folder. Enter "0" in **Device ID** and "50000" in **Samples / ch**, and press the **Run** button while talking to the microphone. The audio data should be displayed on the graph for approximately 2.2 seconds, as shown in **Figure 4-15**. Please note that the data is not recorded for about 0.5 seconds from the beginning. Looking at the block diagram (**Figure 4-16**), you can see that it was recorded with a sampling frequency of 22050Hz, 2 channels, and 16-bit precision. Note that the icons are, from left to right, (1) spanner (input setting) → (2) glasses (input data) → (3) eraser (clear).

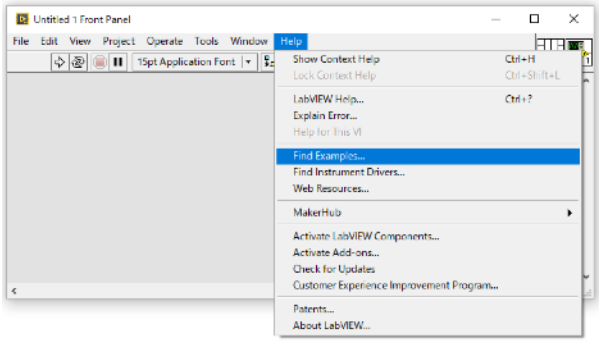


Figure 4-13 "Find Examples..."

Now let's modify this example to create a program

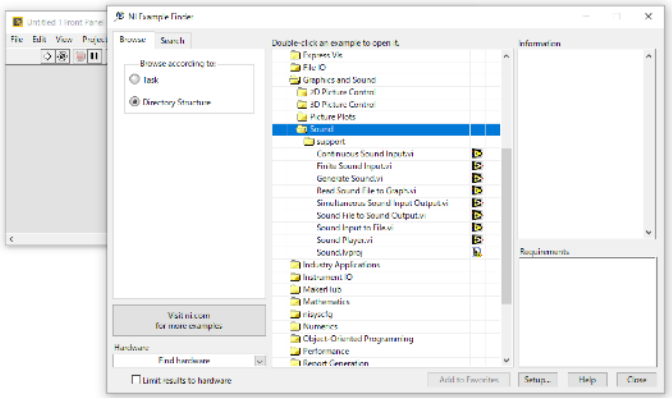


Figure 4-14 Sounds folder of NI Example Finder

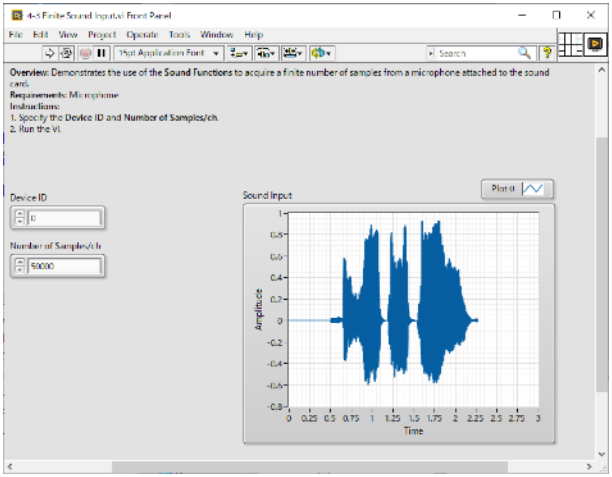


Figure 4-15 Finite Sound Input.vi

that produces sound from the speakers in the "Parrot fashion".

First, save **Finite Sound Input.vi** as **parrot.vi**.

Sound functions are located in **Sound Palette** of **Graphic & Sound Palette** (**Figure 4-17**). Select **Output** to display the audio output function shown in **Figure 4-18**. Drag and drop the spanner icon, pencil icon, hourglass icon, and eraser icon to the block diagram of **parrot.vi**. The hourglass icon **Sound Output Wait.vi** is a function that waits until all data is output. Place and wire the icons as shown in **Figure 4-19**. Sounds

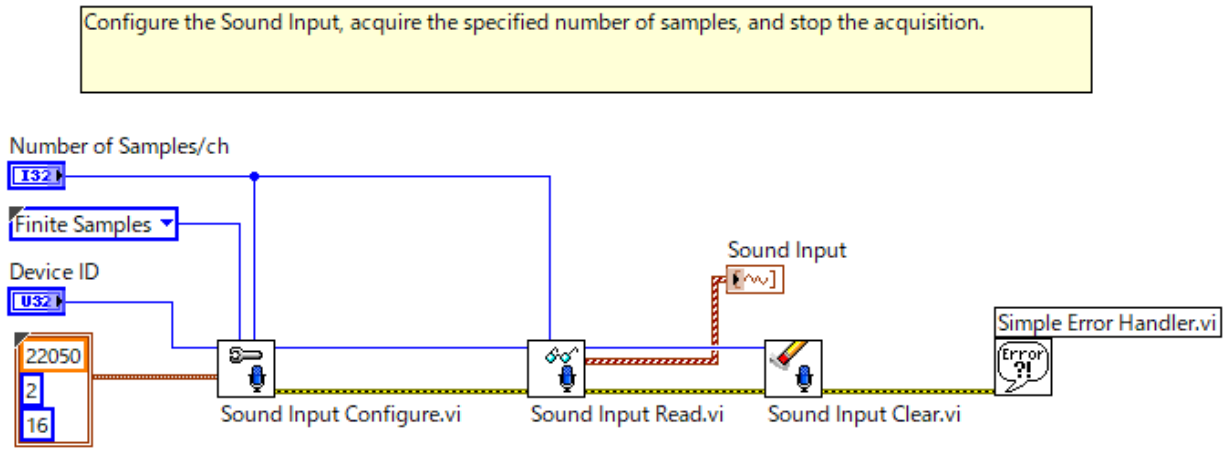


Figure 4-16 Block diagram of Finite Sound Input.vi

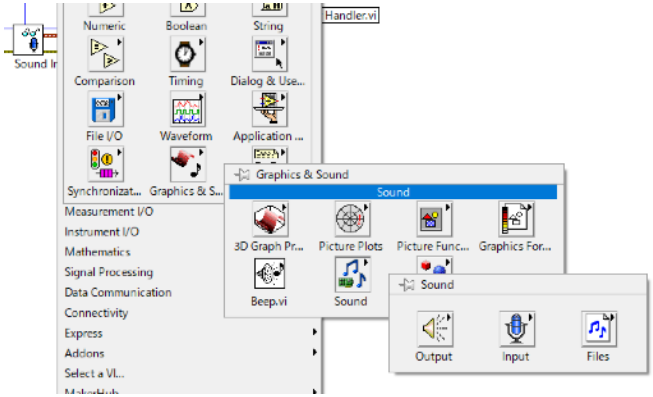


Figure 4-17 ”Graphics & Sound” Palette

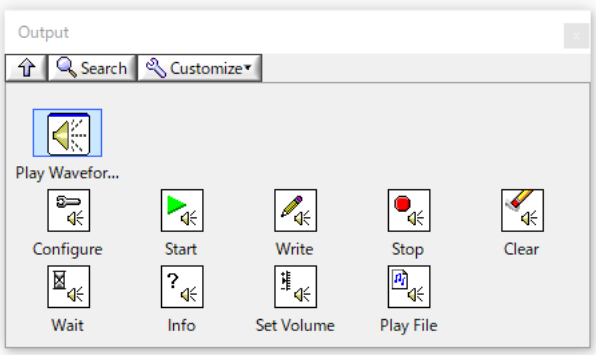


Figure 4-18 ”Output” Palette

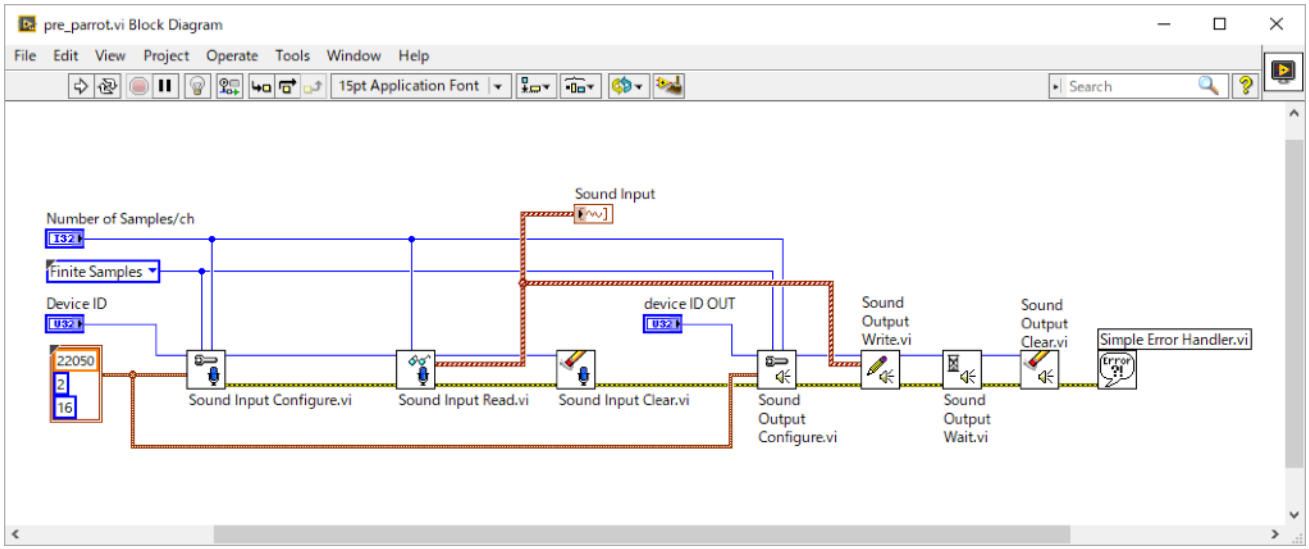


Figure 4-19 Wiring sound output functions

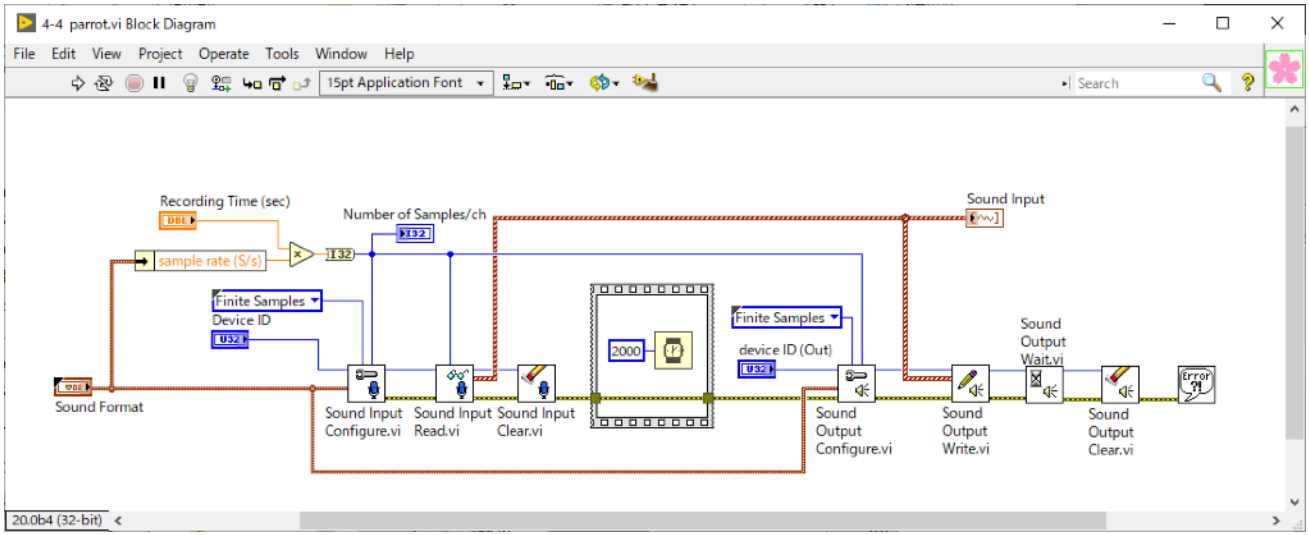


Figure 4-20 Block diagram of "parrot.vi"

recorded by the microphone should be played immediately.

Let's change it to play after a few seconds. Insert a **Wait (ms)** function in the flat sequence structure and set it to wait for two seconds. Now **parrot.vi** plays back the sound two seconds after it is recorded. Specifying the recording time is easier than specifying the number of samples, so let's change that as well (**Figure 4-20**).

4.4 Waveform Data and Array

Save **parrot.vi** as **reverse.vi**. Let's create a program that plays sounds in reverse. The recorded data up to this point is displayed with a thick brown line, and the audio data received by the microphone can be returned to the speaker output by wiring. The recorded data handled here is a collection of audio data for the left and right channels. It is stereo data because the number of channels is set to 2 in the audio format. Use **Indexed Array** function to display the data for each channel. An array is a collection of data of the same type.

Each piece of data in the array is called an element and is specified by its index. Note that the index starts from 0.

The left channel is the element with index 0 and the right channel is the element with index 1. For example, when processing right channel data, use **Index Array** function to extract index 1.

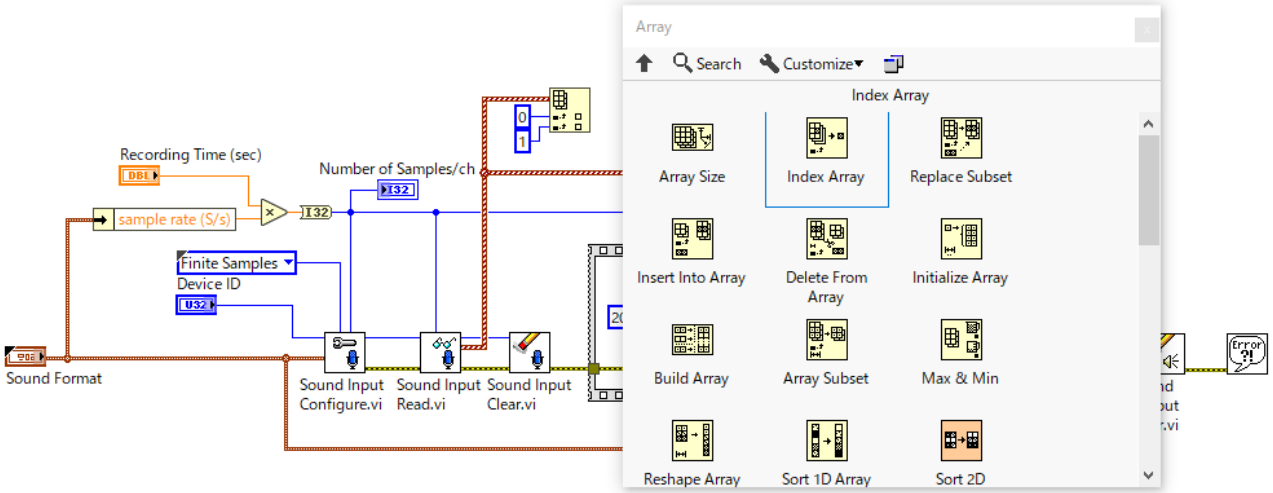


Figure 4-21 Index Array function of Array Palette

Drag and drop **Index Array** function from **Array Palette**. You can add more terminals by left-clicking the bottom of the icon and dragging down. Enter the index of the element to retrieve in the left terminal. Here they are 0 and 1 (**Figure 4-21**).

Using the **Wiring tool**, right-click the terminal to the right of **Index Array** function and select **Create > Indicator** (**Figure 4-22**). A waveform indicator is created. Note that the **Waveform** wire is thinner after being taken out as an element. Looking at the front panel, the waveform indicator consists of **t0**, **dt**, and **Y,** as shown in **Figure 4-23**.

When you execute **reverse.vi**, the data is displayed on the waveform display. **t0** is the start time of the recording, and **dt** is the sampling interval in seconds. The **dt** value displayed is 45 μs, which is the reciprocal of the sample rate 22050 (S / s) specified in the sound format. The recorded data array **Y** is also displayed (**Figure 4-24**).

The number of elements in **Y** is the sample rate multiplied by the recording time (in seconds). The right side of the array name **Y** is **index display**. Some elements of the array are displayed, and the index of the top element is displayed in **index display**. The elements displayed will change when the value of index display is increased or decreased.

The waveform data type is very useful for **data acquisition** because it records the initial time **t0** and the sampling interval **dt** so that the sample times of all elements can be calculated.

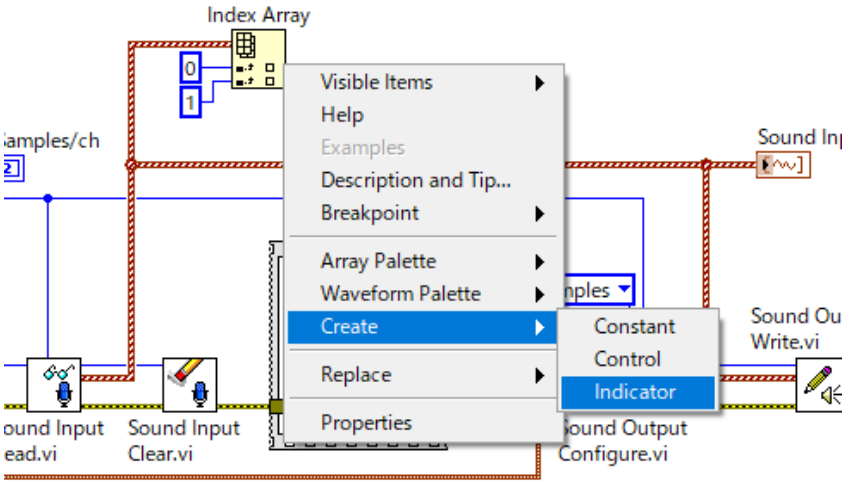


Figure 4-22 Create waveform indicator

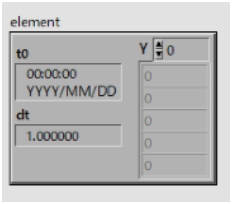


Figure 4-23 Created waveform indicator

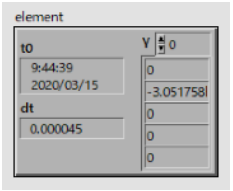


Figure 4-24 Waveform indicator filled with data after execution

Waveform Palette (**Figure 4-25**) is a collection of functions that work with waveform data types. Waveform data types are often used as inputs and outputs for frequency analysis functions.

Drag and drop **Get Waveform Components** and **Build Waveform** on the block diagram from **Waveform Palette**. Then drag and drop **Reverse 1D Array** from **Array Palette**. Place and wire these three functions as shown in **Figure 4-26**.

This allows you to reverse the order of the data in the waveform data array.

Do the same for index "1".

Drag and drop **Build Array** from **Array Palette**. The number of terminals of **Build Array** can be changed by clicking and dragging the top

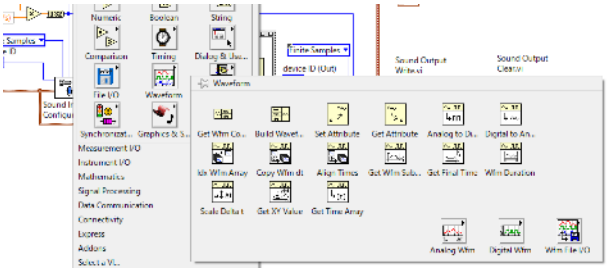


Figure 4-25 Waveform Palette

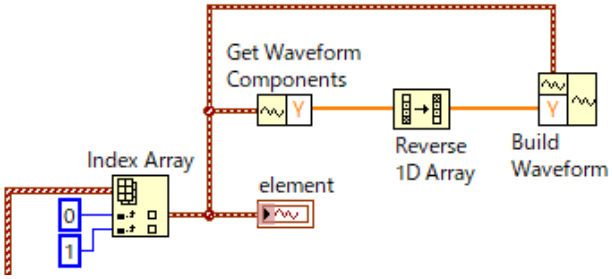


Figure 4-26 Reverse waveform data array

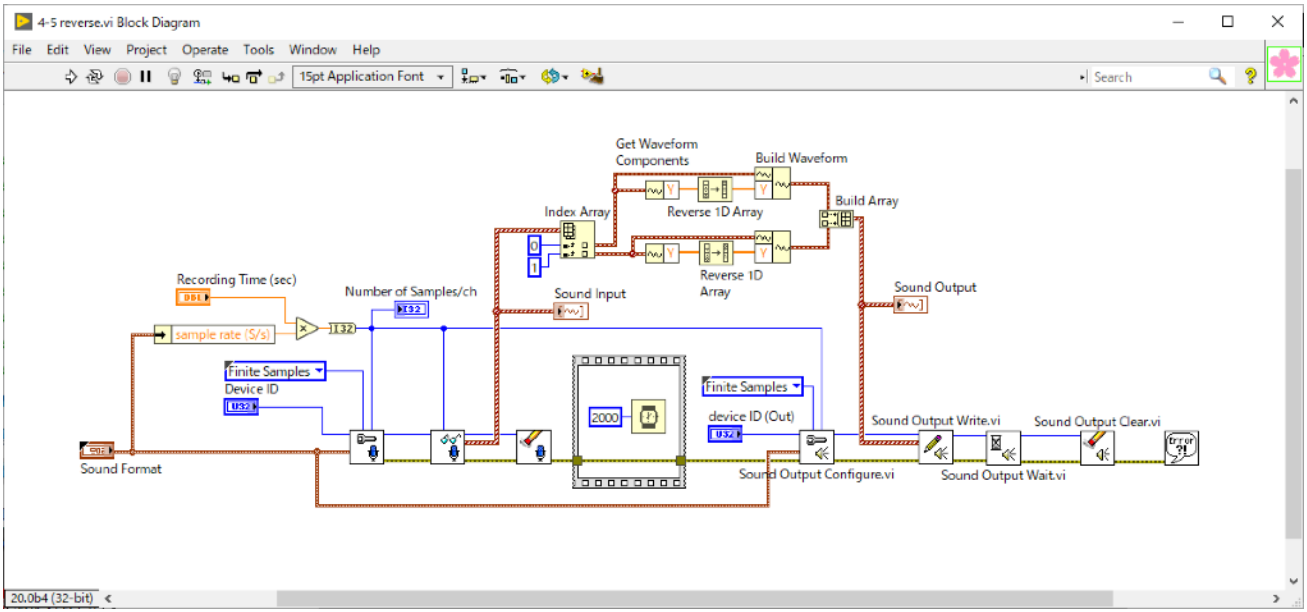


Figure 4-27 Block diagram of reverse.vi

or bottom edge of the icon. Create an array of the two waveform data modified by **Build Array** and input into **Sound Output Write.vi** (**Figure 4-27**).

When you run the completed **reverse.vi**, your words will sound like a distant foreign language. Maybe Japanese?

Write down the **phonetic symbols** of your favorite words, read them in reverse, and record them. The words played at that time should sound like the original words.

4.5 For Loop, Shift Register and Array

Now, let's take a break from making a recording and playback application and discuss **For Loops** and **Arrays**. Familiarity with For Loops and Arrays makes programming in LabVIEW easier.

Figure 4-28 is a block diagram that uses a For loop to calculate the sum from 1 to N. The For Loop will appear as if multiple sheets of paper are overlapping, and the code placed within it will be repeated a specified number of times. Enter the number of iterations in the count terminal. The shift register is a

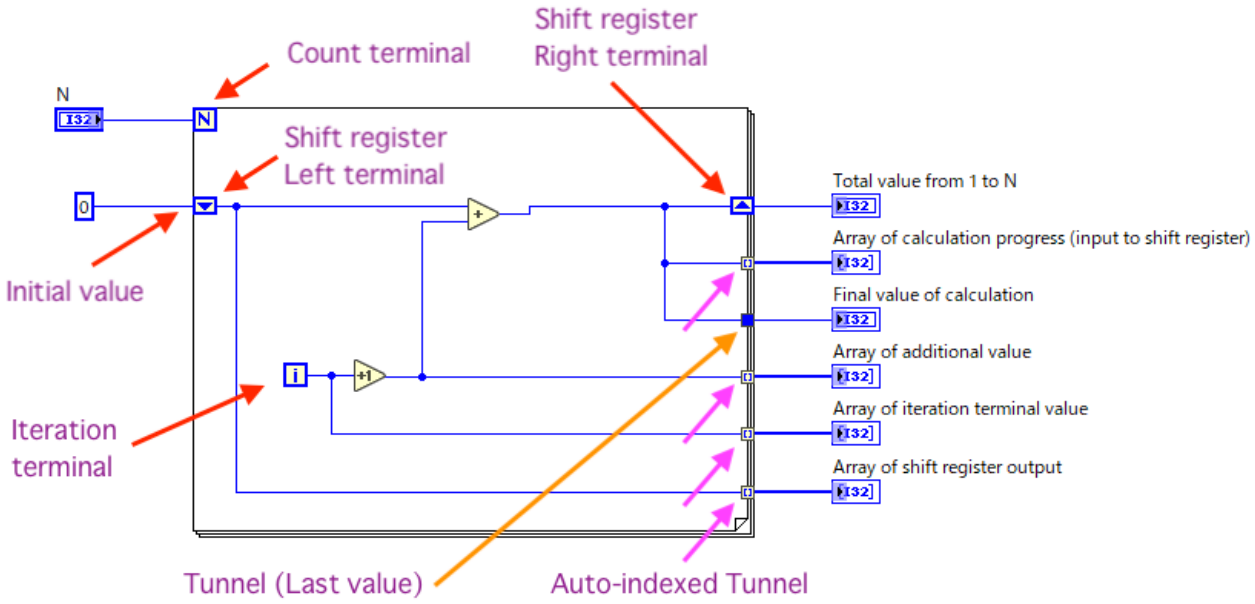


Figure 4-28 Calculate sum from 1 to N

mechanism for transmitting the execution result of the loop to the next iteration of the loop, and the value inputted to the shift register terminal at the right end is outputted from the shift register terminal at the left end in the next iteration of the loop.

In **Figure 4-28**, "0" is input to the left shift register terminal as an initial value from outside the loop, so "0" is output from the left shift register terminal in the first loop iteration. Normally, the initial value is connected, but there are special cases where the initial value is not connected.

The iteration terminal, described as "i", counts up from 0 each time the loop executes. Add "1" to the iteration terminal and add to the left shift register output. The result is connected to the input terminal of the right shift register and used in the next loop.

By repeating the loop N times, the sum of 1 to N can be obtained.

A terminal called **tunnel** is automatically generated when you wire inside and outside the loop. Two types of tunnels are created on the right side of the For loop in **Figure 4-28**.

Auto-indexed Tunnel outputs the value of each iteration of the loop to an array. When the loop executes 10 times, it becomes an array of 10 elements. **Tunnel (Last Value)** outputs only the **Last Value** of the data that came to the tunnel.

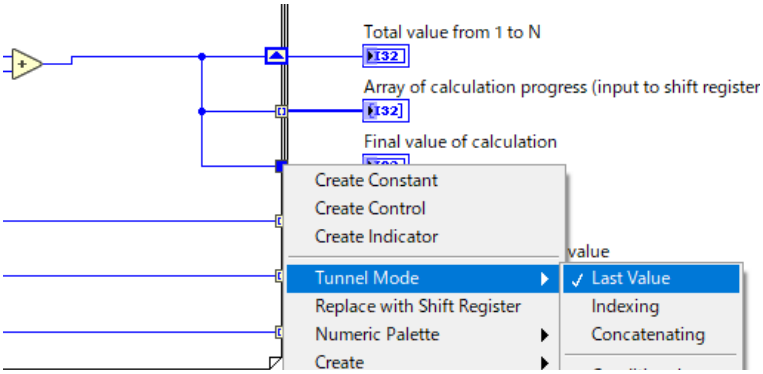


Figure 4-29 Tunnel Mode

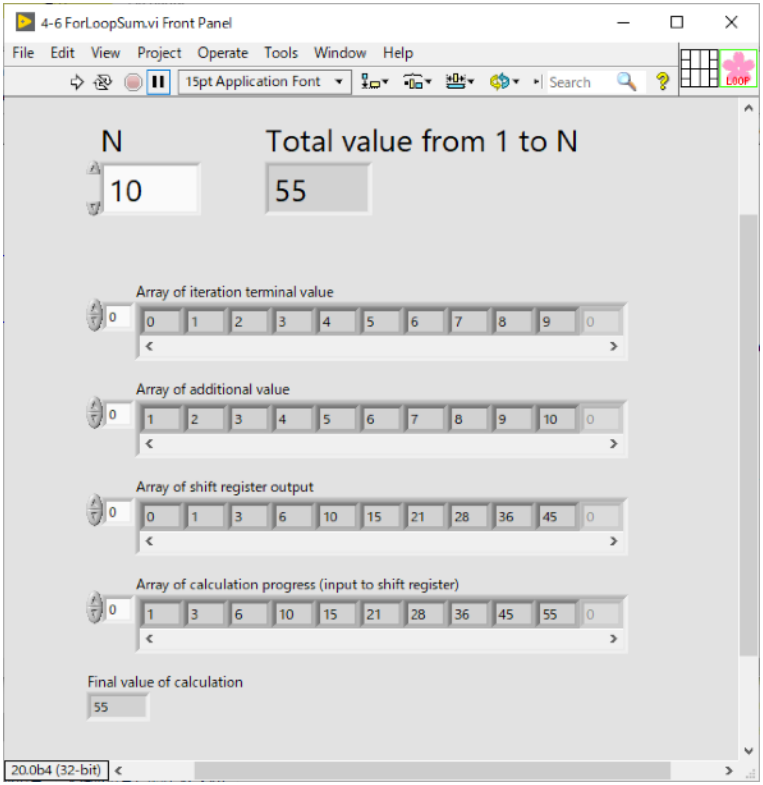


Figure 4-30 Result of calculation with N = 10

For a For Loop, **Auto-indexed Tunnel** is the default, but you can right-click the tunnel and select **Last Value** (**Figure 4-29**). Conversely, **Tunnel (Last Value)** is the default in a **While Loop**, and **Auto-indexed Tunnel** can also be selected.

When run with N = 10, the result looks like **Figure 4-30**. Turn on **Highlight Execution** and execute **ForLoopSum.vi** to check the calculation process. Also check the value output from each tunnel.

There are **conditional options** for convenient tunnel usage.

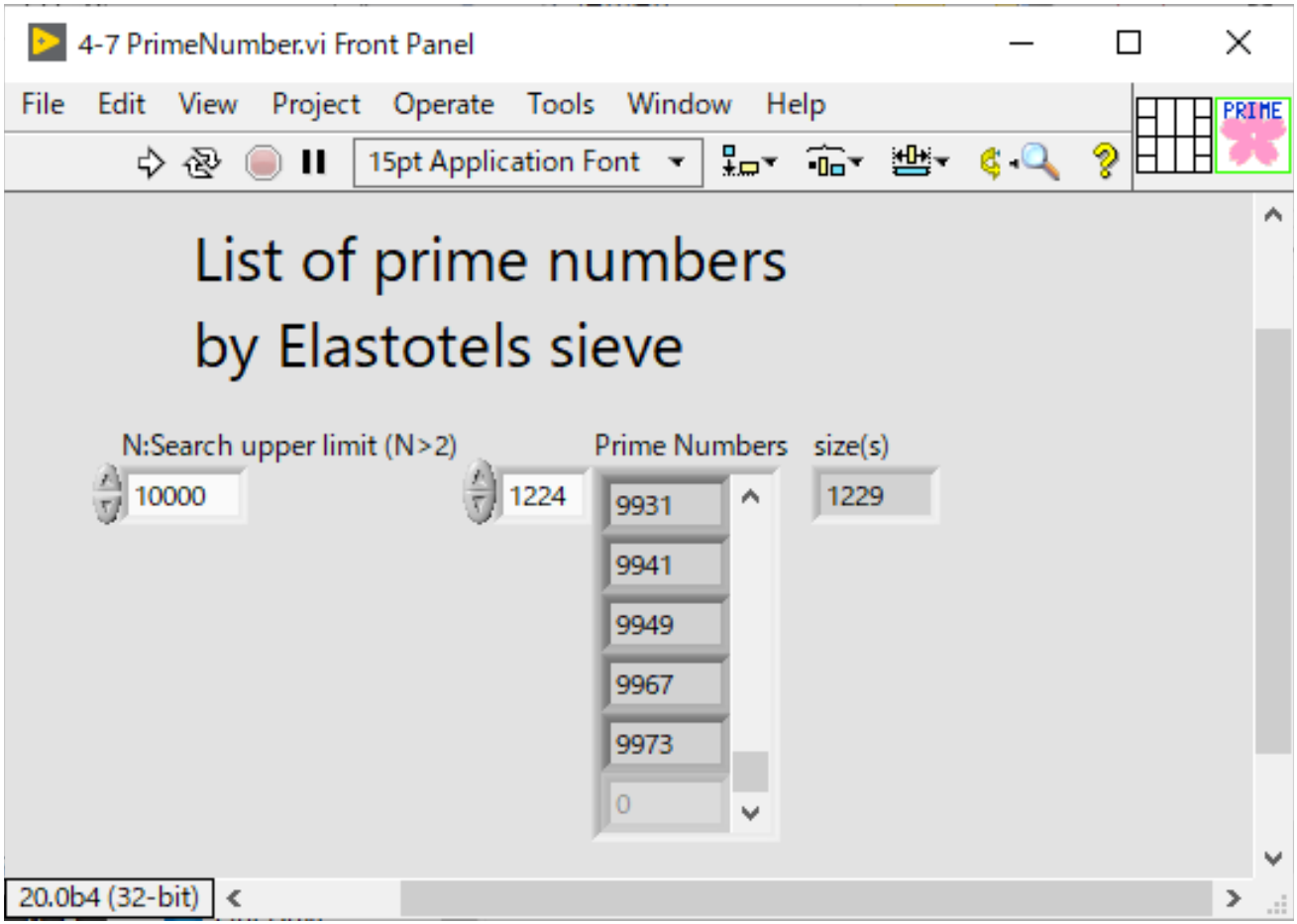


Figure 4-31 Front panel of PrimeNumber.vi

As an example of a conditional option, let's look at the **Sieve of Eratosthenes** program that outputs a list of **prime numbers** less than a specified N. A prime number is a natural number greater than 1 and can only be divided by itself.

Sieve of Eratosthenes means "if you know the prime numbers less than or equal to the square root of N, you can remove all multiples of the known prime number from numbers less than or equal to N." When we consider N = 10000, the prime numbers between 101 and 10000 are obtained by removing all multiples of prime numbers less than or equal to 100.

Even if you want to know the prime numbers up to 10000, you just have to find the prime numbers up to 100, which is the square root of 10000, so I think that many people are impressed by the powerful guidelines.

The front panel of **PrimeNumber.vi** is shown in **Figure 4-31**. Enter the maximum number to search and click **Run** button. The obtained prime numbers are output as an array. **Figure 4-32** shows the block diagram. The For loop on the left creates an array of natural numbers from 2 to the upper limit number to search. Enter this array as the initial value in the shift register of the central While Loop.

In the While Loop, use the **Delete From Array** function to remove the index 0 element from the shift register array. The deleted index 0 element can be obtained from the terminal of the **Delete from array**

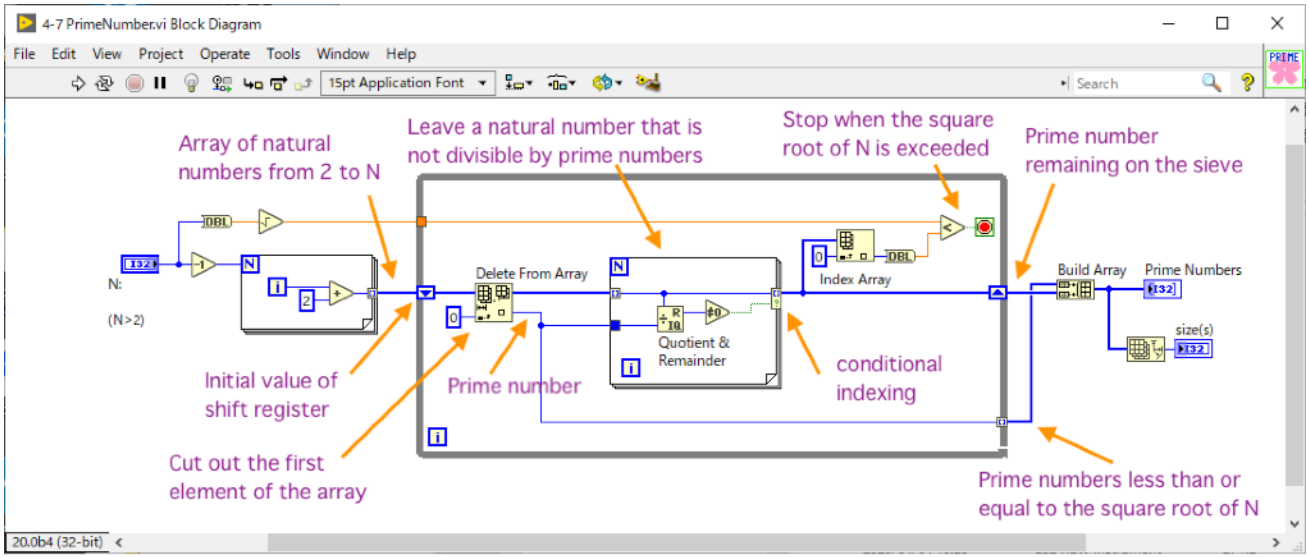


Figure 4-32 Block diagram of PrimeNumber.vi

function. The role of **Delete From Array** function here is to split the input array into the first element and the other elements.

Now let's see the process of finding the prime numbers in the While Loop.

The element at index 0 on the first iteration of the While Loop (iteration terminal "i" is 0) is 2. 2 is the first prime number. The array of natural numbers from 3 to N is the input to the For Loop, and only elements that are not divisible by the prime number 2 are output as an array. As a result of removing the multiples of 2, the index 0 in this array is 3.

The index 0 element of the array output from the shift register at the next iteration of the While Loop (iteration terminal "i" is 1) is 3 and is a prime number. This time the For Loop removes a multiple of 3.

In this way, multiples of prime numbers are excluded each time the While Loop is iterated.

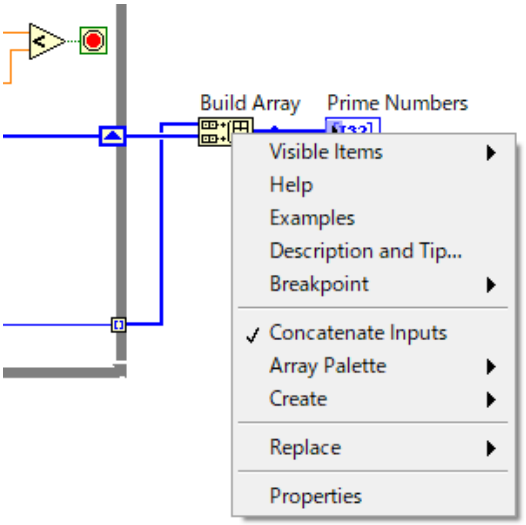


Figure 4-33 Concatenate inputs

Taking a closer look at the For Loop, it uses a **conditional index tunnel** to output an array of elements that are not divisible by a prime number. Note that nothing is connected to the count terminal of this For Loop. The array input for this For Loop is connected by the indexing tunnel, so the iteration of this For Loop will be the number of elements in the input array.

Since the beginning of the array is always a prime number, check if it exceeds the square root of N. If it does not exceed the square root of N, the process continues screening with that prime number.

If it exceeds the square root of N, the process ends.

The number of primes less than the square root of N and the number of primes remaining on the sieve are output from the While Loop. Set the Build Array setting to **Concatenate Inputs** (**Figure 4-33**) and concatenate the two 1D arrays into one 1D array.

You can expand the array's dimensions to 2D, 3D, or 4D by expanding the index display at the top left of the array.

Two-dimensional arrays are often used to represent surface information. **Figure 4-34** shows a sample display of the cone function. We used a nested For loop to write the function values to a 300-by-300 two-dimensional array. In addition to the usual two-dimensional numeric array indicator, you can visualize the data using an **Intensity Graph**, a **3D Graph**, or a **3D Mesh** (**Figure 4-35**). 3D graphs and 3D meshes can be

created from the **3D Graph Palette** in the **Graph Palette** of Controls Palette. There are some parameters for each graph, and you can change the way they are displayed. Please see the help.

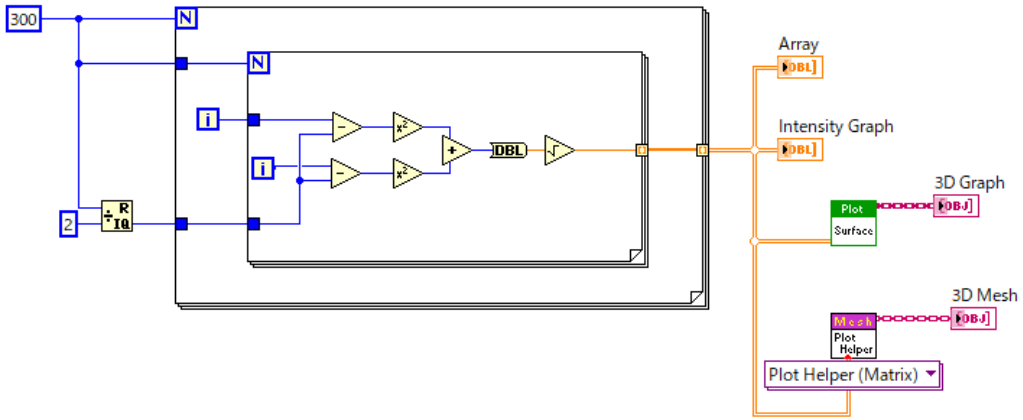


Figure 4-34 Block diagram for various graph indicator for 2D array

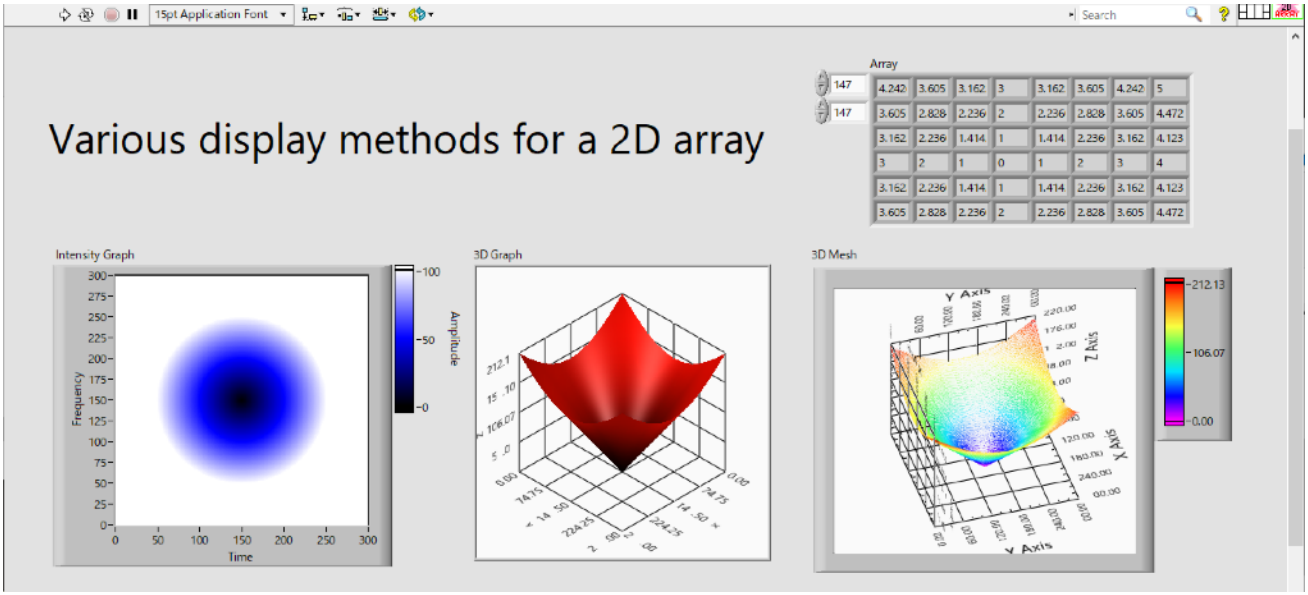


Figure 4-35 Front panel of various graph indicator for 2D array

4.6 Build Waveform Data and SubVI

Before returning to record / playback application, let's see how to build the subVI we talked about in 4.1.

As an example, we will create a program that produces the **Dual-tone multi-frequency** (**DTMF**) signal used for dialing a phone. DTMF is a method of identifying sixteen keys with a combination of four high frequency components and four low frequency components.

PiPoPa.vi (**Figure 4-36**) is a program that generates the tone signal from the speaker by inputting the push keys (0 to 9, A to D, * and #) of the telephone as a character string.

After entering the phone number in the String Control, press the Execute button and you will hear a sound. You'll be surprised if a loud sound suddenly plays, so start with a low volume.

Figure 4-37 shows the block diagram. Here, we use the **typecast** function to convert U8 to a string.

The **typecast** function reformats the data in memory so that the value could be used as another datatype. Read the help and use it.

The sound corresponding to the key is generated by the For loop one by one.

sub_KeyString.vi is a subVI that

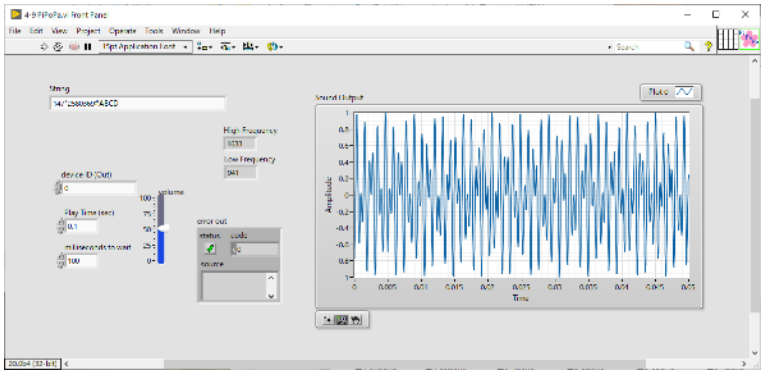


Figure 4-36 Generates dual-tone multi-frequency signale

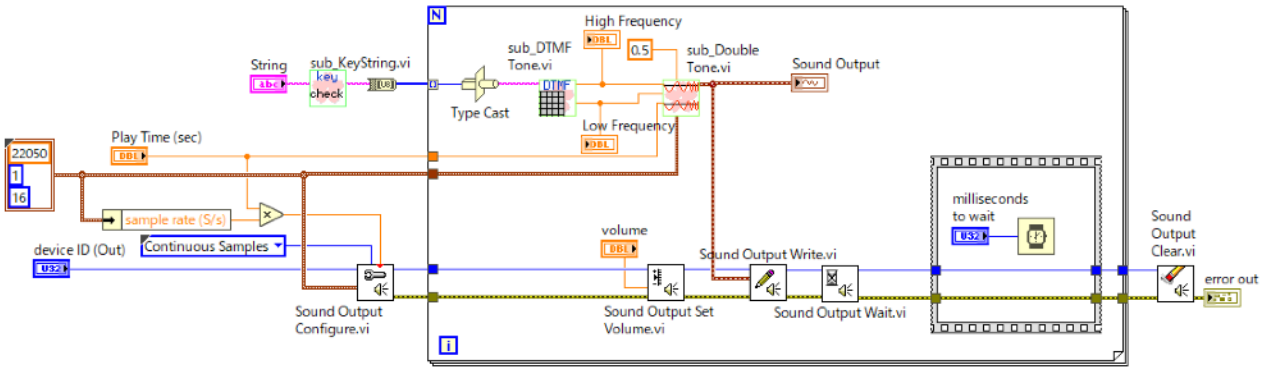


Figure 4-37 Block diagram of pipopa.vi

extracts only touch-tone characters from the string of String control (**Figure 4-38**). It converts strings into ASCII code 0-255 (U8), and if the character corresponds to touch-tone, it outputs the data using conditional index.

Figure 4-39 outputs a set of high and low frequency components for touch-tone characters.

Each column of the touch-tone keypad represents a high frequency component (1209 Hz, 1336 Hz, 1477 Hz and 1633 Hz) and each row represents a low frequency component (697 Hz, 770 Hz, 852 Hz and 941 Hz). For example, if key "6" is pressed, the high frequency component will be output at 1477Hz and the low frequency component will be output at 770Hz.

Note that Key, 2D Array Constant of string, has the same layout as the touch-tone keypad.

Figure 4-40 shows the block diagram of **sub_DoubleTone.vi**. This creates a signal that combines the sine wave of the high frequency component and the sine wave of the low frequency component.

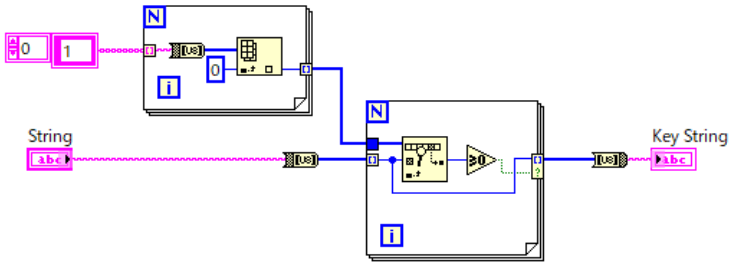


Figure 4-38 Block diagram of sub_KeyString.vi

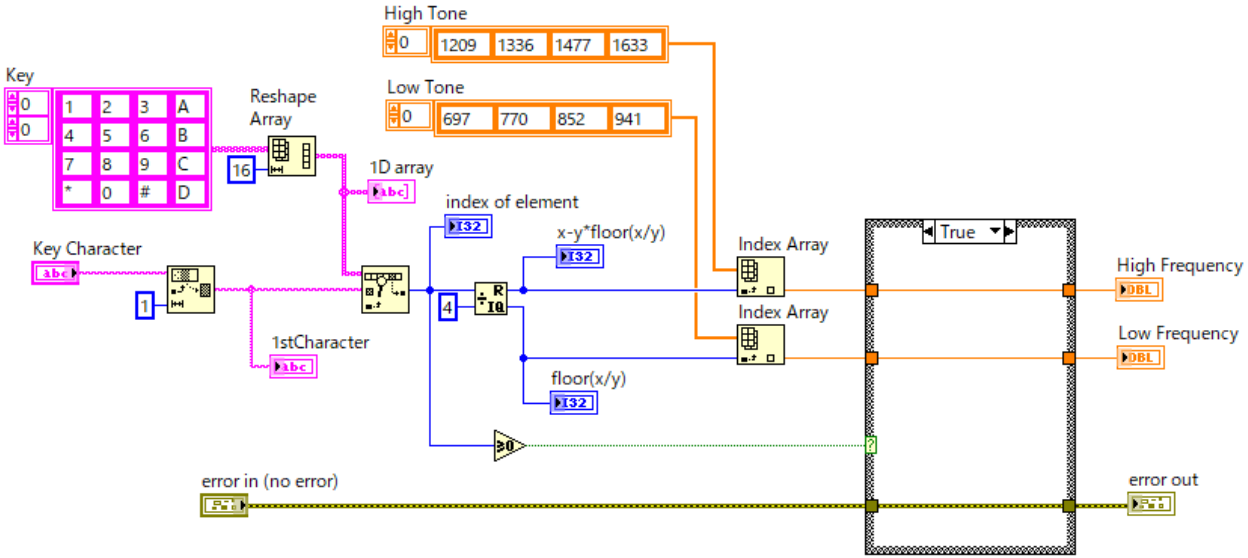


Figure 4-39 Block diagram of sub_DTMF Tone.vi

The functions needed for waveform generation are available in **Waveform Generation Palette** of **Signal Processing Palette** in **Figure 4-41**. I used Sine Waveform.vi. Input sampling info, frequency and amplitude in Sine Waveform.vi. Add two waveform data to make a composite wave.

The front panel is **Figure 4-42**.

Set the input and the output terminals when you create a subVI.

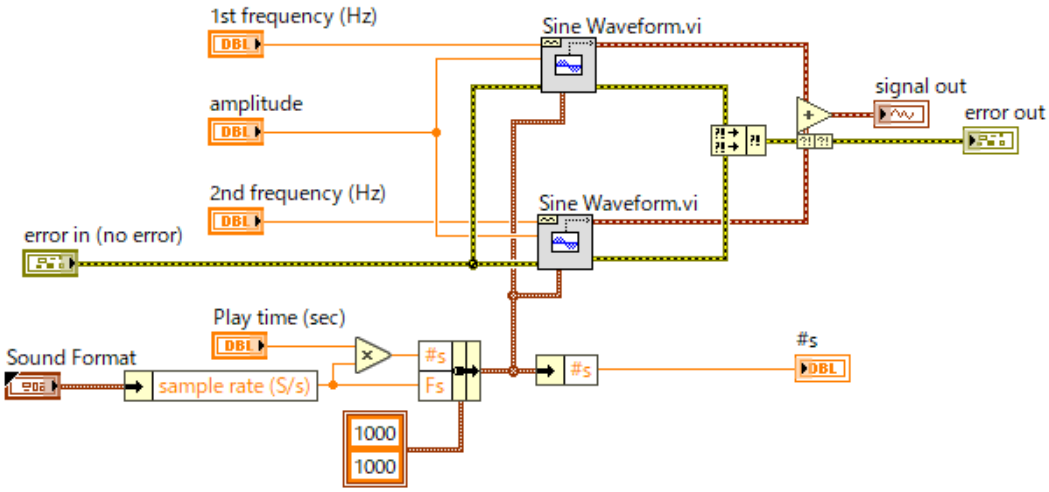


Figure 4-40 Generate DTMF signal

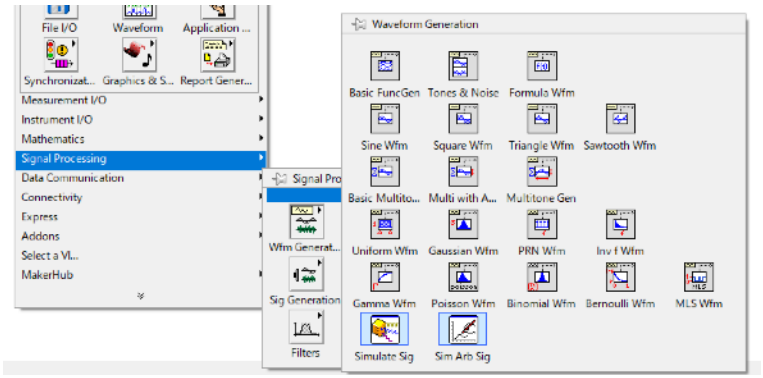


Figure 4-41 Waveform Generation Palette

Before the terminals are configured, there is a blank connector pane at the left of the default icon, as shown in **Figure 4-43**. The general rule is to place input terminals on left, output terminals on right, and error terminals on bottom for ease of wiring on the block diagram.

Figure 4-44 shows the "First Frequency (Hz)" control clicked after clicking the upper left terminal. This is how you associate the input terminal with the control and the output terminal with the indicator. To cancel the associated terminal, right-click the terminal and select **Disconnect This Terminal** (**Figure 4-45**).

You can also create your own icon design. The VI used in this book is unified with the cherry blossom icon. It would be better if the VI's functionality could be represented in an icon, but it hasn't reached that point.

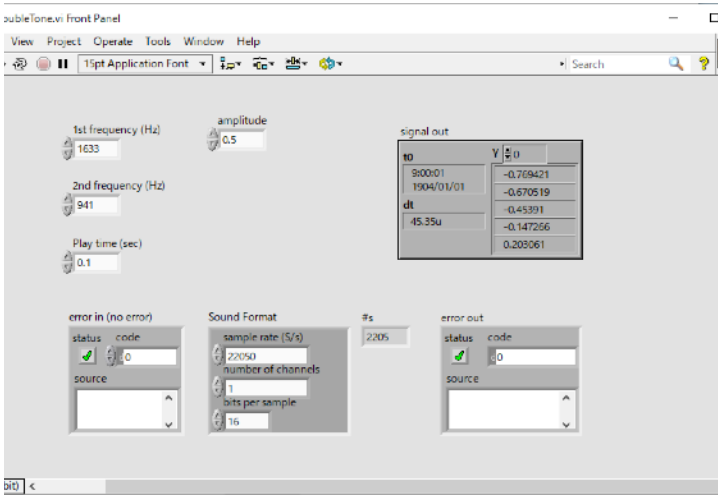


Figure 4-42 sub_DoubleTone.vi

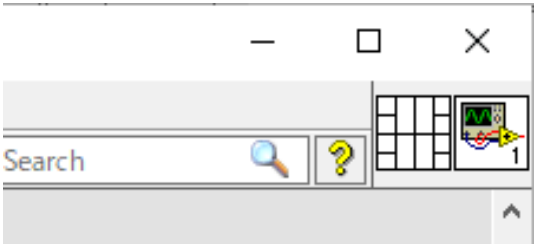


Figure 4-43 Connector Pane

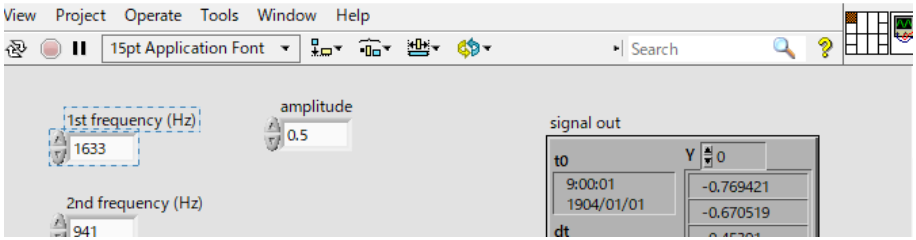


Figure 4-44 Connect terminal and control

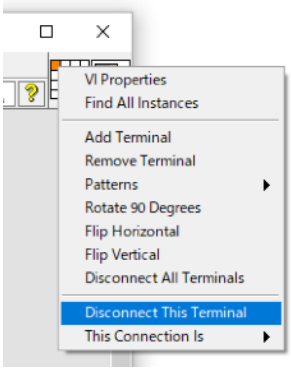


Figure 4-45 Disconnect This Terminal

Double-click the icon to open the icon editor shown in Figure 4-46. Now use **Templates**, **Icon Text**, **Glyphs**, and **Layers** to represent the VI's functionality in an icon.

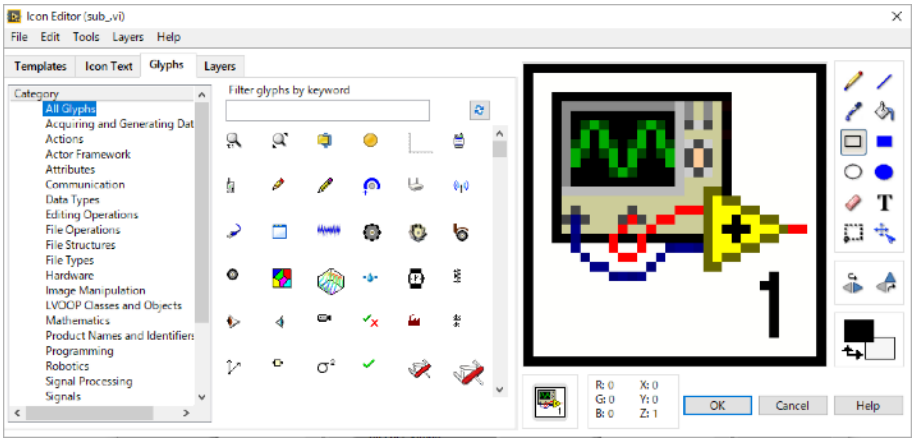


Figure 4-46 Icon editor

4.7 Record and Playback Program

Let's create a program that can record, playback, read files, and save files.

It is a simplified version of **SoundVIEW**. This program has 6 states: initialize, recording, play, save, load and end.

From the File menu, click New VI.

Save it as **easyRecorder.vi**.

Create an Enum (**Figure 4-47**) on the front panel and name it "my_state."

Right-click the Enum "my_state" and select **Edit Item ...** (**Figure 4-48**).

Enter the 6 item names as shown in **Figure 4-49**.

Right-click the Enum "my_state" and select **Make Type Definition** (**Figure 4-50**).

Right-click the Enum "my_state" and select **Open Type Definition** (**Figure 4-51**).

Save it as “my_state.ctl."

When you close the my_state.ctl window, you will see a dialog box that says "Replace the original control 'my_state' with 'my_state.ctl' ?"

Click **Yes** (**Figure 4-52**).

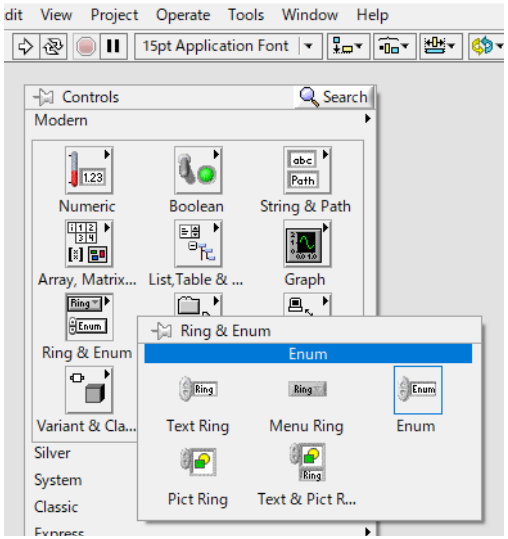


Figure 4-47 Create Enum

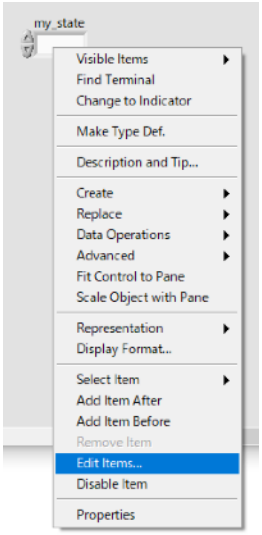


Figure 4-48 Edit items…

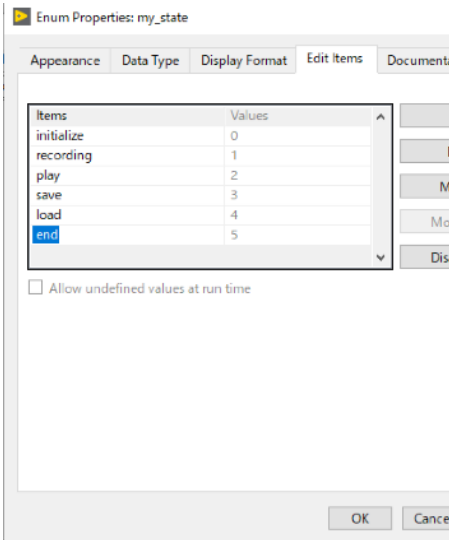


Figure 4-49 Input item name

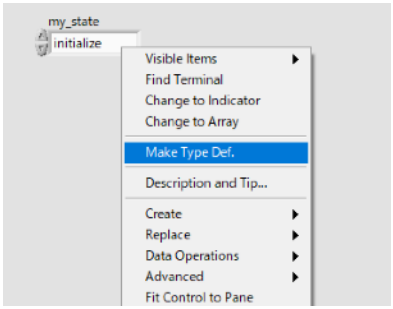


Figure 4-50 “Make Type Def.”

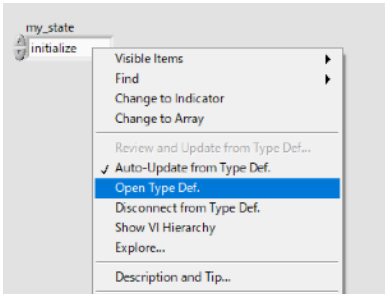


Figure 4-51 “Open Type Def.”

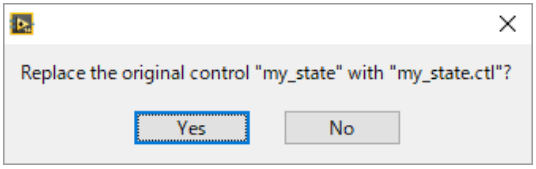


Figure 4-52 Confirmation dialog

Create a shift register in the While Loop (**Figure 4-53**).

Right-click the Enum "my_state" to create a constant, as shown in **Figure 4-54**.

Set the my_state.ctl constant to “initialize” state. Connect it to the left shift register from outside the loop.

When you connect the selector terminal of the case structure and the shift register, the item names “initialize” and “recording” of my_state.ctl are displayed on the case selector label of the subdiagram.

Display the recording subdiagram, right-click the outer frame of the case structure, and select “Add Case” from the pop-up menu to add a case, as shown in **Figure 4-55**.

Similarly, add cases to create all cases. You now have 6 subdiagrams (**Figure 4-56**).

There is also a way to create a subdiagram for all items of Enum my_state at once, right-click the frame outside the case structure and select “Add Case to All Values” from the popup menu.

We have set "my_state" to 6 items, but now we realize that we also need a “stand by” state.

Right-click the entered enumeration "my_state" and select “Open Type Def” (**Figure 4-57**).

The type definition for the Enum "my_state" opens.

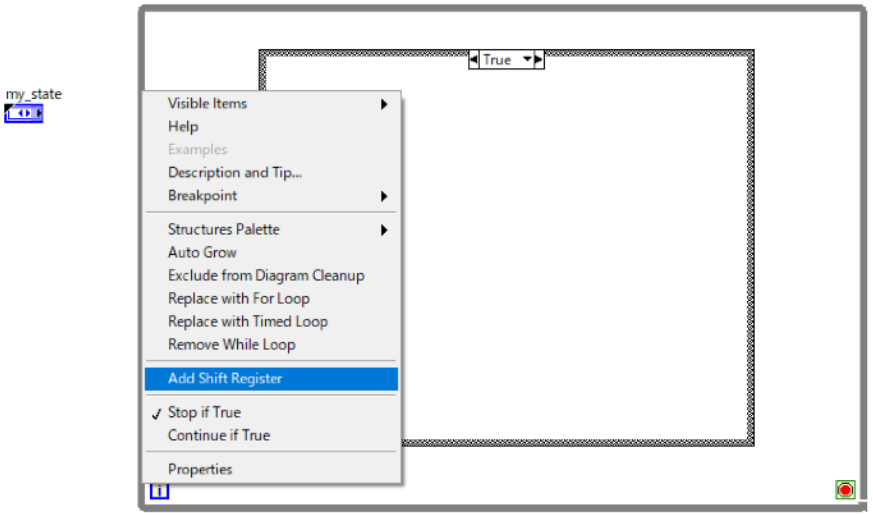


Figure 4-53 Add shift register

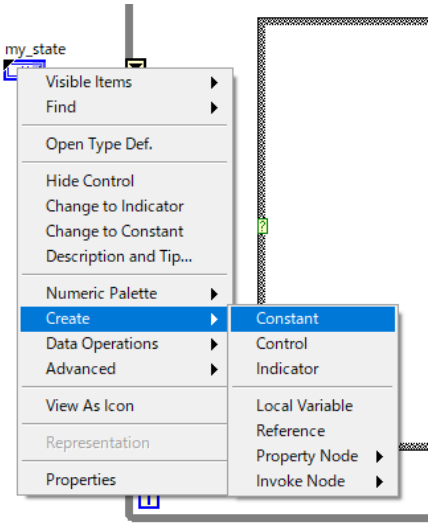


Figure 4-54 Create enum constant

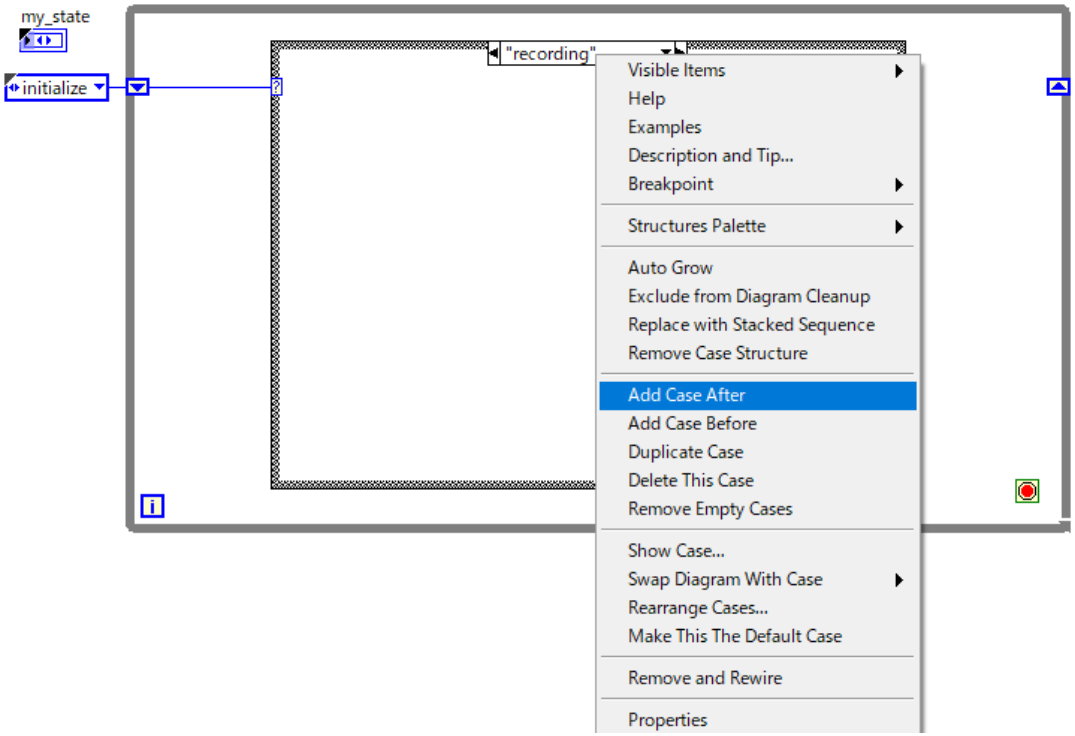


Figure 4-55 Add subdiagram next to “recording”

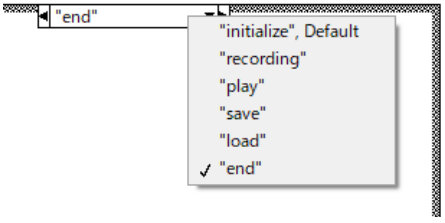


Figure 4-56 Create 6 subdiagram

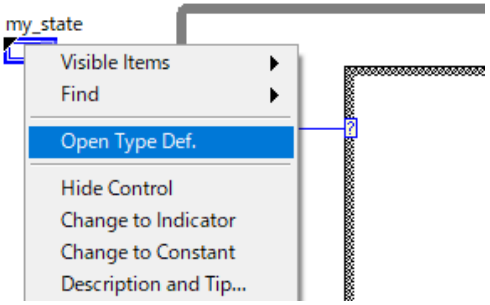


Figure 4-57 Open type definition of my_state

Click the “Insert” button and add “stand by" after "Initialize" (**Figure 4-58**).

The Enum "my_state" constants connected to the shift register are also updated, as shown in **Figure 4-59**. State machines often include enum constants, so you should be able to feel how type definition can make adapting to future changes easier.

Add a **stand by** subdiagram after **initialize** subdiagram(**Figure 4-60**).

Add some buttons and graphs to the front panel (**Figure 4-61**).

The **stand by** subdiagram outputs to the shift register to move to the appropriate subdiagram depending on the button pressed (**Figure 4-62**).

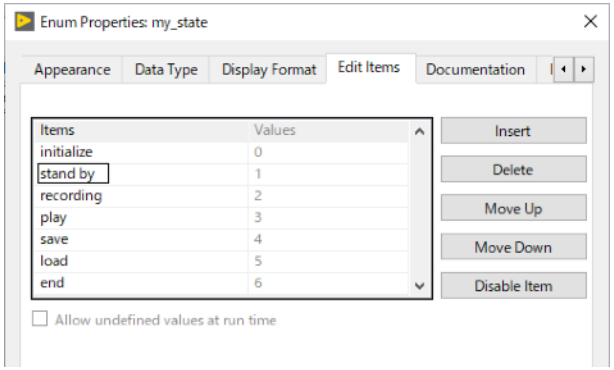


Figure 4-58 Insert “stand by"

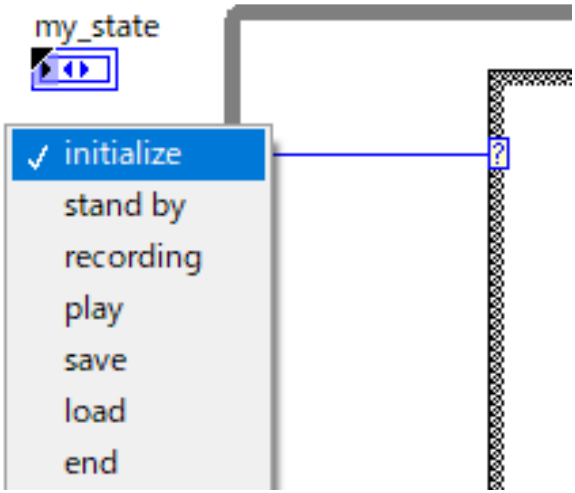


Figure 4-59 Enum “my_state" constant updated automatically

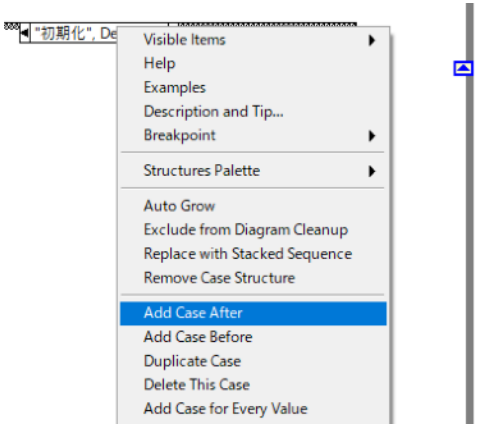


Figure 4-60 Add subdiagram “stand by” next to ”initialize”

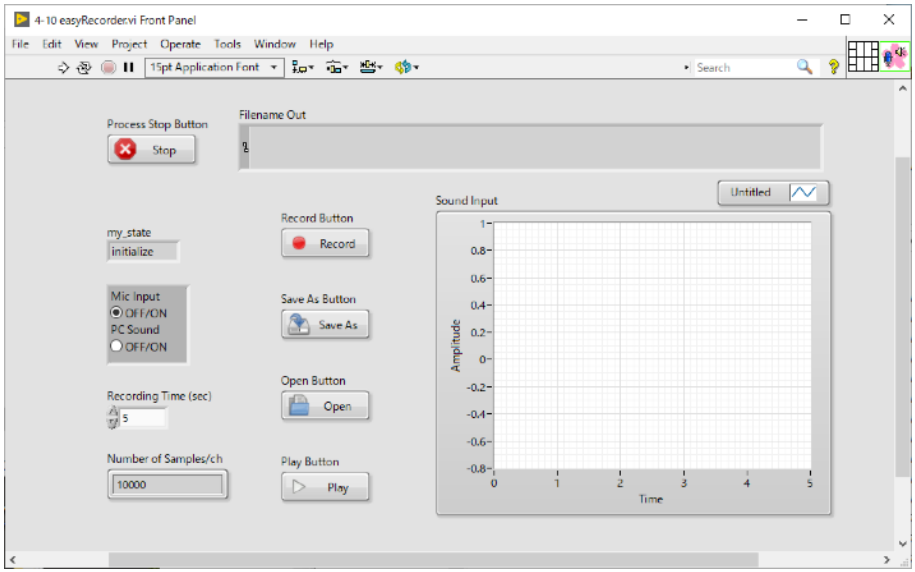


Figure 4-61 Add some buttons and graphs to the front panel

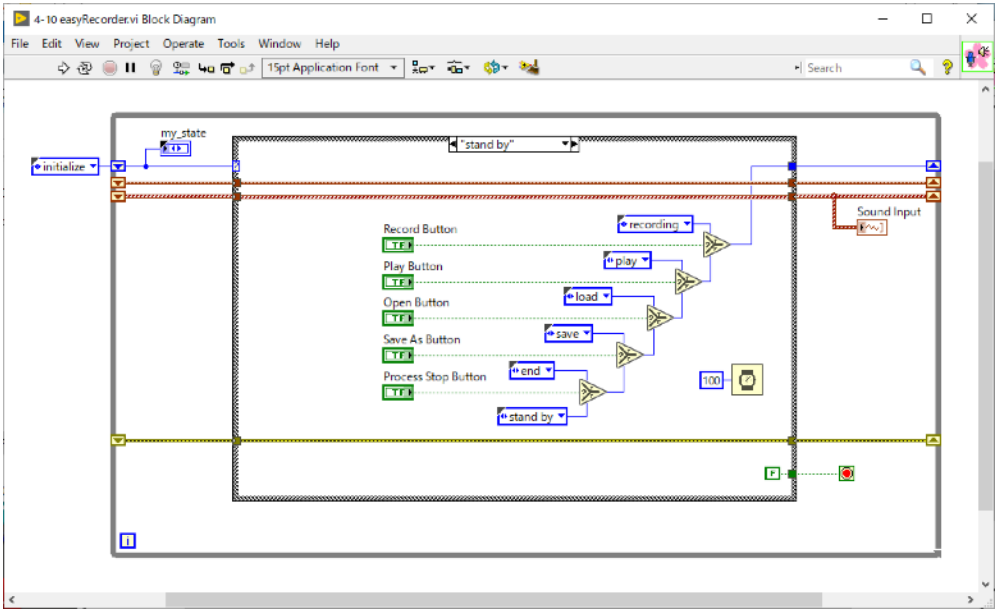


Figure 4-62 “stand by” subdiagram

The **recording** subdiagram (**Figure 4-63**) and **play** subdiagram (**Figure 4-64**) are created by copying the recording and playback portions of parrot.vi.

Collect configuration parameters in the **initialize** subdiagram (**Figure 4-65**).

The **save** subdiagram (**Figure 4-66**) and **load** subdiagram (**Figure 4-67**) use **Write to Measure File** and **Read from Measure File** on **File I/O Palette.**

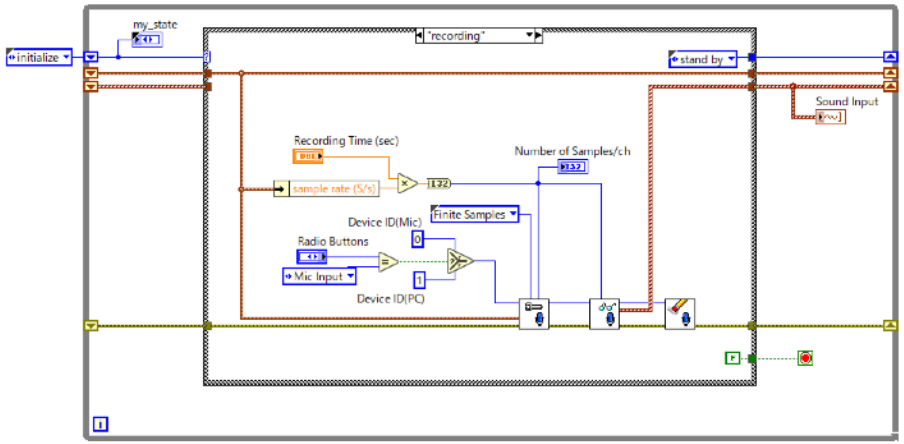


Figure 4-63 “recording” subdiagram

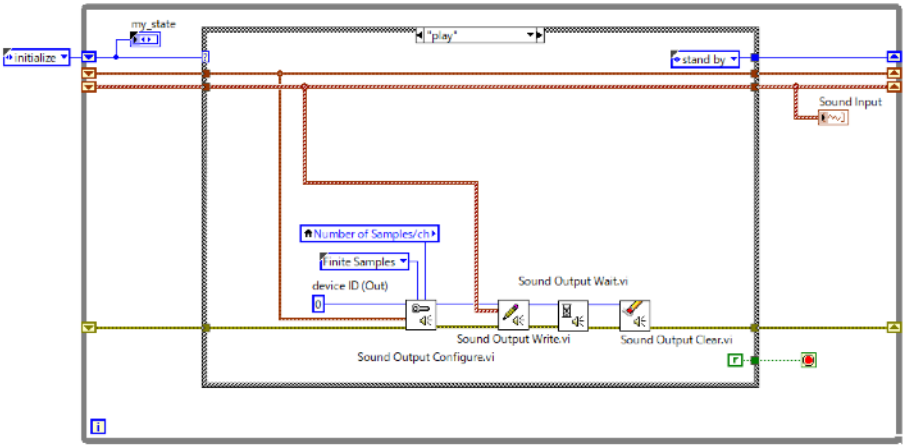


Figure 4-64 “play” subdiagram

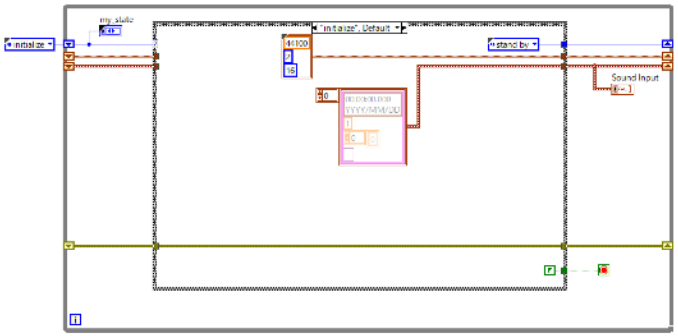


Figure 4-65 “initialize” subdiagram

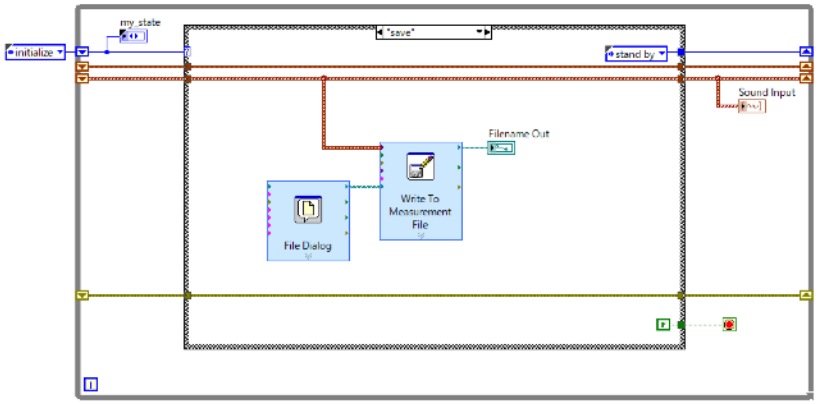


Figure 4-66 “save” subdiagram

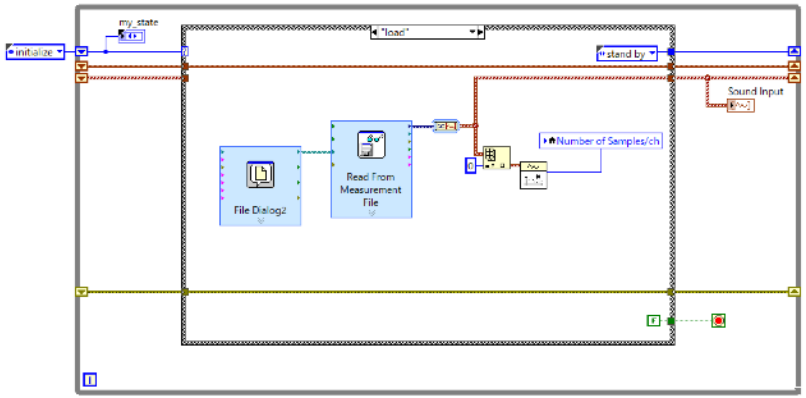


Figure 4-67 “load” subdiagram

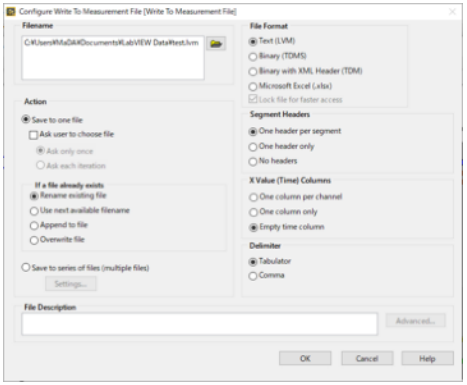


Figure 4-68 Select options for “Write to measurement file” on dialog box

Large light blue icons such as Write to Measurement File and Read from Measurement File are called **Express VIs**.

Double-click the icon to display the dialog box where you can set the parameters.

Figure 4-68 shows the **Write to measurement file dialog box**, and **Figure 4-69** shows the **Read from measurement file dialog box**.

Saving in text format is convenient because it can be opened in a spreadsheet, but the file size tends to be larger than binary. Find the file format that suits your needs.

Figure 4-70 is the **end** subdiagram.

Figure 4-71 shows the front panel during Execution.

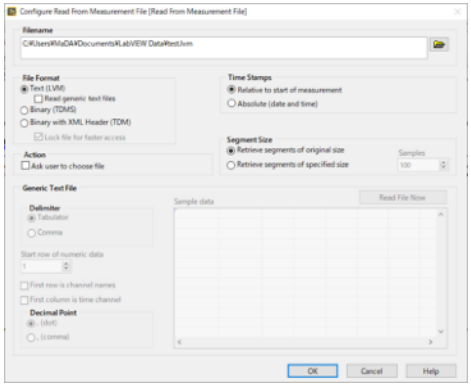


Figure 4-69 Select options for “Read from measurement file” on dialog box

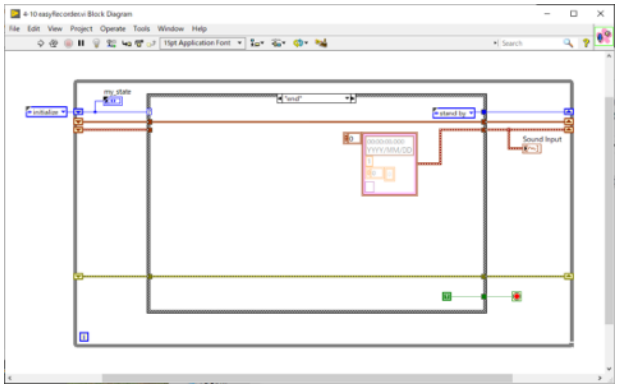


Figure 4-70 “end” subdiagram

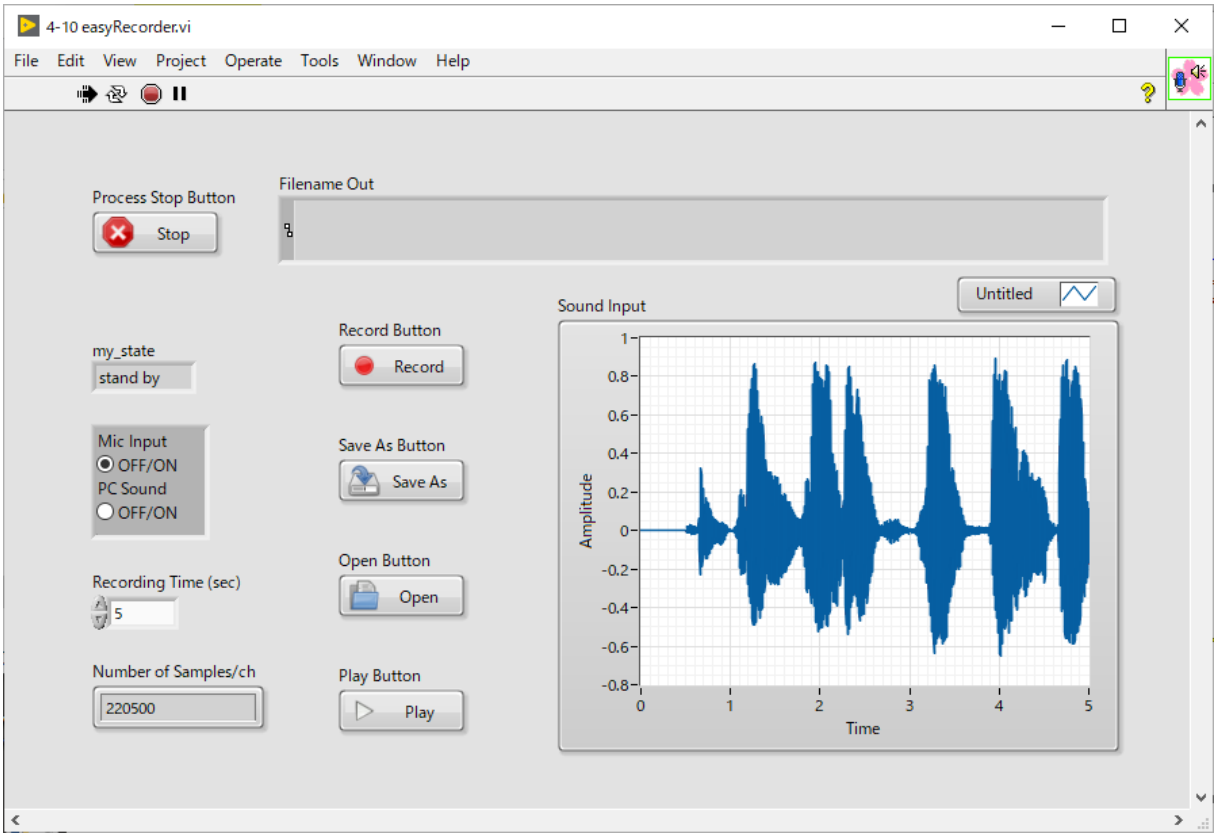


Figure 4-71 Front panel during Execution

Article 4 LabVIEW NXG Web Affinity

The websites you visit each day is created by combining various programming languages such as HTML, CSS, and JavaScript. Therefore, it is difficult to create a website from scratch.

LabVIEW NXG Community Edition comes with add-on software called the "**Web Module**" (**Figure C4-1**) that allows you to easily create a website.

Base of a website is usually made with the programming language called HTML. However, Web Module automatically converts your VI into HTML code, so you can modify a few small parts to finalize the webpage.

In order to publish the website to the rest of the world, you need a PC that runs your website application 24 hours a day, 365 days a year. If you try to do this at home, it will cost you electricity.

However, the web module also comes with a software license called "**SystemLink Cloud**" (**Figure C4-2**). Upload the website app to the cloud and it will run on the cloud. If you follow the access procedure, you can access the website from outside. Of course, there are security settings as well, so you can prevent access to strangers.

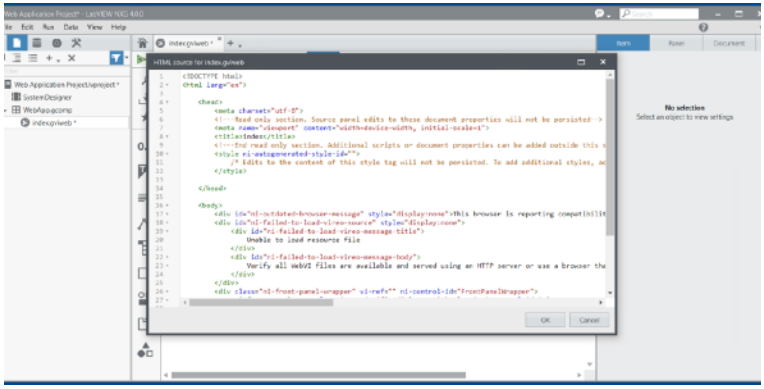


Figure C4-1 LabVIEW NXG Web Module

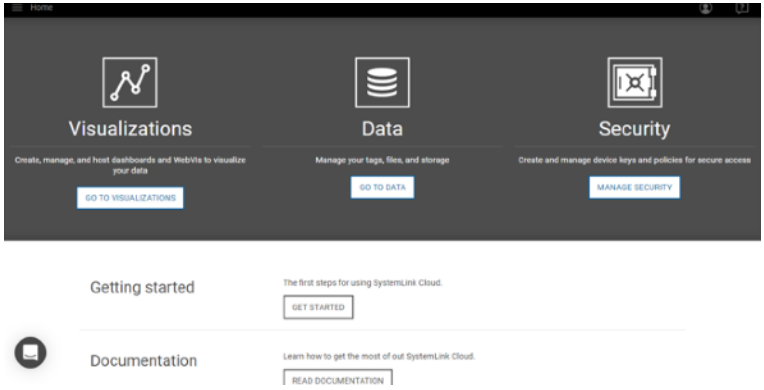


Figure C4-2 SystemLink Cloud

Section 3

Electronics projects with LabVIEW and Arduino

Those just starting electronics projects are encouraged to browse the various introduction videos on YouTube

Chapter 5

LabVIEW and Arduino

In this chapter, we use Arduino and LabVIEW to get a feel for physical computing (interactive systems that can sense and respond to the world around them). We deploy LINX firmware to an Arduino board and program it with LabVIEW. We use digital I/O and analog output with a push button switch to control fan speed.
[Keywords] Blinking LED, Arduino IDE, COM port, breadboard, LINX, LINX Firmware Wizard
[Parts used] Arduino UNO x1, breadboard x1, push button switch x1, 100Ω resistor x1, 5V DC fan x1, wires x4

5.1 How to Install Arduino IDE and Program a Blinking LED

The Arduino is a microcomputer board invented in 2005 in Italy. All its hardware schematics and source code are available for free under public licenses. As a result, the Arduino sparked a revolution almost

overnight and is now being used for do-it-yourself electronics projects all over the world. Projects range in difficulty from a simple weather display system, a fingerprint scanning garage door opener, and even a DNA sequencer. You can even build your own Arduino for about $5 using parts from your local electronics store. In this guide, we use one of the most popular Arduino board: the Arduino UNO. We will use a free development tool called Arduino IDE (Integrated Development Environment) for programming the board. There are countless tutorials online that can teach you how to program the UNO with sensors and motors. Simply search the web for "Arduino" and a sensor name (e.g. "accelerometer") and "project", and you'll find dozens of great tutorials and sample projects!

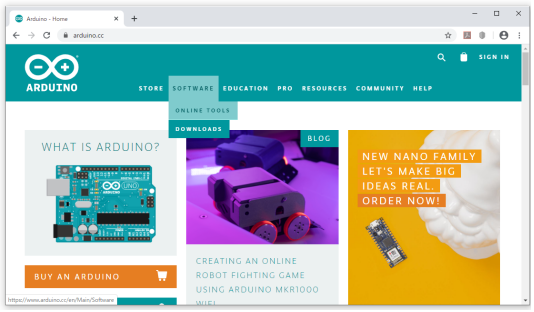


Figure 5-1 Arduino CC homepage

Let's start by downloading the **Arduino IDE**:

Find the Arduino CC homepage by following this link (https://www.arduino.cc), or by searching the web for "Arduino CC" (**Figure 5-1**).

Go to the Arduino IDE download screen by selecting "SOFTWARE" and then "DOWNLOADS".

Select the "Windows installer" (**Figure 5-2**).

Select either "Contribute & Download" or "Just download" to begin your download (**Figure 5-3**).

Figure 5-2 Arduino IDE download screen

Once the download is complete, run the installer to begin your installation. Should you have any issues during the installation, you can find installation guides by searching the web for "Arduino IDE download".

Once the installation is complete, let's open the Arduino IDE.

We can see whether the Arduino is functional by writing a simple program to blink the LED. Let's try running a sample program that does this:

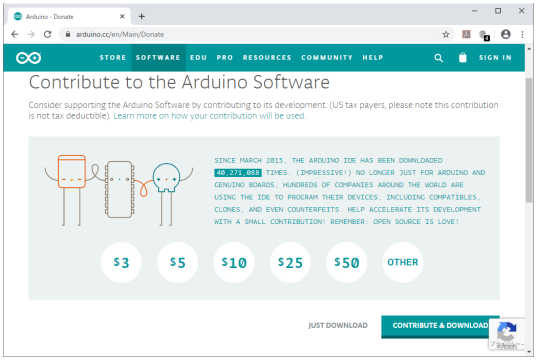


Figure 5-3 Download

In the Arduino IDE menu bar, select **File >> Sketch >> 01.Basics >> Blink** (**Figure 5-4**). A window named **Blink.ino** will open (**Figure 5-5**).

Connect the Arduino UNO to your PC via a USB cable.

In the **Blink.ino** window menu bar, select **Tool >> Board >> Arduino/Genuino UNO** (**Figure 5-6**).

In the **Blink.ino** window menu bar, select **Tool >> Serial Port >> [your port with the Arduino UNO]** (**Figure 5-7**) . Deploy and run the program on your Arduino UNO by clicking the run button on the top left corner of the window (the right-facing arrow button shown in **Figure**

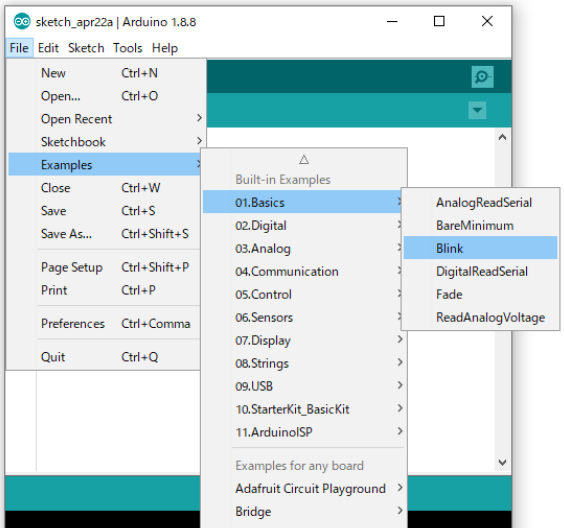


Figure 5-4 Open sample program, Blink

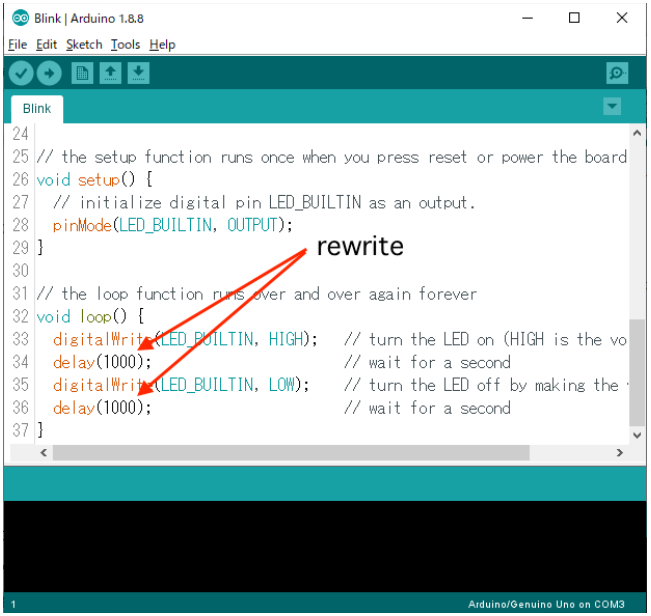


Figure 5-5 “Blink.ino” window

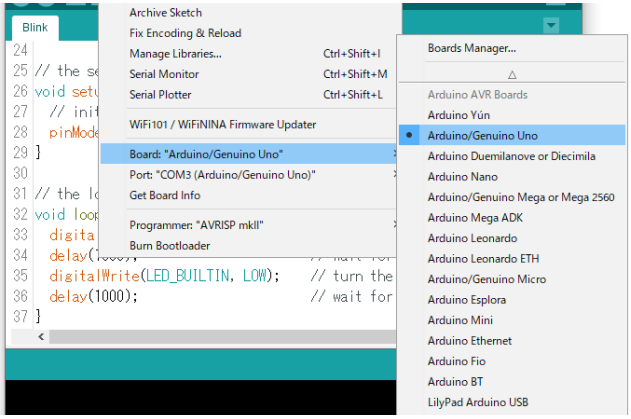


Figure 5-6 Select “Arduino/Genuino UNO”

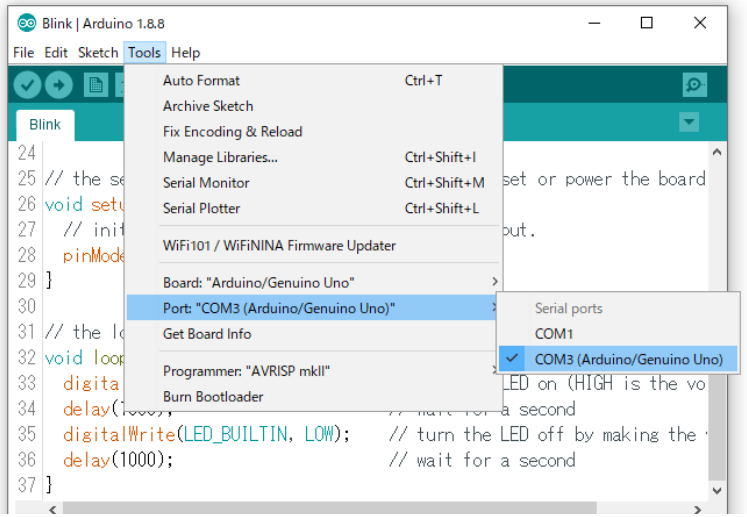


Figure 5-7 Select your port with Arduino UNO

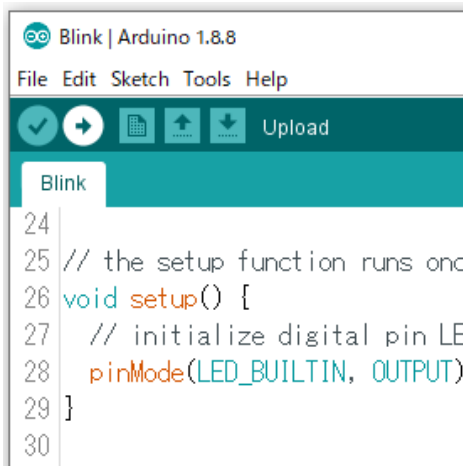


Figure 5-8 Upload “Blink.ino”

5-8).

Confirm that the **Arduino UNO LED** (shown in orange in **Figure 5-9**) is blinking every second

In fact, your Arduino UNO LED was probably already blinking every second when you connected to it via USB since UNOs are generally shipped with the Blink sample program already deployed. Let’s try changing the blink speed to make sure we’re in control:

In the **Blink.ino** window, change line 2 from “delay(1000);” to “delay(100)” as indicated in **Figure 5-5.**

Deploy and run the program on your Arduino UNO by clicking the run button on the top left corner of the window (the right-facing arrow button shown in **Figure 5-8**).

Confirm that the Arduino UNO LED now blinks several times per second.

5.2 Arduino Inputs and Outputs

Let’s learn about the Arduino UNO’s primary input and output pins in **Figure 5-9**.

Pins 2 through 13 at the top of the figure can be used for digital input or output. Pins 0 and 1 are used for USB serial communication, so they’re generally not used for digital I/O. Pin 13 is internally connected to the LED we were just controlling. Pins with a tilda (~) before the number can also be used to create a PWM (pulse

width modulation) signal, which allows you to adjust an LED's brightness or change a fan's speed between a range of 0 and 255. This is done through a 480Hz 0-to-5V pulse signal with 256 different pulse widths.

The pins at the bottom of **Figure 5-7** are the analog input and the power source pins. Pins A0 through A5 each read 0-5V at roughly 5mV increments, but pins A4 and A5 can also be used for I2C communication (explained further in Chapter 7 of this guide).

You can continue reading this guide if you'd like a more intuitive way to program the Arduino using LabVIEW! (If you'd like to know more about programming using the Arduino IDE, there are hundreds of helpful tutorials online; just search the web for "Arduino IDE tutorial".)

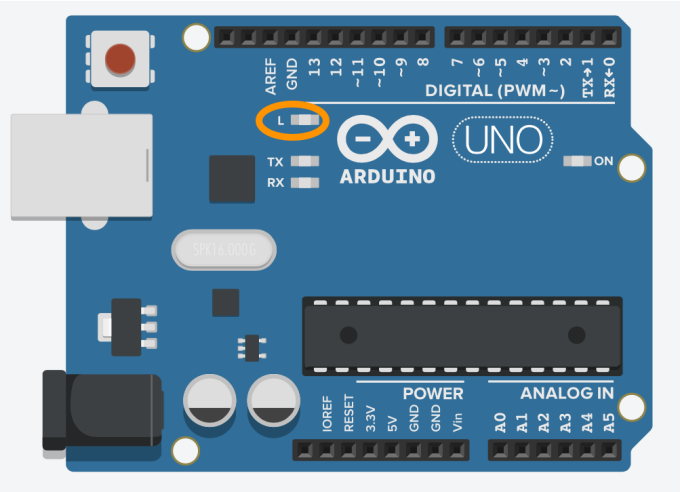


Figure 5-9 LED and input / output pins of Arduino UNO

5.3 Controlling your Arduino with LabVIEW

You can use your Arduino's I/O ports from LabVIEW using a LabVIEW addon named **LINX**. LINX is a free addon provided by Digilent Inc (a National Instruments Company) and is a convenient way to control common embedded platforms like Arduinos, Raspberry Pis, and BeagleBone Blacks using LabVIEW. They have a webpage on **LabVIEW MakerHub** with tutorials, FAQs, and support forums, so please visit them for more information (**Figure 5-10**).

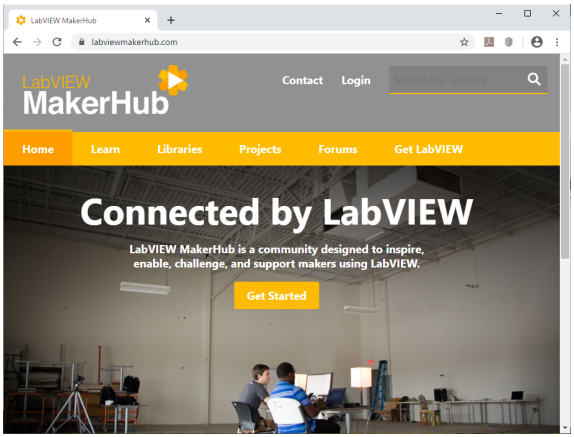


Figure 5-10 Webpage on LabVIEW MakerHub

After deploying the LINX firmware to your Arduino, you can send commands and data via the serial port with easy to use LabVIEW VIs. LabVIEW Community Edition has LINX preinstalled, but on other Editions you will need

to install LINX through the **VI Package Manager** (**Figure 5-11**). Although easy to use, the LINX interface introduces some latency between commands and their responses and is not recommended for high-speed communication. In Chapter 7, we will cover how to control an Arduino without using LINX.

We will use the **LINX Firmware Wizard** to deploy the **LINX firmware** to the Arduino UNO. (The LINX firmware is just another Arduino program, so we can deploy other programs like **Blink.ino** and use the Arduio for other things as well.)

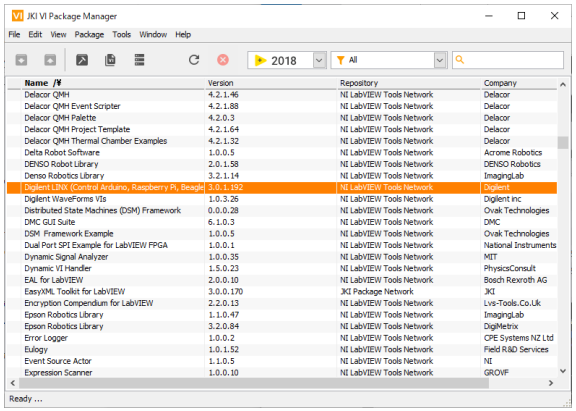


Figure 5-11 VI Package Manager

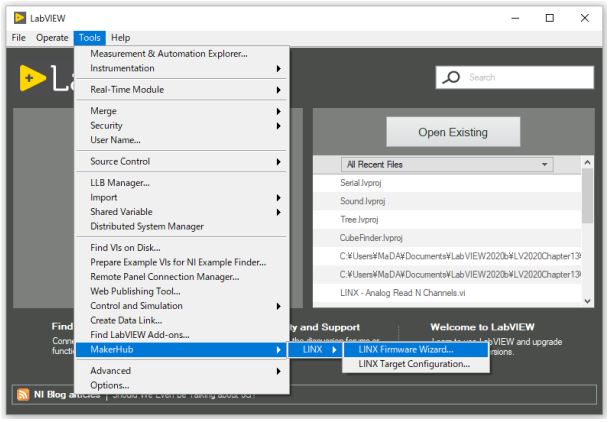


Figure 5-12 LINX Firmware Wizard..

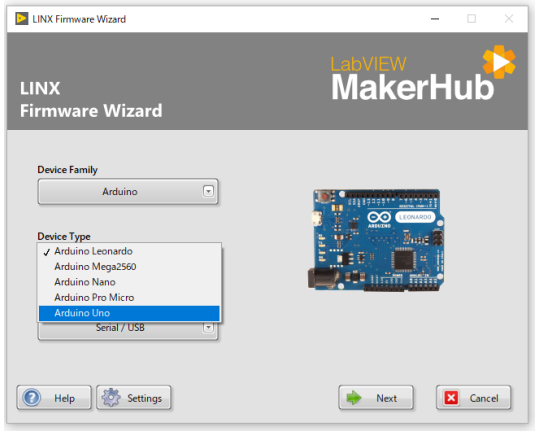


Figure 5-13 Select Arduino UNO

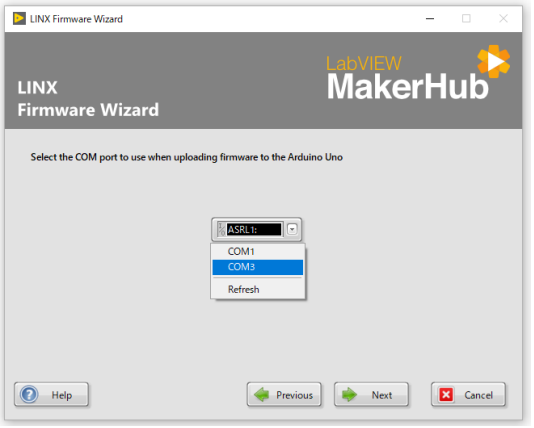


Figure 5-14 Select serial port

From the Tools menu, select **MakerHub -> LINX -> LINX Firmware Wizard** (**Figure 5-12**). In the next window, select **Device Type -> Arduino UNO** and press the Next button (**Figure 5-13**). Next, select the COM port with your Arduino UNO and press the Next button (Figure 5-14). The next screen with ask you to choose between Firmware Version and Upload Type, but just click the Next button on the bottom (**Figure 5-15**). The LINX firmware deployment will then begin (**Figure 5-16**). When the deployment is complete, the window shown in **Figure 5-17** will appear. Click the Launch Example button to load **LINX – Blink (Simple).vi**.

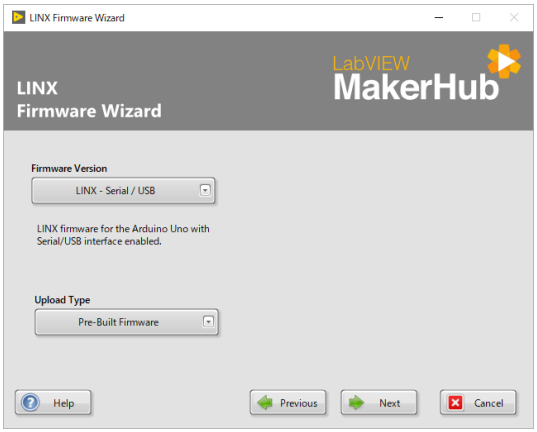


Figure 5-15 Click “Next” button

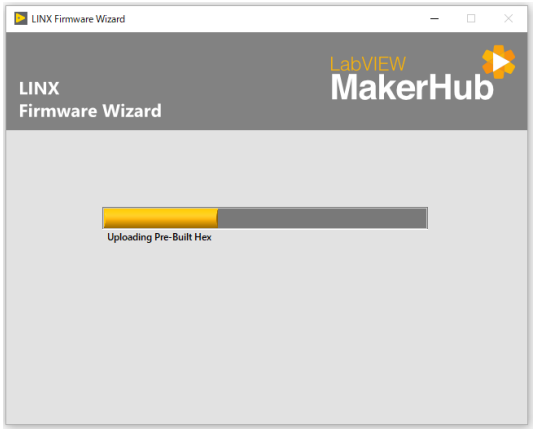


Figure 5-16 LINX firmware deployment

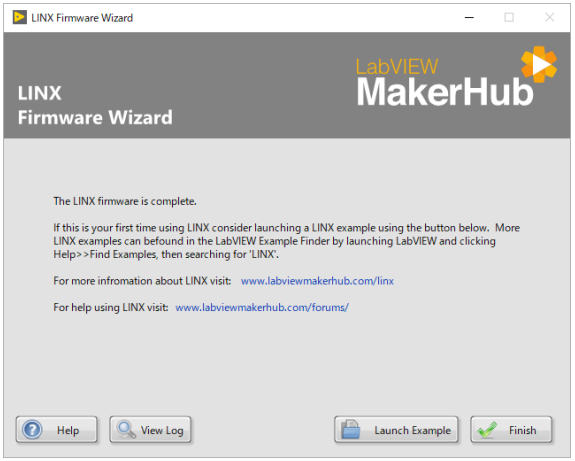


Figure 5-17 Deployment is complete

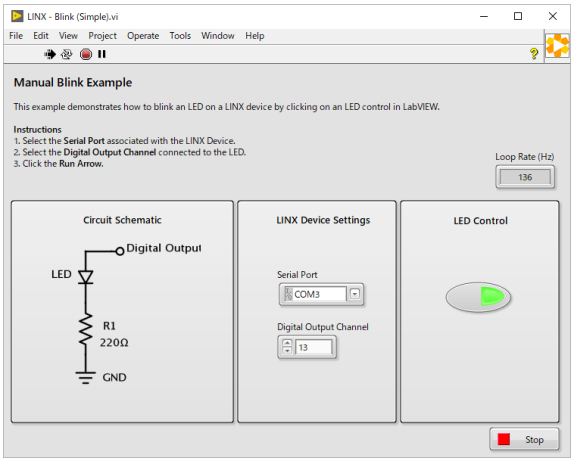


Figure 5-18 LINX - Blink (Simple).vi

Leave the "Digital Output Channel" at 13, select the serial port, and click the Run button (Ctrl+R, or the arrow button on the top left of the window). It will take about 5 seconds to respond, but you'll soon see the Loop Rate (Hz) updated to 100 which means the program has started (**Figure 5-18**). You can then use the **LED Control** button to turn the Arduino LED on or off.

Now, we are ready to control the Arduino. Please look at the digital output sample block diagram in **Figure 5-19**.

The reason why the program didn't respond right away was because it was checking the deployed LINX firmware in the **Open.vi**. The command being sent to the Arduino depends on the board type, so the program takes the time to make sure it's connected to the correct board type.

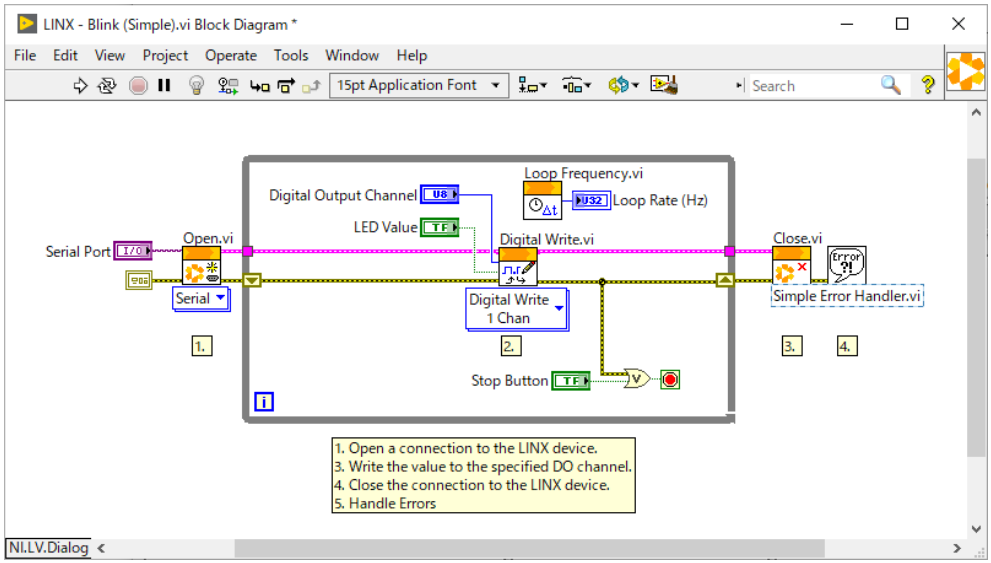
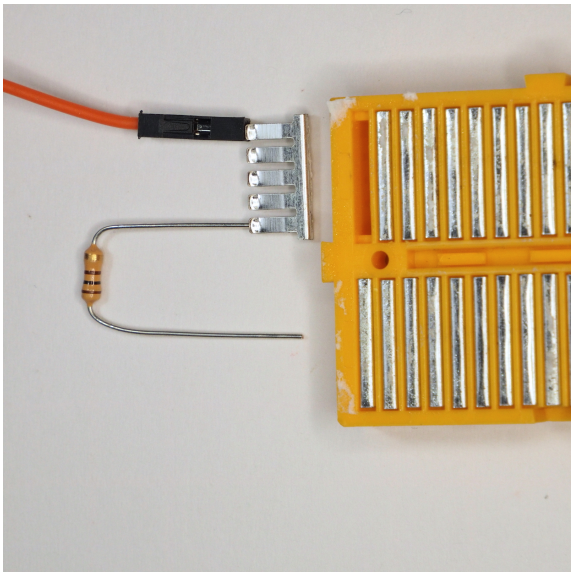


Figure 5-19 Block diagram of "LINX - Blink (Simple).vi"

5.4 Make a Switch Counter

Unplug the Arduino from USB port before wiring. Let's make a program for digital-input. A counter counts up one by one when the switch is pressed and goes back to 0 when it exceeds 10.

Place a tact switch and a resistance on the breadboard as shown in **Figure 5-20**. The two pins on same side of the tact switch are conducted when the switch is pressed. Breadboard contains multiple metal rails in it. You can see the metal rails as shown in **Picture 5-1** when you remove a tape on back side of it. Once pins of components are inserted to holes of breadboard, it establishes electrical connections by sandwiching the pins with springs of the rails. Breadboard is useful because it can make a circuit without soldering iron, which helps you to do trials and errors easily . Be sure to check positions of the rails inside if it's your first time to use breadboard since there are various types of breadboard.



Picture 5-1 Metal rails in breadboard

The connection starts from 5V pin of the Arduino and

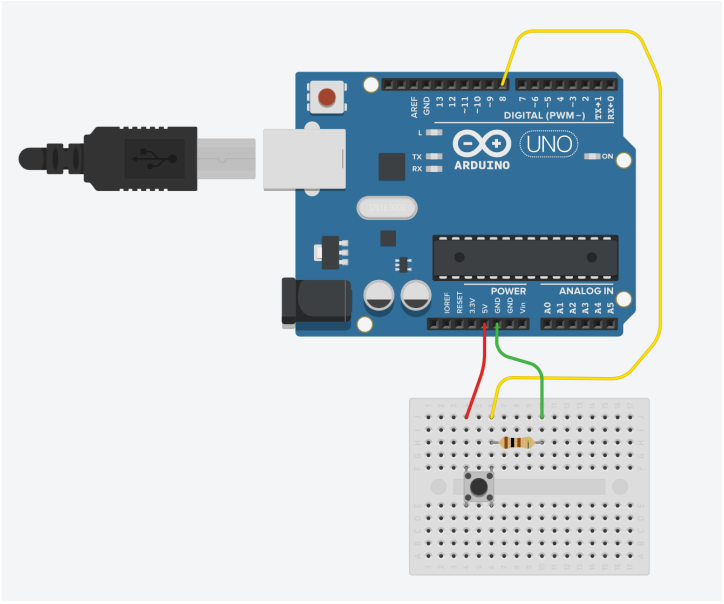
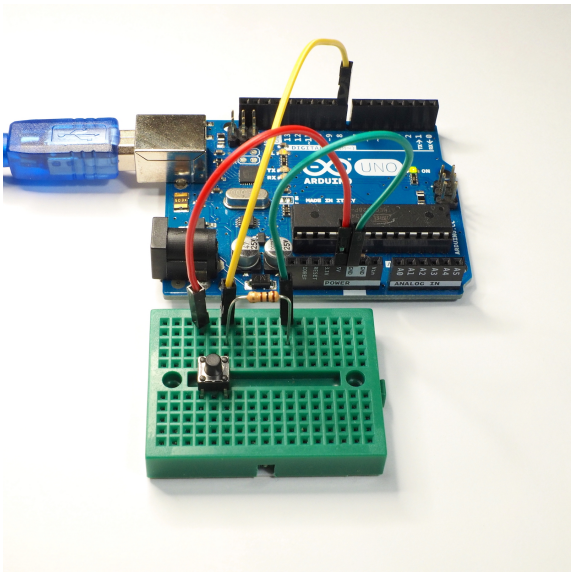


Figure 5-20 Wiring diagram



Picture 5-2 Placement and wiring of parts

continues to the left pin of tact switch, the right pin of that, 100Ω resistance and the ground.

Connect Pins 8 to the same rail as the right pin of the tact switch and the resistance are connected. Compare your wiring with **Picture 5-2**.

Use a digital I/O sample VI **LINX – Digital Read 1 Channel.vi**. As shown in **Figure 5-21**, select "Find examples.." from help menu (**Figure 5-22**).

Following **Figure 5-23**, select a serial port

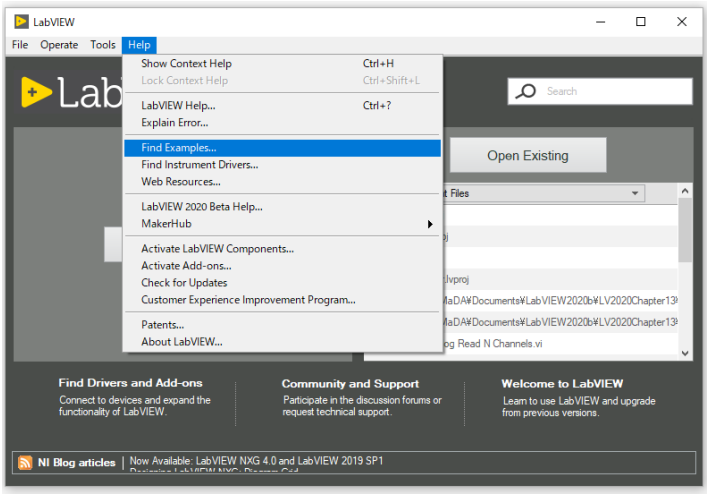


Figure 5-21 Select "Find examples.." from help menu

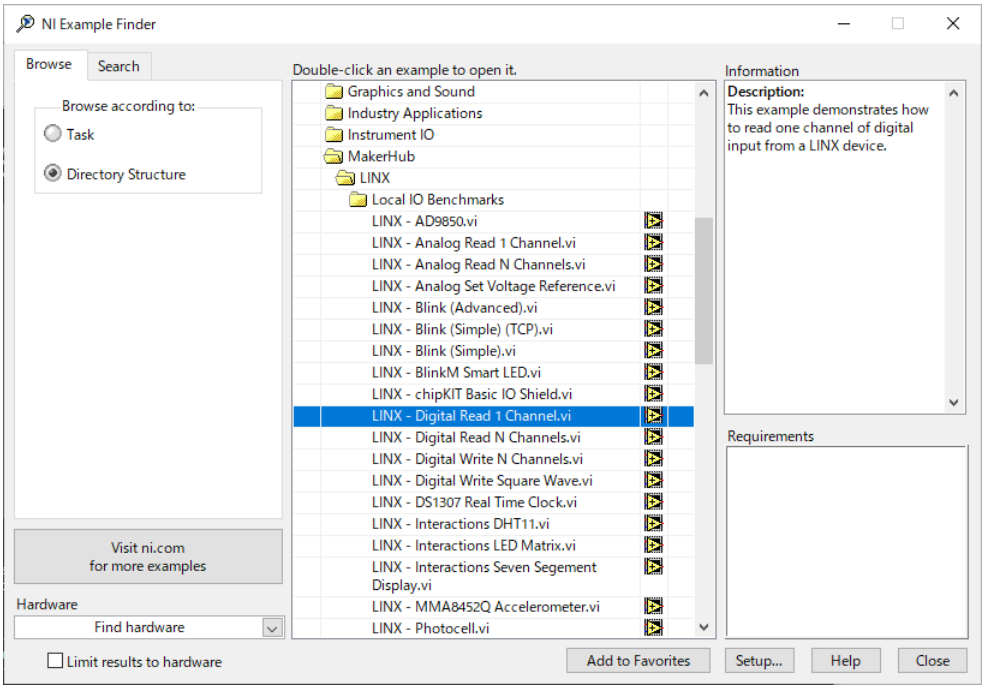


Figure 5-22 Find Examples…

connected to the Arduino and set DI Channel to 8, then press the run button. Check if DI Values turns ON as you press the tact switch. A block diagram looks similar to that of **LINX - Blink (Simple).vi** (**Figure 5-24**).

Save the program as **PushCounter.vi**, let's make a counter which counts up when the switch is pressed and released, and goes back to 0 when it exceeds 10.

Hint 1:Wait until the switch is turned OFF after it's turned ON.
Hint 2:Keep counter value by using shift register.
Hint 3:Input 0 to counter value if counter value exceeds 10.

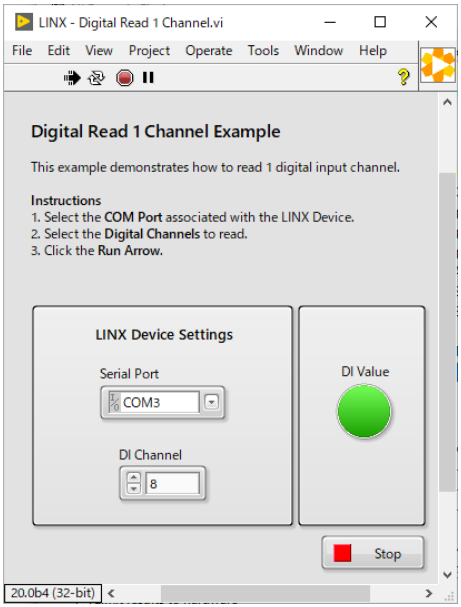


Figure 5-23 LINX - Digital Read 1 Channel.vi

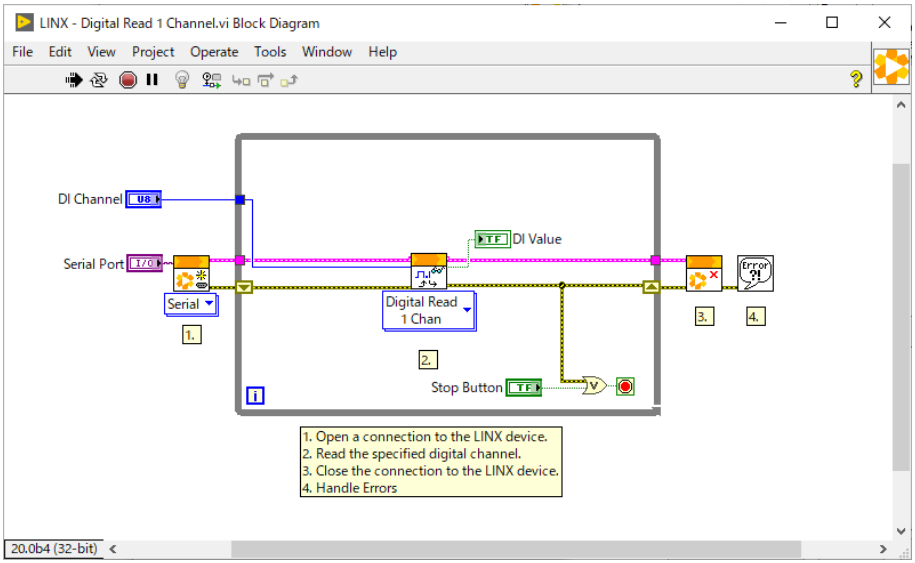


Figure 5-24 Block diagram of "LINX - Digital Read 1 Channel.vi"

5.5 Make a Fan Controller

Unplug the Arduino from USB port, make additional wiring for a fan as shown in **Figure 5-25**. Connect the black wire to ground, the red wire to Pins 9. Refer to **Picture 5-3** as well.

Open a sample VI of LINX, **LINX - PWM 1 Channel.vi** through **NI Example Finder**. Connect the Arduino to USB port, configure serial port settings of **LINX - PWM 1 Channel.vi**. Change PWM channel to “9” (**Figure5-26**).

Press the run button and input "0.5" to **Duty Cycle (0-1)**. You may find the fan starting to move.

Even if the fan does not start while **Duty Cycle** is set to "0.0~0.4", don’t worry, it’s not something wrong with your Arduino.

It’s a good habit to put default value or value range into label of Control, for example **Duty Cycle (0-1)**.

That can be effective for those who associate PWM with 0-255 to avoid inputting values more than 1.

However, it is not user-friendly and dangerous sometimes because using the Increment and Decrement Buttons may input out of range values such as "1.5" or "-0.5".

In such cases, change Maximum, Minimum and Increment setting in **Data Entry** tab (**Figure 5-27**)in the property as shown in **Figure 5-28**.

A block diagram looks similar to **Figure 5-29**. Now you can know that you should input value "0-1" to **PWM Set Duty Cycle.vi**.

Then, control fan speed with the tact switch by customizing **PushCounter.vi** which counts up value with the tact switch.

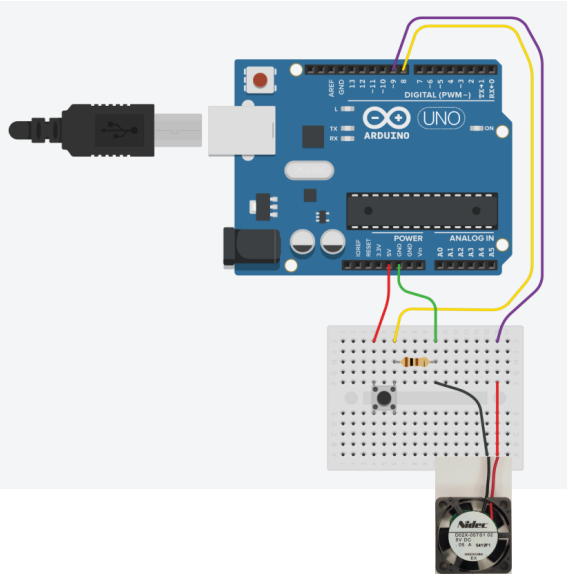
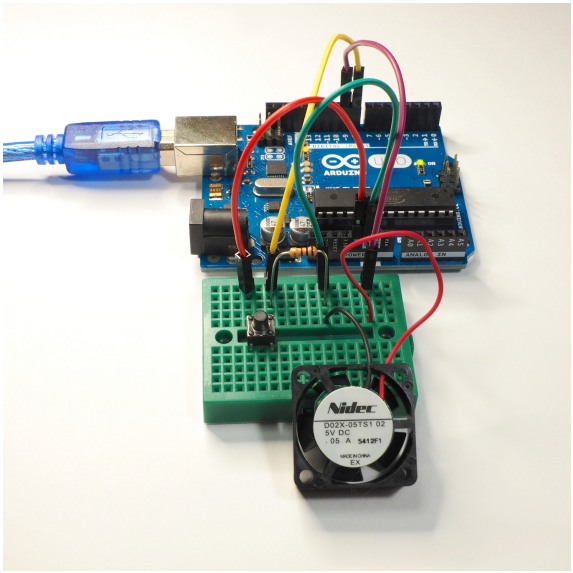


Figure 5-25 Wiring diagram



Picture 5-3 Placement and wiring of parts

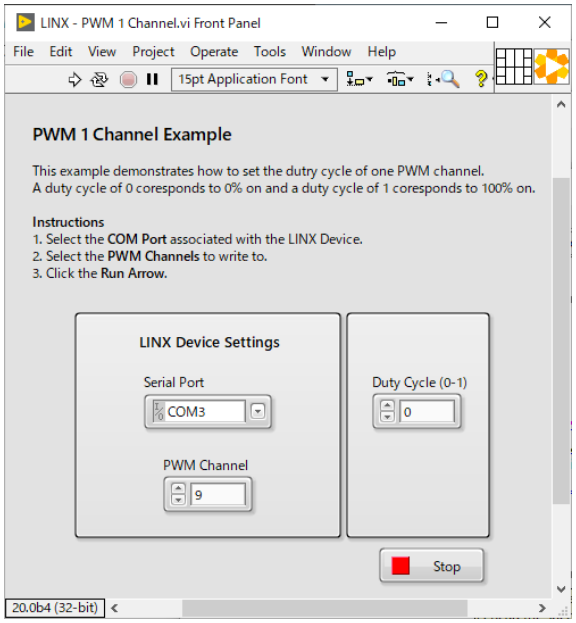


Figure 5-26 LINX - PWM 1 Channel.vi

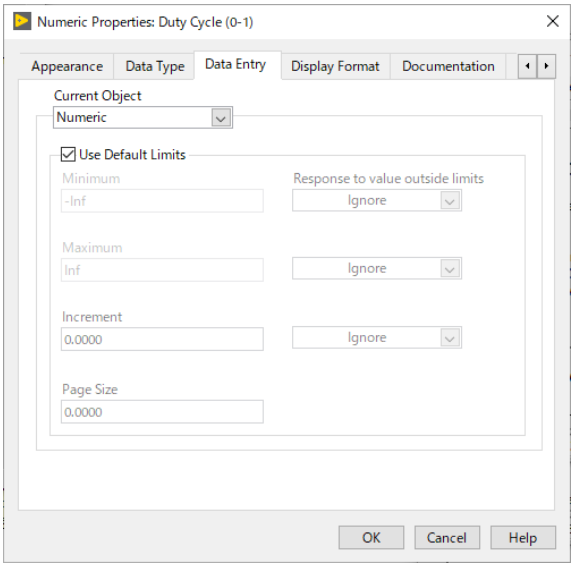


Figure 5-27 Numeric properties dialog box

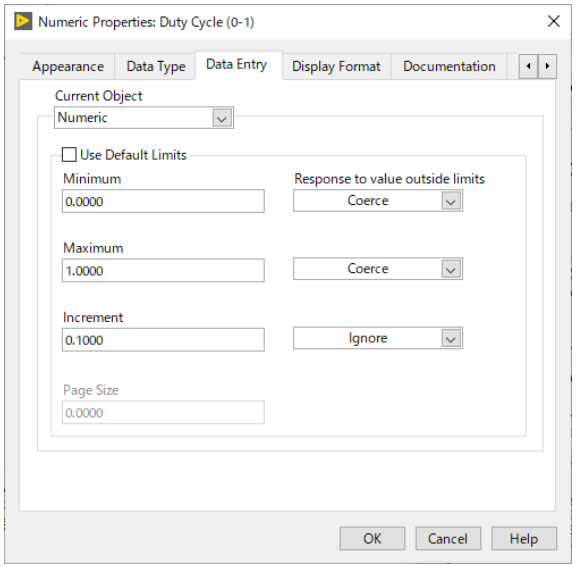


Figure 5-28 Set data entry

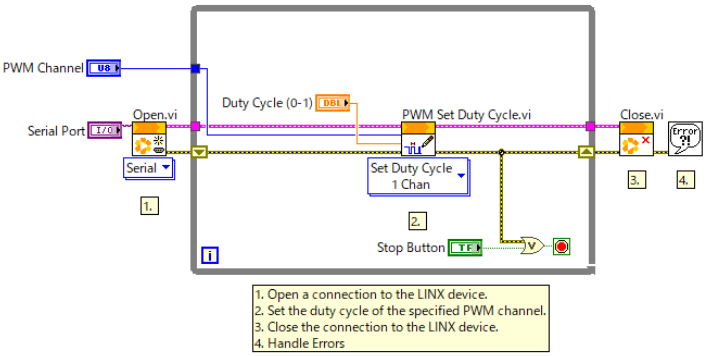


Figure 5-29 LINX - PWM 1 Channel.vi

You may see that a block diagram of **PushCounter.vi** is similar to **Figure 5-30**.

Multiply the counter value by "0.1" and connect it to input terminal of **PWM Set Duty Cycle.vi**.

The Functions palette of LINX can be found in **MakerHub > LINX** (**Figure 5-31**). Digital I/O, Analog I/O and PWM functions are in a **Peripherals** folder. **PWM Set Duty Cycle.vi** can be found in directory shown in **Figure 5-32**.

Now the block diagram looks like **Figure 5-33**

When the counter value gets increased, fan's speed gets faster. And the fan stops with the counter value 0. Although it seems working well, it's still bothersome to stop the fan. It can be stopped only after pressing button till the counter value gets 10.

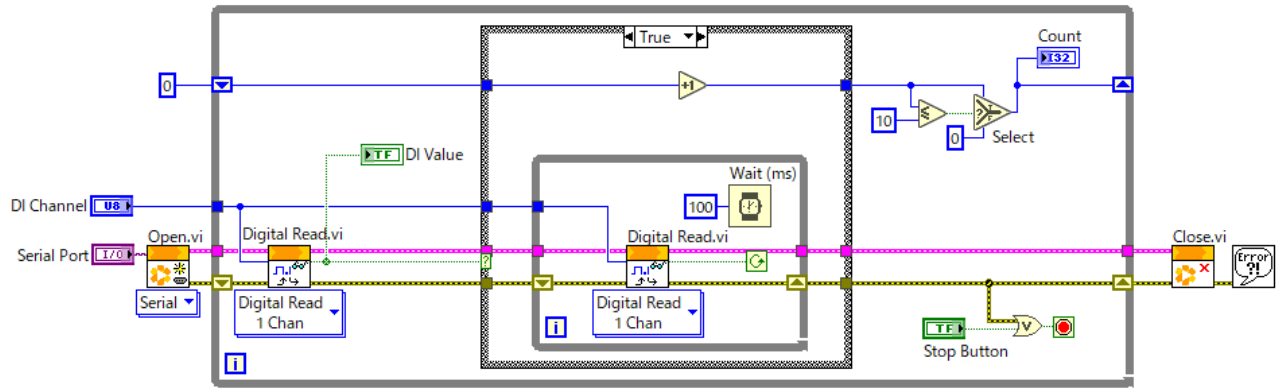


Figure 5-30 PushCounter.vi

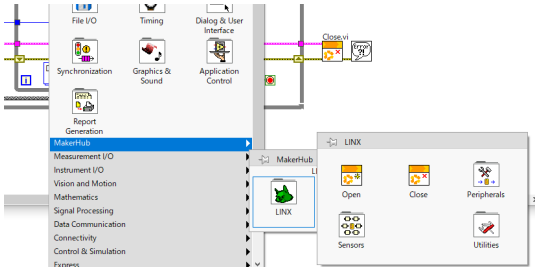


Figure 5-31 Functions palette of LINX

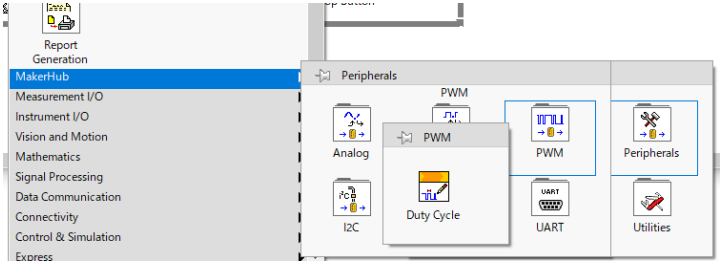


Figure 5-32 PWM Set Duty Cycle.vi

FanControlWithLongPushStop.vi of **Figure 5-34** is a program that has been customized to be able to stop when the tact switch is held down for 2 seconds.

Tick count (ms) measures time while the tact switch is held down. The program outputs True if it's less than 2 seconds. When ["≦10 counts value" and "≦2 seconds"] it uses the counter value as it is, but make it back to 0 in other case.

Make wiring simple by binding channel numbers of the tact switch and the fan into a cluster named **Hardware Setting**. (**Figure 5-35**)

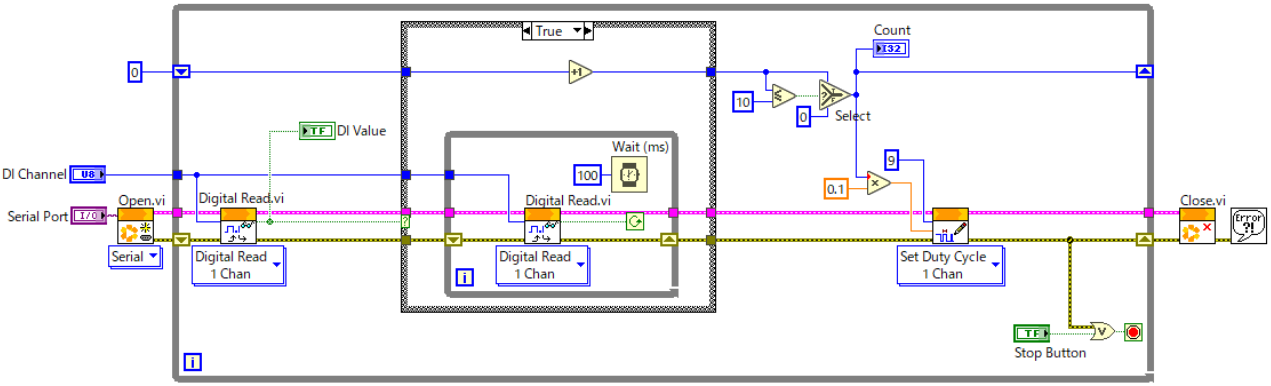


Figure 5-33 FanControl.vi

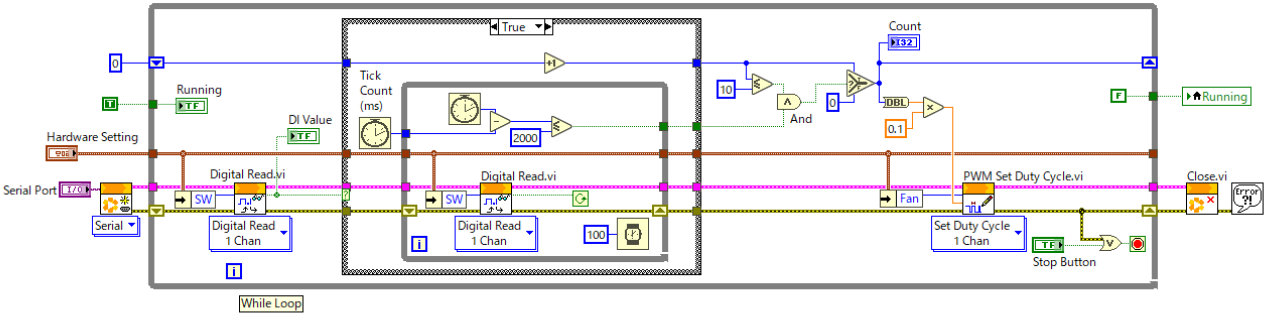


Figure 5-34 FanControlWithLongPushStop.vi

Because **FanControlWithLongPushStop.vi** stops after the tact switch is held down for 2 seconds and "released", let's customize the program so that it can stop without releasing the tact switch but only holding the switch for 2 secounds.

Although **FanControlWithLongPushImmediateStop.vi** (**Figure 5-36**) seems working well because the fan stops anyway, there is only one "count".

Please customize it to work correctly by inserting While loop to wait for **Digital Read.vi** outputting False after execution of **PWM Set Duty Cycle.vi** is completed.

In the end, some Notes for you. Functions of LINX have cluster terminals named **LINX Resource** on its top as input/output.

You can not connect **LINX Resource** wires to more than 2 LINX functions by making wire branch since LINX works by sending commands and receiving response with specific Arduino.

Even though an error may not occur sometimes, place functions by connecting each other with daisy-chain.

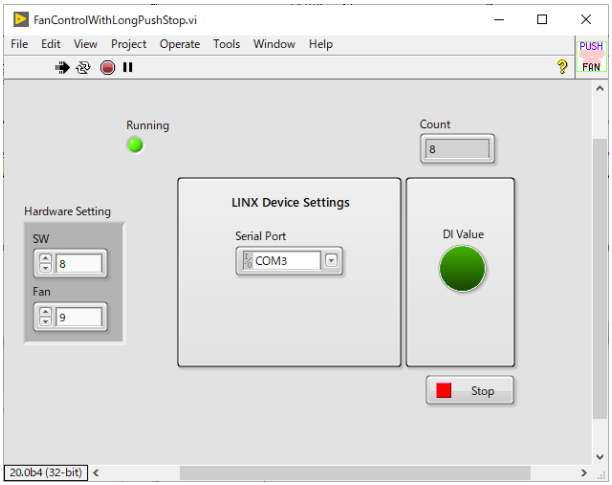


Figure 5-35 Front panel of FanControlWithLongPushStop.vi

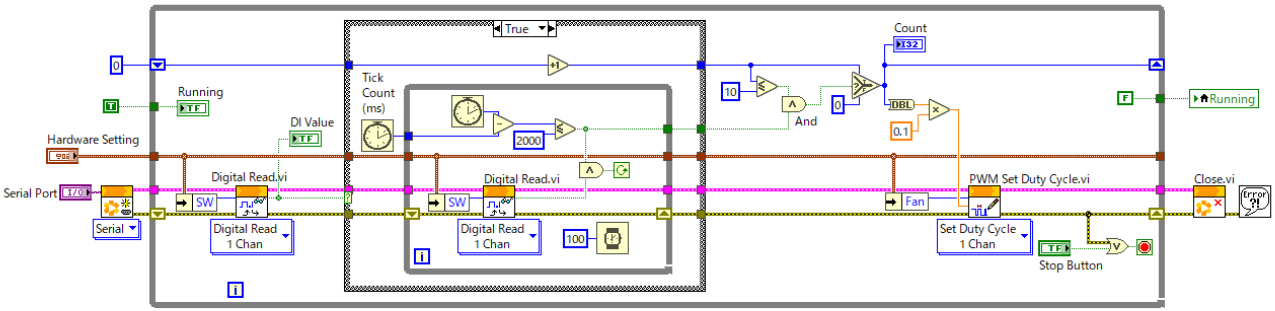


Figure 5-36 FanControlWithLongPushImmediateStop.vi

5.6 Change the Operation of the Switch

How was the customizing **FanControlWithLongPushImmediateStop.vi**?
Didn't your block diagram stick out of your display?
By the way, Boolean Controls in LabVIEW has 6 types of mechanical actions as shown in **Figure 5-37**.

Behavior of **PushCounter.vi** which you programed in the section **5.4 Make a Switch Counter** was that "counting up when the tact switch is pressed and then released". This behavior is close to that of **Latch When Released** in 6 types of mechanical actions.

We'd like to introduce a program which we implemented the switch behavior of **Latch When Released** programmatically.

Block diagrams of **Push Counter(LatchWhenReleased).vi** (**Figure 5-38**) and **Fan Control with Long Push Stop(LatchWhenReleased).vi** (**Figure 5-39**, **Figure 5-40**) were quickly coded by a professional, so please try to decipher those.

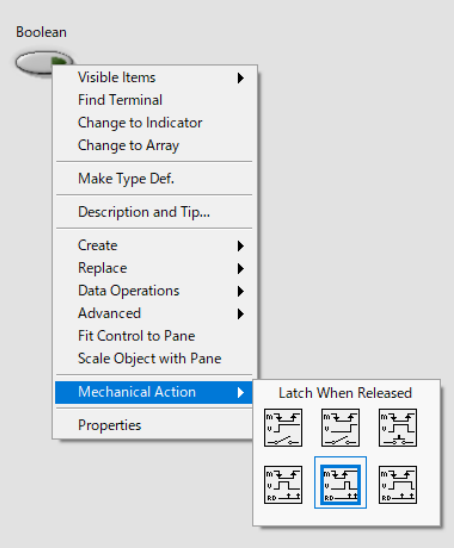


Figure 5-37 mechanical actions of Boolean Control

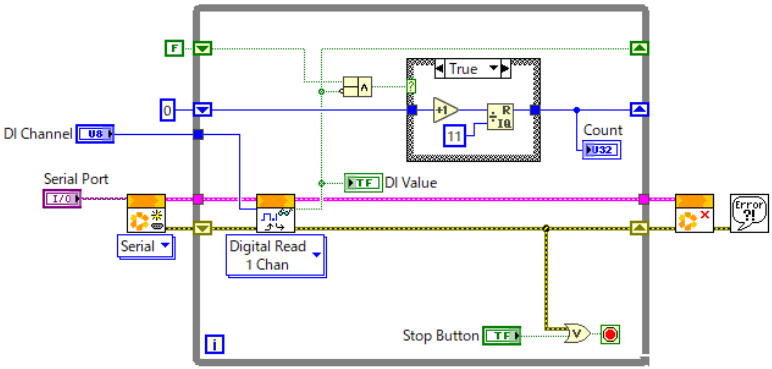


Figure 5-38 Block diagrams of "Push Counter(LatchWhenReleased).vi"

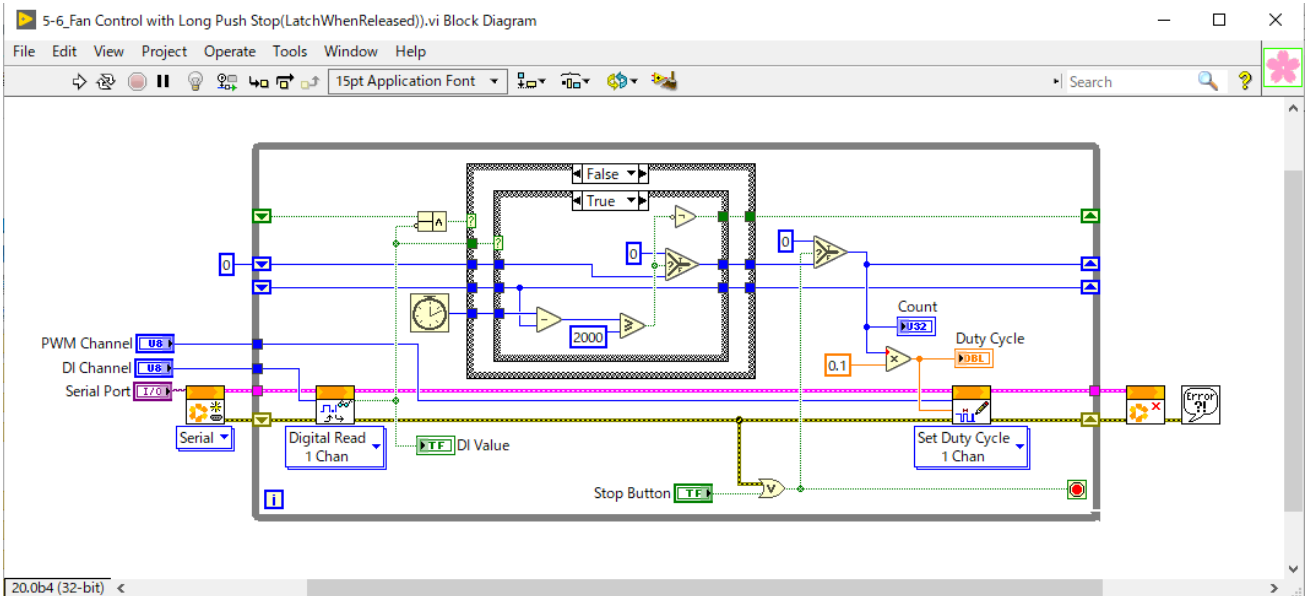


Figure 5-39 ”Fan Control with Long Push Stop(LatchWhenReleased).vi”（False & True Case）

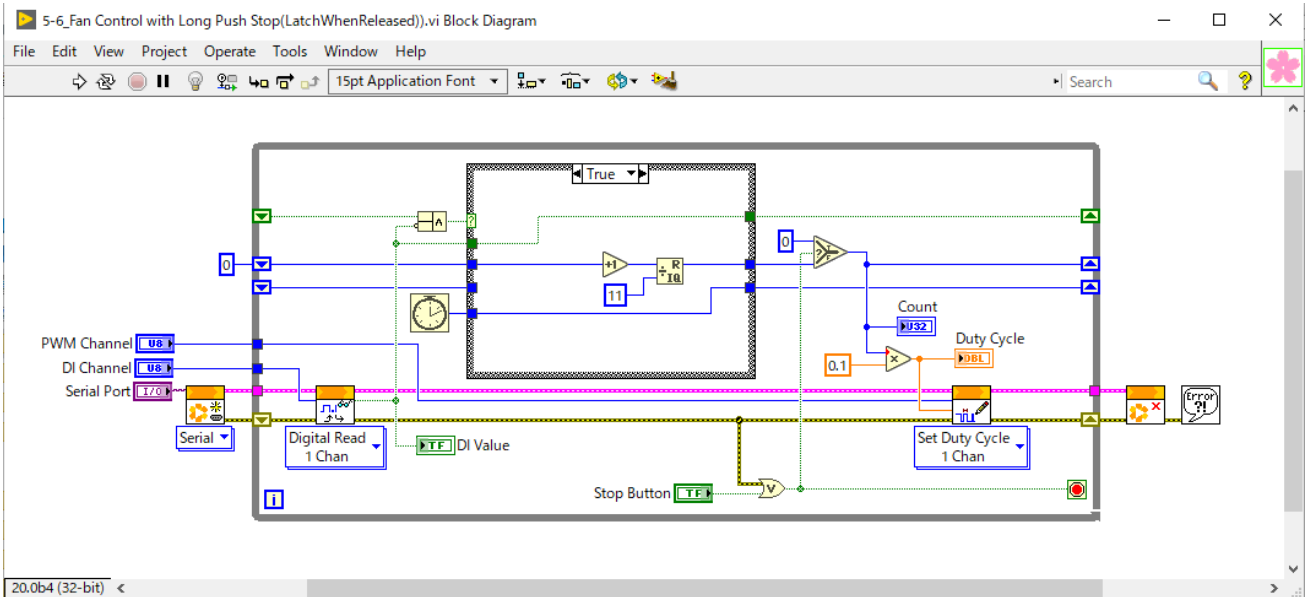


Figure 5-40 ”Fan Control with Long Push Stop(LatchWhenReleased).vi”（True Case）

Article 5

LabVIEW NXG and Hardware

Data Acquisition Device (**DAQ** for short) is the hardware that can measure voltage and input/output digital signal. Used by connected to PC with USB and so on. In LabVIEW NXG, there is a big change to DAQ programing.It is necessary to configure MAX/min of voltage range and a number of times to measure voltage for 1 second in order to use DAQ, which is checked after completing most of program in almost all cases.

In LabVIEW NXG, you can check and configure these settings of DAQ without programming. (**Figure C5-1**). Moreover, you can use a program with a diagram which is generated automatically based on these settings.

LabVIEW was originally developed in order to equip non-programmer engineers and scientists with skills to make a program for measurement and controlling.LabVIEW NXG has strongly succeeded this purpose and can do more things without programing compared to LabVIEW.

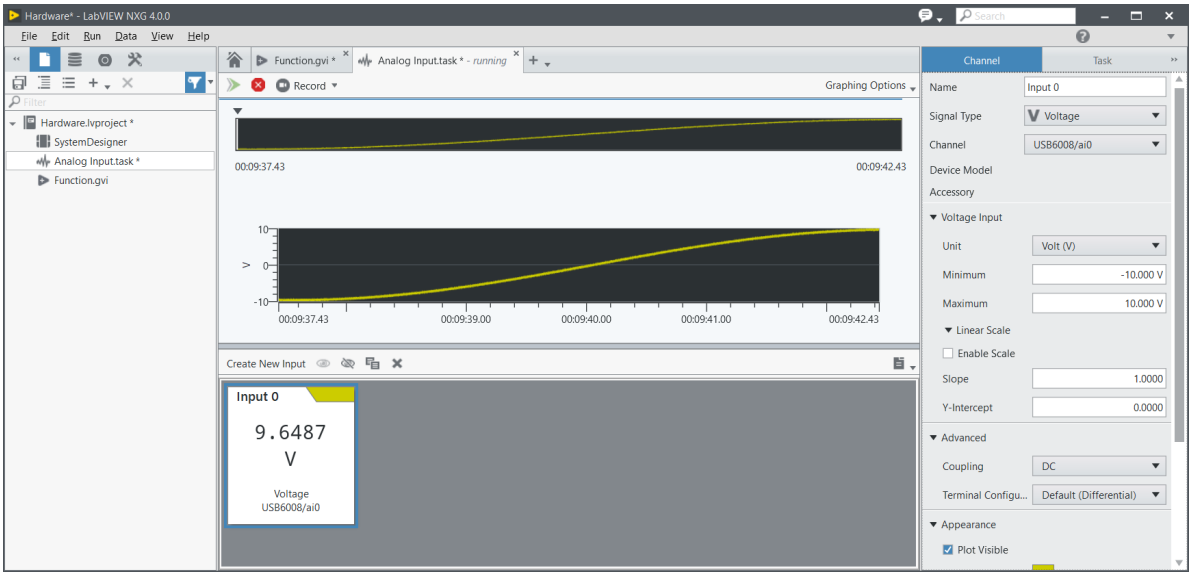


Figure C5-1 Hardware project window of LabVIEW NXG

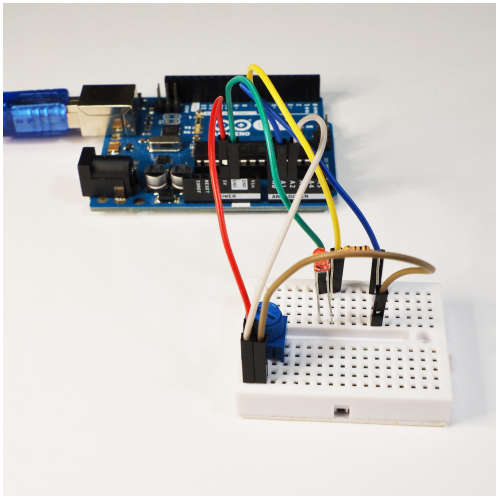
Chapter 6

Investigating LED Properties

We will use the 3 analog input channels of the Arduino to measure the voltage-current properties of the LED (**Picture 6-1**). We will apply the measured data into formula to acquire the regression curve. The formula is a little advanced, but this is an example of the data analysis.
[Keywords] X-Y Graph, 1D Array Sort, Logarithmic Transformation, Linear Regression
[Parts used] 1 Arduino UNO, 1 breadboard, 1 LED, 1 resistor (100Ω), 1 Variable Resistor (10kΩ), 6 wires

6.1 Assembling the Experimental Circuit for LED Voltage-Current Properties

When lighting the LED, we must place a current limiting resistor to forbid too much current flow into the LED. The current flow into the LED increases exponentially when the voltage applied increases beyond the **forward voltage (Vf)**, and affects the LED luminance greatly. Since it is more difficult to control the luminance with voltage than current, we include resistor into the circuit to alter the current. In this chapter we will observe the exponential growth of the current. In order to conduct the experiment, we will need an LED, a 100Ω resistor, a 10kΩ variable resistor, 6 wires, a breadboard, and an Arduino UNO. Further explanations on the variable resistor and the experiment circuit is on the **6. Appendix**. Remove the Arduino from PC's USB port and wire it as shown in **Figure 6-1**.



Picture 6-1 Measuring V-I properties of LED

Connect the **5V pin** of the Arduino to the 10kΩ variable resistor, 100Ω resistor, and the LED in series and the other end of

the LED to the Arduino **ground pin**. Note that the variable resistor has three legs. Turn the knob to the leftmost position and place the variable resistor on the table so that the tip of the arrow is facing down left. Looking from the top, connect the left pin to **5V** and center pin to 100Ω resistor. Since LED has polarity, connect the shorter leg to **ground**. Current will flow from 5V, variable resistor (0kΩ to 10kΩ), 100Ω, LED, and to ground. Changing the value of the variable resistor will alter the voltage applied and current flow into the LED. We will measure the current and the voltage with Arduino's analog input. Use a new wire to connect the point between 100Ω resistor and LED to **A0**. This is the voltage applied to the LED. Use another wire to connect the point between the 10kΩ variable resistor and the 100Ω resistor to A1. Subtracting **A0** voltage from **A1** voltage to get the voltage applied onto the 100Ω resistor. Using Ohm's Law (I=V/R) we find the current flowing on the circuit, i.e. the current flowing in the LED. To observe that the 5V power supply is stable, connect a wire from the point between 5V and 10kΩ variable resistor to **A2**. Check the circuit once again to make sure everything is in place. Then connect the Arduino to PC's USB port. When connected, the LED will light up, and when you turn the knob of the variable resistor to the right, the LED will light dimmer. Now we are ready for the experiment.

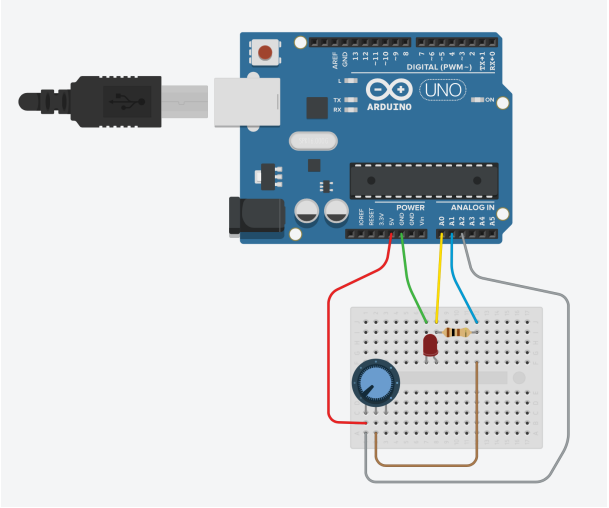


Figure 6-1 Circuit diagram of experimental circuit

6.2 Measuring LED Voltage and Current

Using the three analog input channels of the Arduino, we will measure the LED voltage-current properties with LINX. Open **LINX – Analog Read N Channels.vi** from NI Example Finder (**Figure 6-2**). Click the "▽" icon on the right side of the Serial Port control and select the COM number that the Arduino is connected to. If nothing is shown even though your

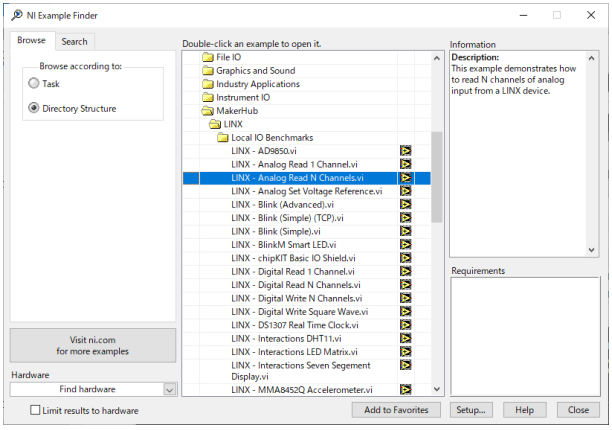


Figure 6-2 NI Example Finder

Arduino is connected to the USB port, click the "Refresh" on the bottom of the pull-down menu. We will input the analog input numbers into the AI Channel(s) array. Since we are using **A0** to **A2**, we input 0, 1, and 2. Press the Run button and turn the variable resistor knob left and right and you will see the Waveform Chart "Analog Data" result change (**Figure 6-3**). **A0** should be around 2V, **A1** should be between 2V to 5V depending on the position of the variable resistor, and **A2** should be 5V. You may have noticed that there is

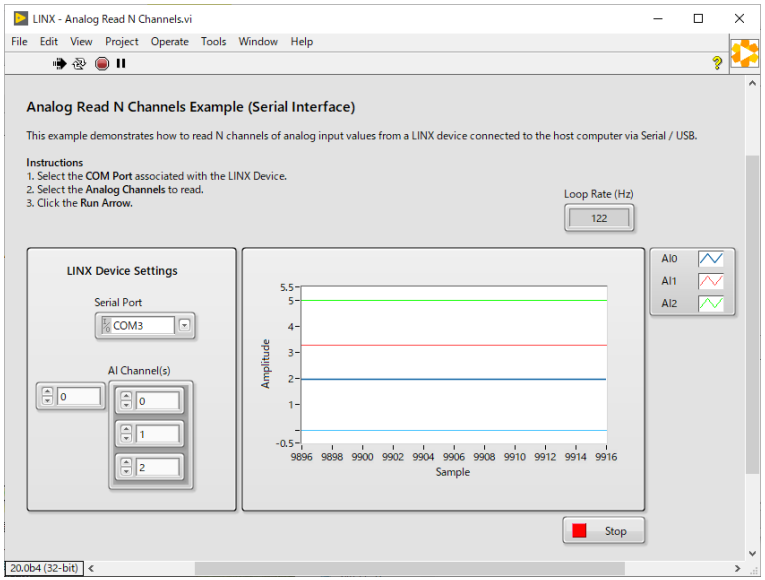


Figure 6-3 LINX – Analog Read N Channels.vi

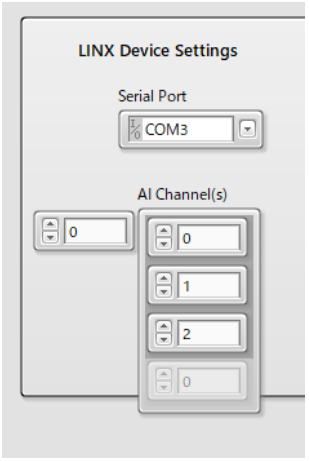


Figure 6-4 Elements of Channels array

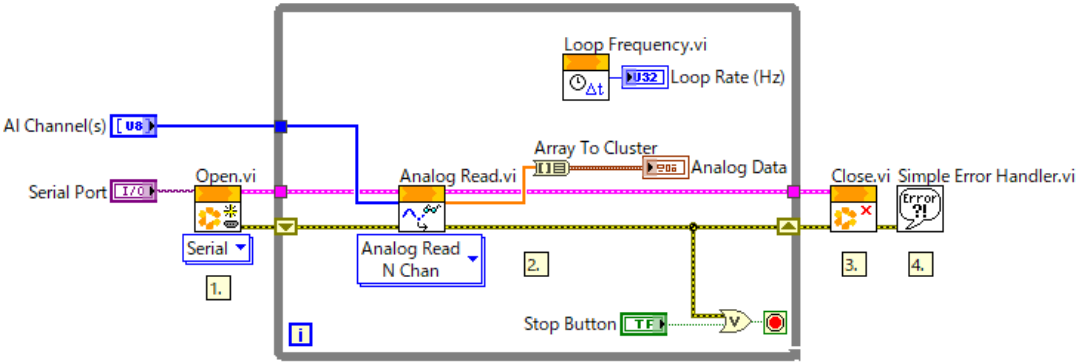


Figure 6-5 Block diagram

data around 0V as well. Hover the cursor over the Plot Legend and drag down the little blue square and you will see "Plot 3" below "A2." Do the same for "AI Channel(s)" array as in **Figure 6-4** and you will see there are only 3 items in the array and only 3 channels are set. From the top of LabVIEW, select "Window > Show Block Diagram" (**Figure 6-5**) to open the block diagram. Right-click on the block diagram to open the Functions Palette and hover to Array Palette. Left-click on "Array Size" and place it near **Analog Read.vi** and connect it to determine the number of items in the array (**Figure 6-6**). Run the VI and observe the "size(s)" indicator, and you see that it is 3 as expected. Double click on "Array To Cluster" in between **Analog Read.vi** and waveform chart, and you will see a small window "Cluster Size" displayed (**Figure 6-7**). Input "3" and click the OK button. Now run the VI again to see that the fourth plot has disappeared. Study the help file on waveform chart to notice that we need to connect the data as cluster when we connect multiple plots onto the same waveform chart.

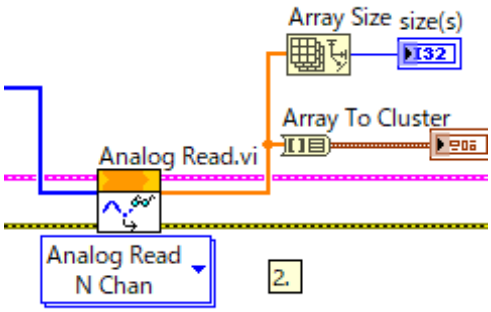


Figure 6-6 Size of the output array

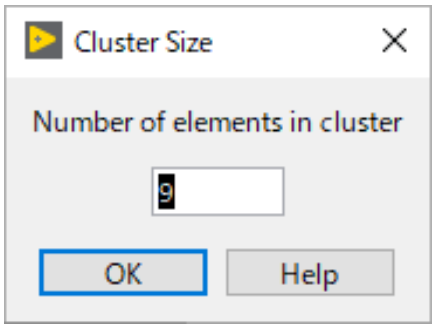


Figure 6-7 Number of elements in cluster

Save **LINX – Analog Read N Channels.vi** as **LED VI Curve.vi**. We want to delete the text on the front panel but cannot since it is locked. Move the mouse cursor to the top left of the explanation text and drag the mouse over the rectangle to select the textbox. Click on the "Reorder" icon on the toolbar and select "Unlock" to delete the textbox (**Figure 6-8**). Using "Index Array" function in the Array function palette, extract item 0, 1, and 2 from **Analog Ready.vi**. Item 0 is the voltage applied to the LED. Value of Item 0 subtracted from Item 1 is the voltage applied to 100Ω resistor, and divide this voltage by 100Ω to get the current flowing in this circuit (**Figure 6-9**). We are able to calculate the voltage and current applied to LED

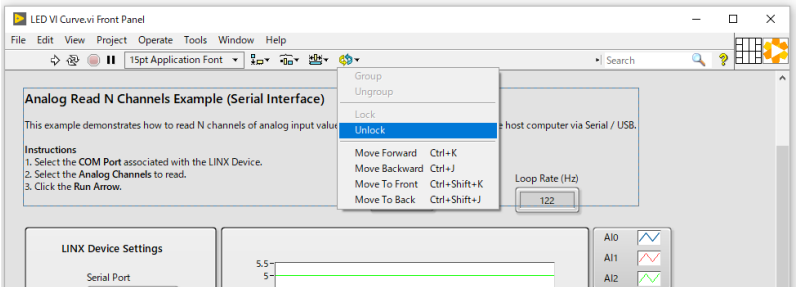


Figure 6-8 Unlock the description text

this way. Connect numeric indicators to the LED voltage and current to observe that when the voltage becomes higher and current becomes greater, the LED luminance is greater, and when the voltage becomes lower and current is less, the luminance is lower (**Figure 6-10**).

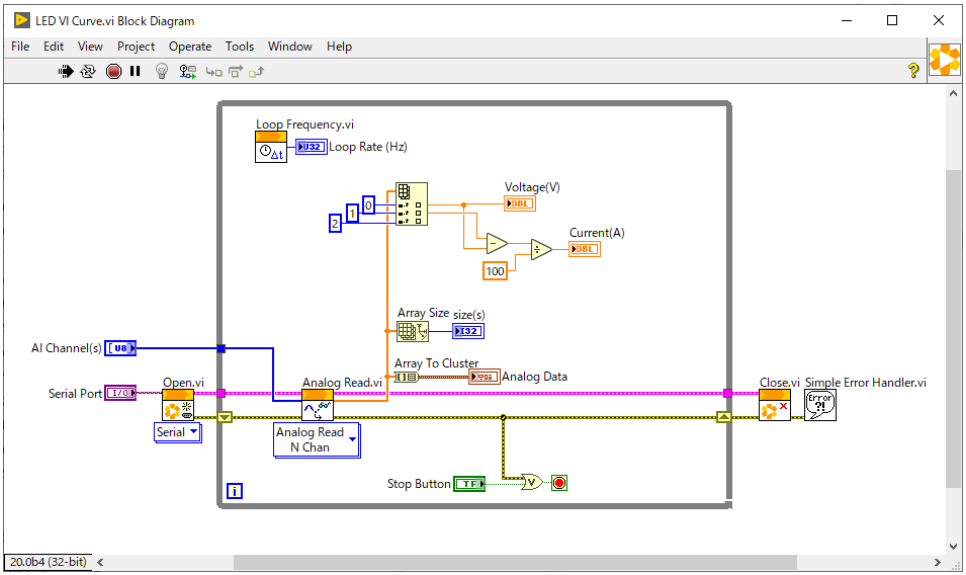


Figure 6-9 Block diagram to display voltage and current

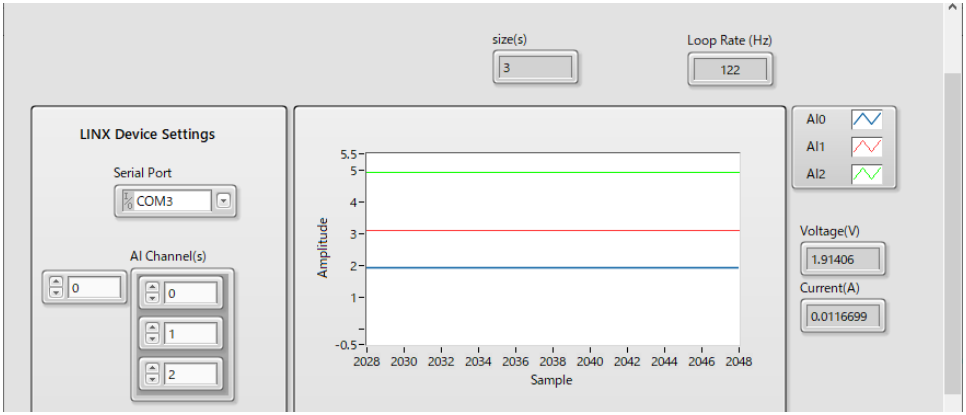


Figure 6-10 Front panel displaying voltage and current

6.3 The Program to Display LED V-I Property Curve

Let us plot the properties in the X-Y Graph with voltage in X axis and current in Y axis. You may already have the knowledge to write this VI on your own, but we have provided an example VI. Open **6-2_LED VI Curve.vi** in the program folder. When the "Add data" button is pressed, voltage and current measured are added to voltage array and current array and are plotted onto the X-Y Graph. Set the "Serial Port" and AI Channel(s) and press the Run button. Change the voltage and current with variable resistor and press the "Add data" button accordingly. As data is accumulated you will see a graph increasing to the top right (**Figure 6-11**). Make sure to fill the gap by adding data while controlling the variable resistor. The regression analysis button will be explained in section 6.4. This example VI does not have the function to save the data, so data will become unusable after VI is stopped. Therefore, do not stop the VI just yet. Open the block diagram and observe if the structure is as you have expected (**Figure 6-12**). Take a closer look at the case structure that works when the "Add data" button is clicked. Here we are sorting the voltage from low to high. This is included because voltage and current must be paired to correctly analyze the data. For both voltage and current, we add the data into their respective array and then create another array with item as clusters

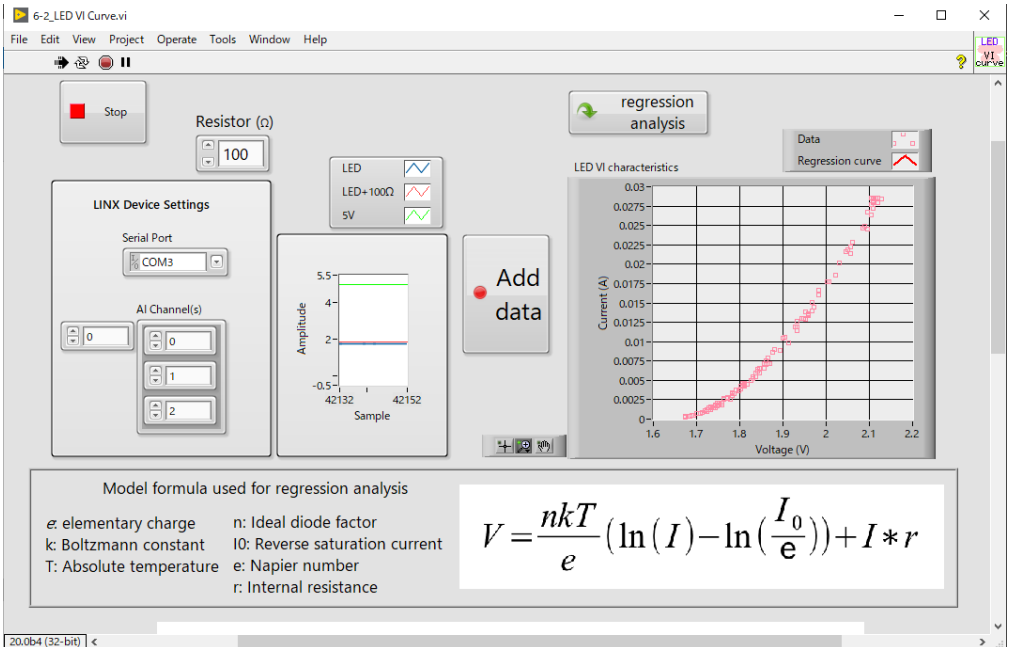


Figure 6-11 Adding data to X-Y graph

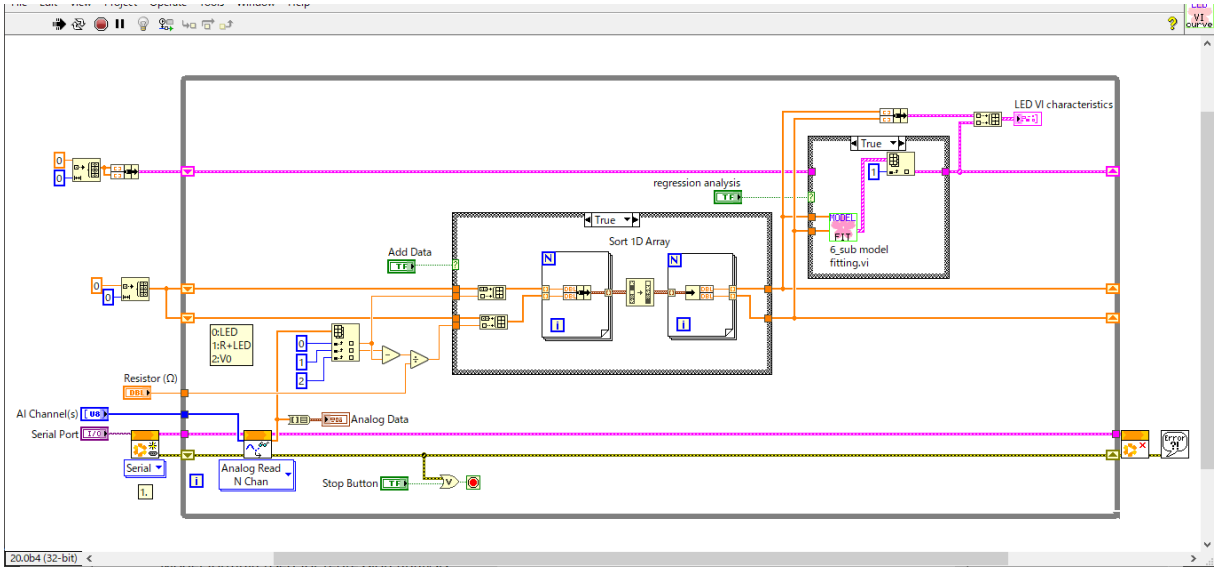


Figure 6-12 Block diagram

with a single pair of voltage and current. Using “Sort 1D Array” we sort this array of clusters. Since this function sorts the array based on the 0th item of the cluster, notice that voltage is set as the 0th item. After they are sorted, the cluster is separated again and are put back into voltage and current arrays. You can also use “Sort 2D Array” function if available. To display the data onto the X-Y Graph, we assemble the 1D array data for X-axis and 1D array for Y-axis as a cluster. To display multiple plots onto the X-Y Graph, we create an array of this cluster.

6.4 Regression Analysis of LED V-I Property Data

By modifying the voltage while adding the data, we can observe that current increases when the voltage is increased with slight disparity. **LED** is short for **Light Emitting Diode**, and it is a type of a diode. From here on we will explain how to display the LED voltage and current data as approximation equation. As we acquired the data repeatedly while changing the resistance, you may have noticed that voltage and current pair has shifted position for similar resistance. You may have expected the data to align in the same position as data points increase, but sorry to disappoint you. There are many potential reasons for this misalignment. You may think that voltage and current are read exactly at the same time, but in reality they are read one by one, so there is a time difference. Turning the knob of the variable resistor too quickly would also cause the

voltage and current data difference. Temperature difference of the LED could be another reason. As you can see, the difference is yielded by improperly controlled factors. First line of **Figure 6-13** is the model equation for voltage-current property of the diode PN Junction. Looks complicated; it is exponential function you learn in high school. Simply put, it shows that changing voltage V grows current exponentially. Variables are explained in **Figure 6-14**. Elementary charge and Napier's constant are both "e," but to differentiate, elementary charge is written in italics. A model function simplifies natural behavior and display the effect as a mathematical function.

When the current flow into an LED becomes greater we cannot ignore the voltage drop caused by **internal resistance (r)**. The first equation on **Figure 6-15** represents this and **(– I * r)** is the voltage drop by inner resistance. If you look at this equation closely, **current (I)** is used both on the left and right hand side of the equation, so we cannot solve for current just by substituting in **voltage (V)**. The third equation in **Figure 6-15** solved for voltage. You may feel uneasy about how voltage is expressed as a function of current, but for convenience we will continue the analysis by placing current for x-axis and voltage for y-axis.

The procedure of comparing between the experiment data with rounds of error and a model equation to acquire probable parameters and substituting them in the model equation to express the experiment data is called **regression analysis**. For this experiment, we can say that we are deriving the diode **ideality factor (n)**, **reverse saturation current (I_0)**, and **internal resistance (r)** from the data you acquired by pressing the "Add data" button multiple times. Once n, I_0, and r are determined, we are able to estimate the relationship between current and voltage for an LED.

Press the "regression analysis" button to display the regression curve on the X-Y graph (**Figure 6-16**). Bold line is the regression

$$I = I_0 \exp\left(\frac{e}{nkT} V - 1\right)$$

$$I = \frac{I_0}{e} \exp\left(\frac{e}{nkT} V\right)$$

$$V = \frac{nkT}{e}\left(\ln(I) - \ln\left(\frac{I_0}{e}\right)\right)$$

Figure 6-13 Model equation without inner resistance in consideration

e : elementary charge
k : Boltzmann constant
T : absolute temperature
n : ideal diode factor
I_0 : Reverse saturation current
e : Napier's constant

Figure 6-14 Constants

$$I = I_0 \exp\left(\frac{e}{nkT}(V - I*r) - 1\right)$$

$$I = \frac{I_0}{e} \exp\left(\frac{e}{nkT}(V - I*r)\right)$$

$$V = \frac{nkT}{e}\left(\ln(I) - \ln\left(\frac{I_0}{e}\right)\right) + I*r$$

Figure 6-15 Model equation with inner resistance in consideration

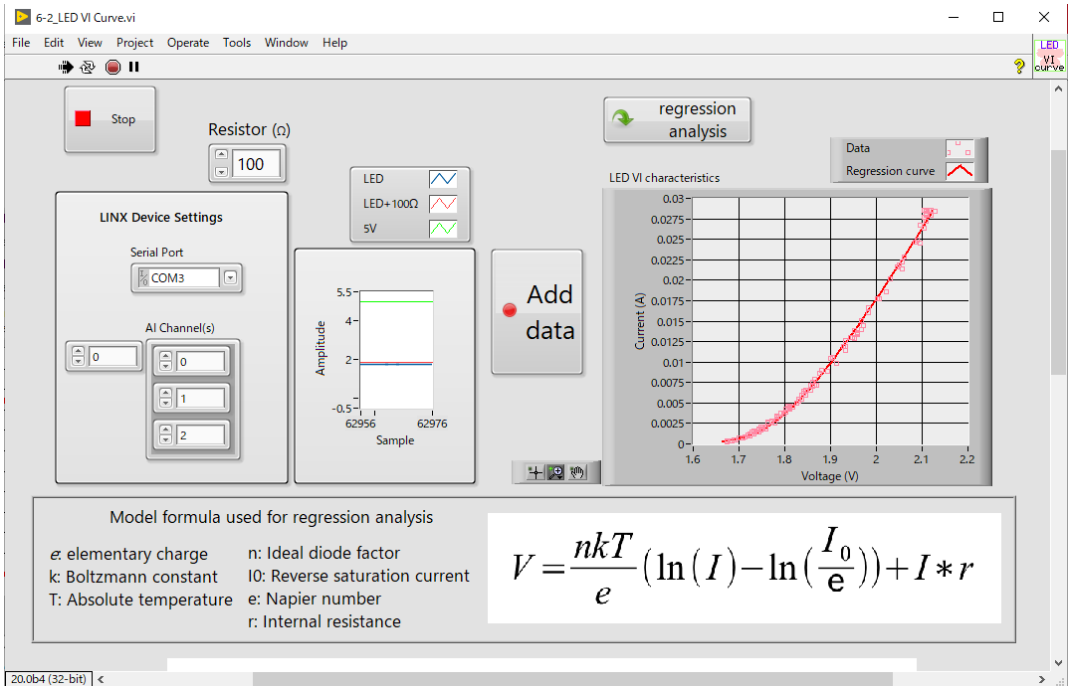


Figure 6-16 Displaying the regression curve

curve acquired from various data points. Let us open the sub VI that performs the regression analysis. Open the block diagram of **6-2_LED VI Curve.vi** and double click **6_sub model fitting vi** located in the true case of case structure connected to "regression analysis" button. Front Panel similar to **Figure 6-17** should be displayed, but there should be no values in controls and indicators. You may have various front panels and block diagrams open on your display, Open the front panel of **6-2_LED VI Curve.vi** and press "regression analysis" button once again. Now values will be inputted onto controls and data will be displayed on the three graphs. **The rightmost graph** is the final result of the regression, and this will be displayed on the front panel of **6-2_LED VI Curve.vi.** Open the block diagram of **6_sub model fitting.vi** and it should be displayed like **Figure 6-18**. **Figure 6-19** is the equation extracted from third row of **Figure 6-15**. The **ln()** in the equation is **natural logarithm**. The right hand side of the equation is a combination of the linear function of ln(I) and the linear function of I.

The leftmost graph on **Figure 6-17** displays the natural logarithm of measured current data on the x-axis and voltage on the y-axis. Look at the legend and you will see that there are four types of data: Small Current

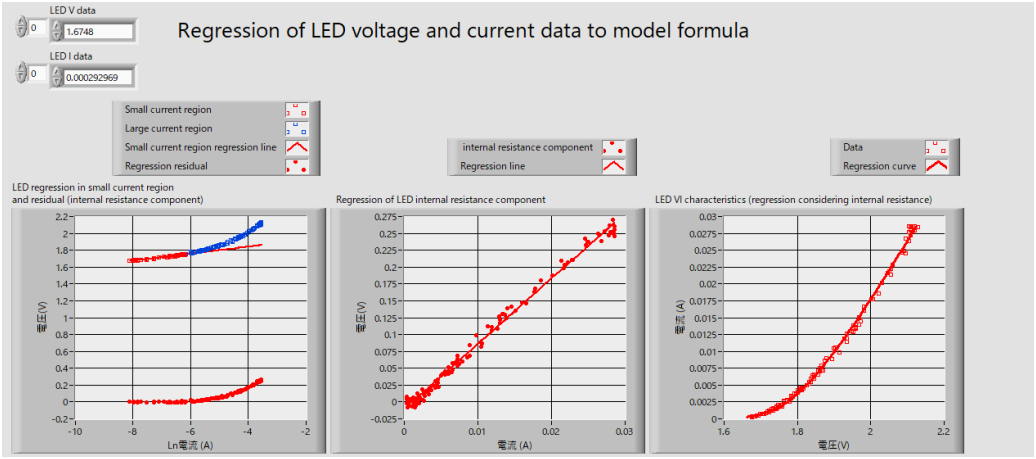


Figure 6-17 Sub VI for regression analysis

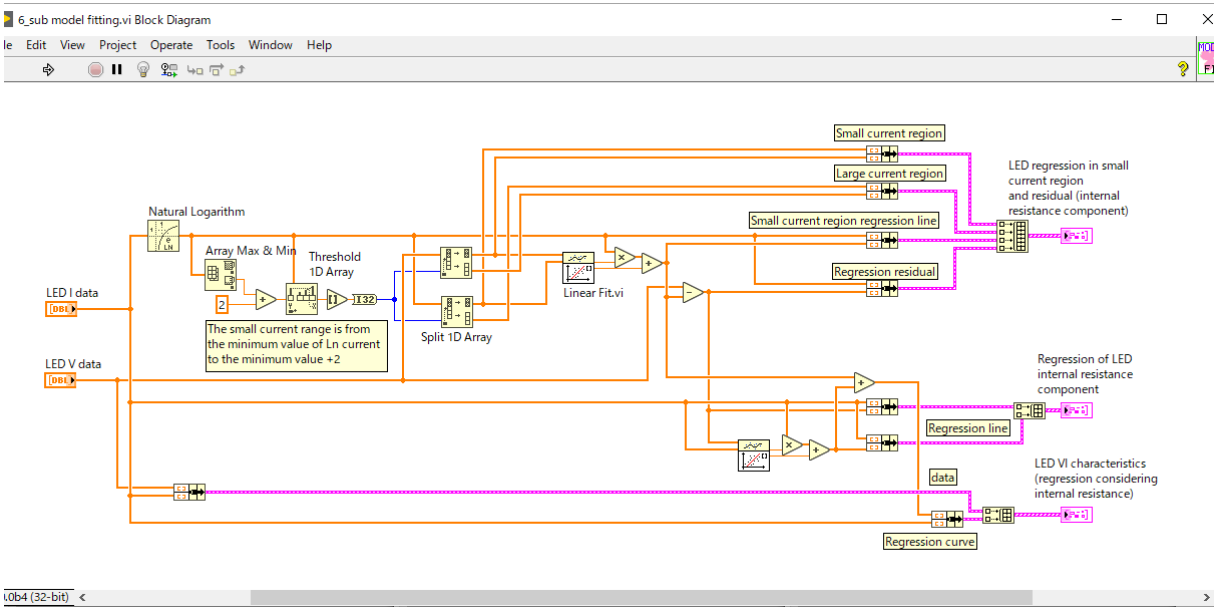


Figure 6-18 Block diagram

Figure 6-19 Model equation with inner resistance in consideration

$$V = \frac{nkT}{e}\left(\ln(I) - \ln\left(\frac{I_0}{e}\right)\right) + I * r$$

Region, Large Current Region, Small Current Region Regression Line, and Regression Residual. This corresponds to the block diagram where four data wires are connected to the graph on the top. The method to separate the "Small Current Region" and "Large Current Region" will be explained later. **Small Current Region** increases linearly while Large Current Region curves up gradually. Look at the equation on **Figure 6-19** and remember that the right hand side is the combination of linear function of ln(I) and linear function of I. The reason why Small Current Region changes linearly is because the **linear function of ln(I)** is prominently visible. "Small Current Region Regression Line" is the linear regression of Small Current Region displayed with red solid line. Subtracting "Small Current Region Regression Line" from the experiment voltage data and graphing the result, we get "**Regression Residual**," which is the **linear function of I**. The reason why "Regression Residual" is curved despite it being linear function is because the x-axis is the natural logarithm of current. The **graph on the center** shows current as x-axis and voltage as y-axis, and as shown on the legend, it displays the inner resistance element and its linear regression. Inner resistance element is the Regression Residual on the leftmost graph. The **rightmost graph** shows voltage as x-axis and current as y-axis and displays the measured data with the summation of two regression data plotted as a regression curve.

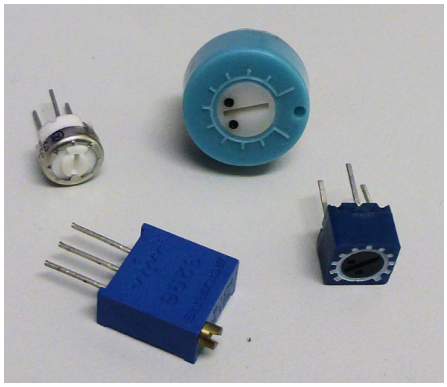
Linear Fit.vi function is used for regression analysis. This function allows you to set analysis parameters, so study the description in the help document. The split between "Small Current Region" and "Large Current Region" is done at the point where the graph's "Regression Residual" becomes greater. We have tested three types of LED and made the split at point "(the minimum of the natural log of current data) + 2" and were able to conduct regression without a problem, so we believe that readers should get similar results. The optimal point differs by the properties of an LED, so feel free to modify the point. The regression analysis done in this chapter is an experiment done at the university level, so some points may have been difficult to understand due to lack of explanation. The block diagram should be easy to ready, so feel free to investigate further.

6.Appendix　Additional Note on Variable Resistorsand the Experimental Circuit

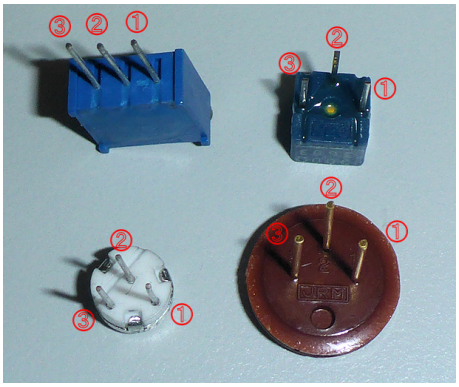
6.Appendix.1　About Variable Resistors (VR)

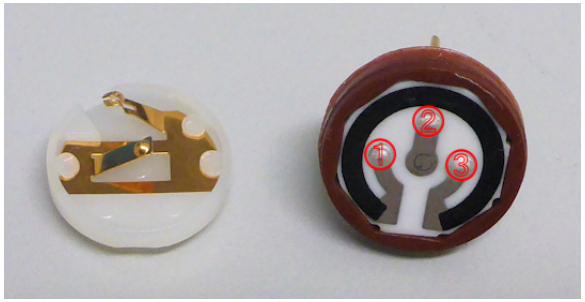
Variable resistors are also called potentiometers, and there are several types, but as long as they are 10kΩ, they do not have to be the same shape as the picture in the main text. Small screwdrivers can be used to turn the knob of VRs in **Picture 6-A1**. Red ①, ② and ③ in **Picture 6-A2** are pin numbers. If they are not engraved, the pin assignments can be inferred from the positions of the three leads.

The structure inside is almost identical, and when disassembled it looks like **Picture 6-A3** (disassembled VR with light blue casing). The flat and square one has a reduction gear and is capable of precise adjustment of 25 revolutions one way.

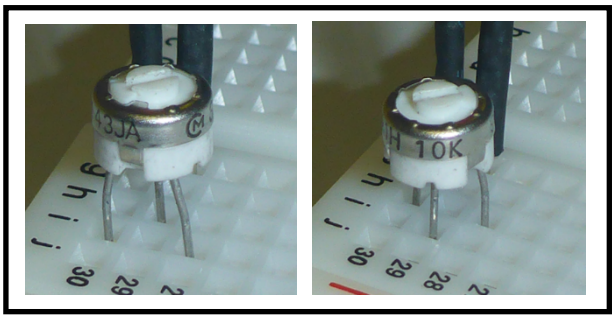


Picture 6-A1: Various types of VRs





Picture 6-A3: Inside a variable resistor



Picture 6-A4: How to insert into a breadboard

The black band across the No. 1 and No. 3 pins is the resistive element. The resistance of 10kΩ is the value between ① and ③.When you turn the knob, the gold spring rotates while making contact with pin 2 and the resistive element. Depending on the angle, the contact position to the resistive element changes, and the resistance value between 1-2 and 2-3 changes continuously.

6.Appendix.2 Knob Rotating Direction and Value of Resistance

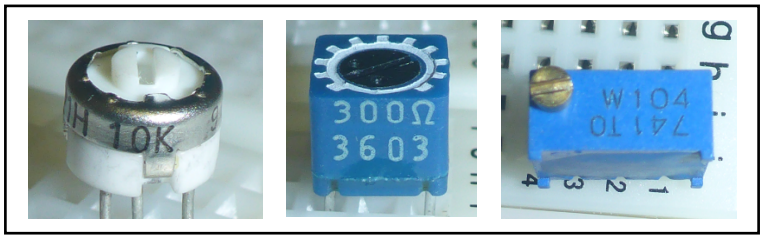
In general, when the knob is turned all the way to the right, the maximum value between 1-2 (in this case, 10kΩ) and the value between 2-3 is 0Ω. We can use this as a three-terminal voltage divider or a two-terminal resistor. The experiments in Chapter 6 use two terminals (between 1-2 or 1-3) because they are used as series current limiting resistors.

Depending on whether you use 1-2 or 1-3, the direction of turning the knob and the direction that brightens the LED will change. For **Picture 6-A4**, the left VR dims LED when turned right, and the right one brightens LED when turned right. If it was not what you expected, shift the terminals and try again. The direction will not affect the measurement even if it is reversed.

6.Appendix.3 Notation of Resistance Value

When reading the resistance values written on the parts, it is necessary to pay attention to the mix of notations. Picture 6-A5 shows 10kΩ and 300Ω from the left. These are written as 10K or 300 ohms, so it is easy to understand, but how many ohms do you think the right end is? The hint is 104.

The law is "two-digit value + one-digit exponent (power of 10)". In this case, 10×10^4 (four zeros followed by 10) makes 100000 = 100kΩ. For example, 472 is 4700 = 4.7kΩ. If there is no unit and the first digit is not 0, you can assume it follows this law. If it is simply written as 100, there is a possibility of 10Ω or 100Ω, so please measure it with a tester and confirm it.



Picture 6-A5: Notation of resistance value

6.Appendix.4 LED Polarity

The longer leg of the lead wire is the positive side of the LED. If you cut and align them when you put them on the breadboard, you will no longer know its sides, but try both sides. If the polarity is reversed, the current will not flow and the LED will not glow, so don't worry.

6.Appendix.5 Schematic of the Measurement System

You can understand further if you are able to read the schematic.

In this experiment, since variable resistor, resistor and LED are used as individual parts (called discrete device), it is easy to compare between the real circuit and the diagram (**Figure 6-A1**). It is difficult to view the

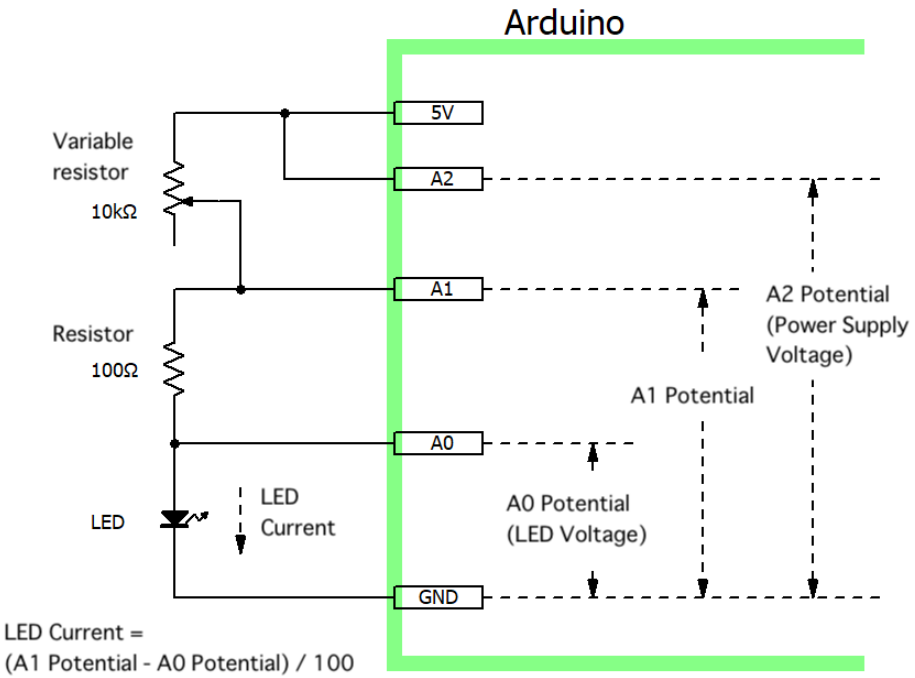


Fig. 6-A1 Experimental circuit diagram

inside of sophisticated devices such as IC, but an analogous circuit diagram can be used to illustrate its behavior.

The current flows from the 5V power supply terminal through the variable resistor - fixed resistor - LED to the GND (ground, overall reference) terminal. A0, A1, and A2 are terminals for voltage measurement. In reality, the current flows into A0 and A1, but the value is very small, so we ignore it.

What is important here is that all the analog inputs of the Arduino are measuring the potential difference from the GND reference. Therefore, the voltage at both ends of the 100Ω fixed resistance required to detect the LED current is obtained by subtracting the A1 and A0 potentials.

Article 6
Data Analysis with LabVIEW NXG

We are able to understand many things by analyzing the acquired data. Nowadays we can play our favorite music just by telling the smart speaker to play music. This is because it acquires our voice as data and performs analysis on it. There are also smartphone apps that displays jogging distance and the calories spent when we run with it, and this is done by analyzing the vibration put on the smartphone. “Data analysis” is crucial to understand our surroundings. LabVIEW NXG contains many functions for analysis, so you can assume it is able to perform most types of analysis. You have seen from the previous example VIs that some level of programming must be done before the analysis, but LabVIEW NXG enables us to perform analysis during programming. This allows us to check if the functions we are using are correct and to acquire the result while programming, enhancing convenience.

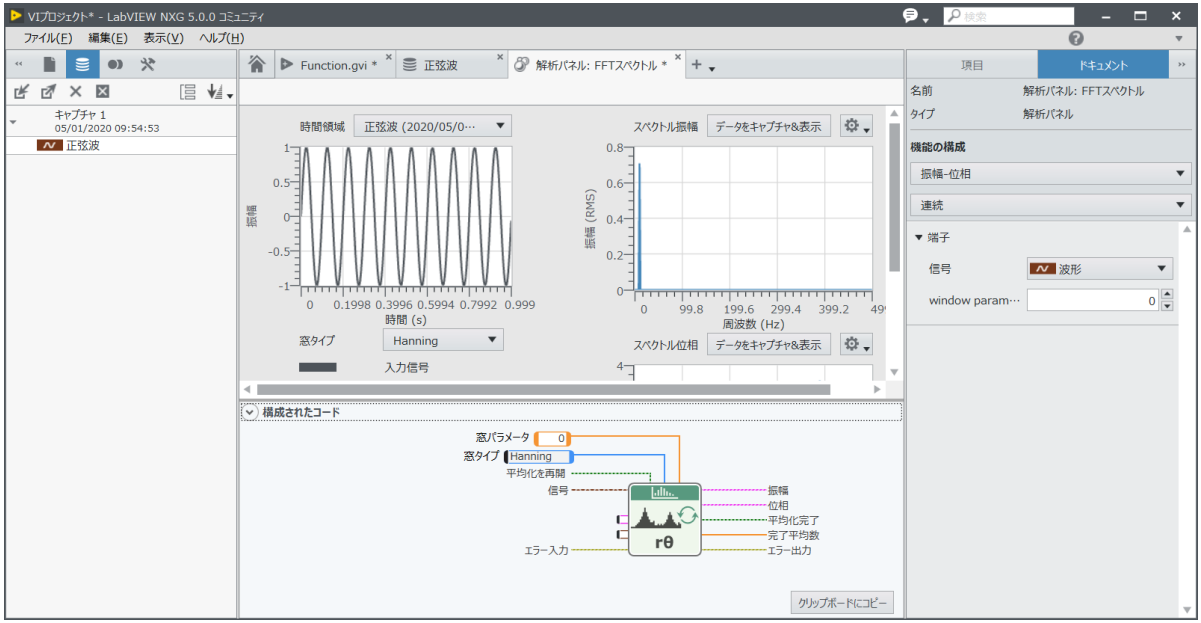


Figure C6-1 LabVIEW NXG Analysis panel

Chapter 7

Using the Latest Semiconductor Sensors

This chapter introduces how to use the Arduino as a sensor interface for LabVIEW when using the latest semiconductor sensors used in smartphones and automobiles for your hobby and daily life.

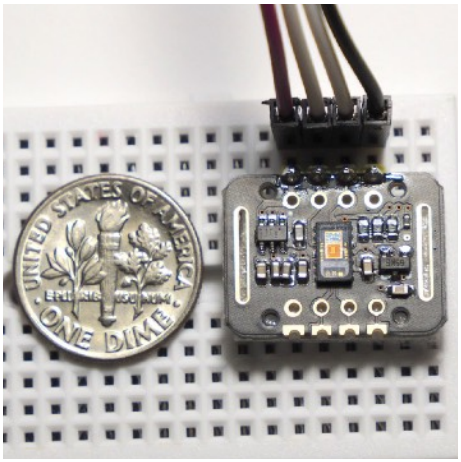
[Keywords] Semiconductor sensor, heart rate sensor, MAX30102, I2C, SDA, SCL, data sheet, register, serial port, queue, producer consumer design pattern
[Parts used] Arduino UNO x1, breadboard x1, MAX30102 module x1 (Header pin soldering required), wires x4

7.1 Semiconductor Sensors Used in Smartphones and Automobiles

Many semiconductor sensors and actuators are used in smartphones. The principle itself has been used for a long time, but by combining semiconductor microfabrication technology and peripheral circuits, it has become a compact, high-precision, lightweight sensor / actuator. When you turn your smartphone vertically, it turns to a vertical screen, and when you turn it horizontally, it turns to a horizontal screen. The function to launch the app by shaking the smartphone uses a 3-axis accelerometer or 3-axis gyro sensor. Semiconductor microfabrication technology is also used to produce tiny microphones that are difficult to imagine from looking at the size of a karaoke microphone.

Semiconductor microfabrication technology is also used in RF switches used in high-frequency communication circuits. Many types of sensors such as 3-axis accelerometer, 3-axis gyro sensor, pressure sensor, flow sensor, environment (atmospheric pressure / temperature / humidity) sensor, air quality sensor, and object detection radar are also used in automobiles.

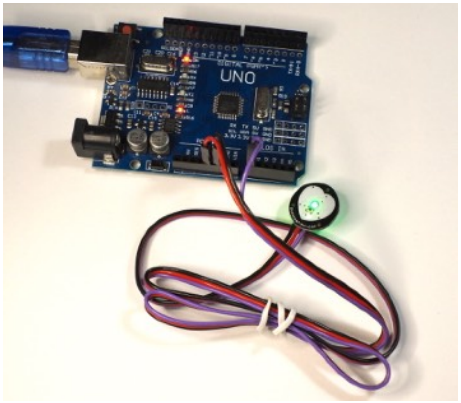
Small, high-performance semiconductor sensors are difficult to solder and difficult to use unlike traditional electrical components, but they can be purchased as a module with peripheral circuits. Many webpages and blogs describe how to use the accelerometer and environment (atmospheric pressure, temperature, humidity) sensors with Arduino, so they are easy to use.



Picture 7-1 MAX30102 Module

7.2 Sensor to Measure Heart Rate

The MAX30102, which measures heart rate, is located in the center of the module in **Picture 7-1**. A high-performance semiconductor sensor that integrates an LED, LED driver, photodetector, AD conversion, filtering, and communication interface. Touch this device with your fingertips to measure your heart rate. The percentage of oxygen bound to hemoglobin (oxygen saturation) changes with the heartbeat of the blood flowing through the skin. The optical reflectance of a particular wavelength of light changes depending on the oxygen saturation. Using this phenomenon, MAX30102 emits light from the built-in LED and receive the reflection from skin with a photo detector to measure the heartbeat. This sensor can also measure oxygen saturation (SpO2), so try it later.



Picture 7-2 Heart beat sensor (Analog Output type)

Several MAX30102 modules are available and range in price from $ 2 to $ 30. If purchased with reference to **Picture 7-1**, it can be used with the pinout diagrams in this manual.

A heart rate sensor (**Picture 7-2**) that can measure with Arduino's analog input is also widely used, but the MAX30102 is more convenient because it can measure more stably.

You will need to use a soldering iron to solder the included pin headers. If you really don't want to solder, you can temporarily test with through-hole test wires.

7.3 Operate with Sample Program for Arduino

Let's use the Arduino library and sample programs to make sure your MAX30102 board is not defective and your wiring is correct.

Now install the library for the MAX30102. In the Arduino IDE, select "**Sketch> Include Library> Manage Library ...**" and the **Library Manager** window (**Figure 7-1**) will appear. Enter max3010x in the field labeled "**Filter your search ...**" and the SparkFun MAX3010x pulse and proximity sensor library will be displayed, as shown in **Figure 7-2**. It may take some time depending on internet condition. Click **Install**.

This library is compatible with the MAX30100, MAX30101, MAX30102, and MAX30105. When the library installation is complete, close **Library Manager**.

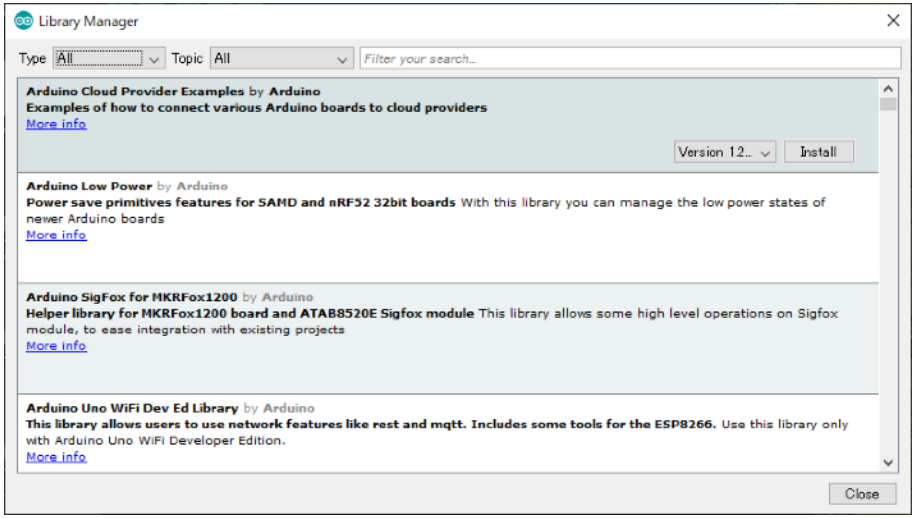


Figure 7-1 Library Manager

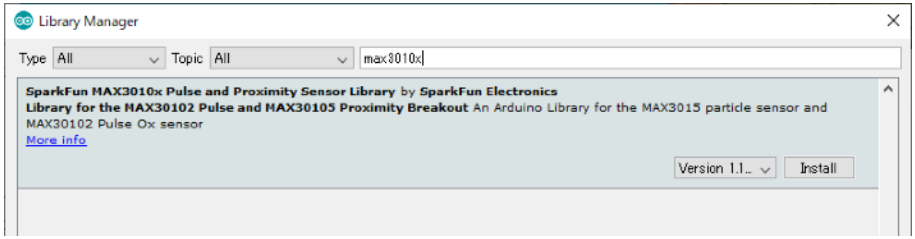


Figure 7-2 Library for the MAX30102

Select "**File> Examples> SparkFun MAX3010x Pulse and> Example4_HeartBeat_Plotter**" in Arduino IDE.

Example4_HeartBeat_Plotter.ino program opens (**Figure 7-3**). This sample sketch is a program that outputs heart rate data to the serial port and displays it on **Serial Plotter** of Arduino IDE.
Upload **Example4_HeartBeat_Plotter.ino** to Arduino.

Remove Arduino from the USB port. Connect MAX30102 to the Arduino UNO (**Table 7-1**). A pin name is printed on the back of MAX30102 module. Refer to **Figure 7-4** for the correct wiring.

Connect your Arduino to the USB port and select **Serial plotter** from the **Tools** menu. When you place your finger on the sensor, you will see a heartbeat pulse on top of large DC component. This confirms that the MAX30102 board is not defective and your wiring is correct.

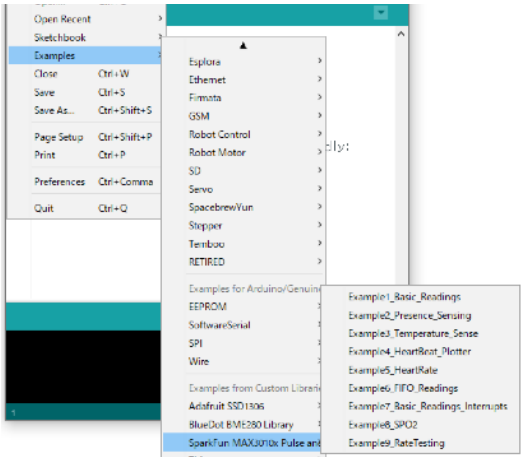


Figure 7-3 Sample Sketch

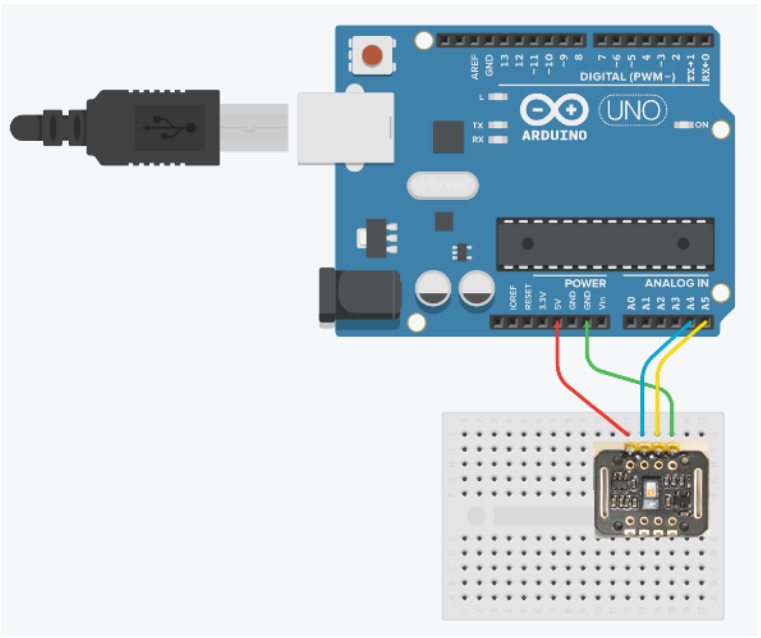


Figure 7-4 Wiring for MAX30102

MAX30102	Arduino UNO
GND	GND
SCL	A5
SDA	A4
VIN	5V

Table 7-1 MAX30102 Pins and Arduino UNO Pins

7.4 Get the Datasheet

It's a very small sensor, but it's highly functional, so get a data sheet. Search for **MAX30102 datasheet** to find **MAX30102.pdf**. You can learn how to use the MAX30102 by reading this datasheet.

A schematic of the circuit called the Typical Application Circuit (**Figure 7-5**) is available on the datasheet. From this figure, you can see that the data register can be read and written by an external processor via I2C communication. There is a red LED and an infrared LED on the left end, and there is an ADC that converts the analog value measured by the photodetector into digital data.

Just by looking at this figure, you can see that this small sensor has complicated functions built in. If you read this datasheet in more detail and create a program that uses the functions of I2C communication of LINX, you can obtain oxygen saturation (SpO2) and heart rate data without using a library. You can also deepen your understanding of the datasheet by examining the Arduino library. The library is located at "**Documents> Arduino> Libraries> SparkFun MAX3010x Pulse and Proximity Sensor Library**". If you can't find it, search for **MAX30105** on your PC.

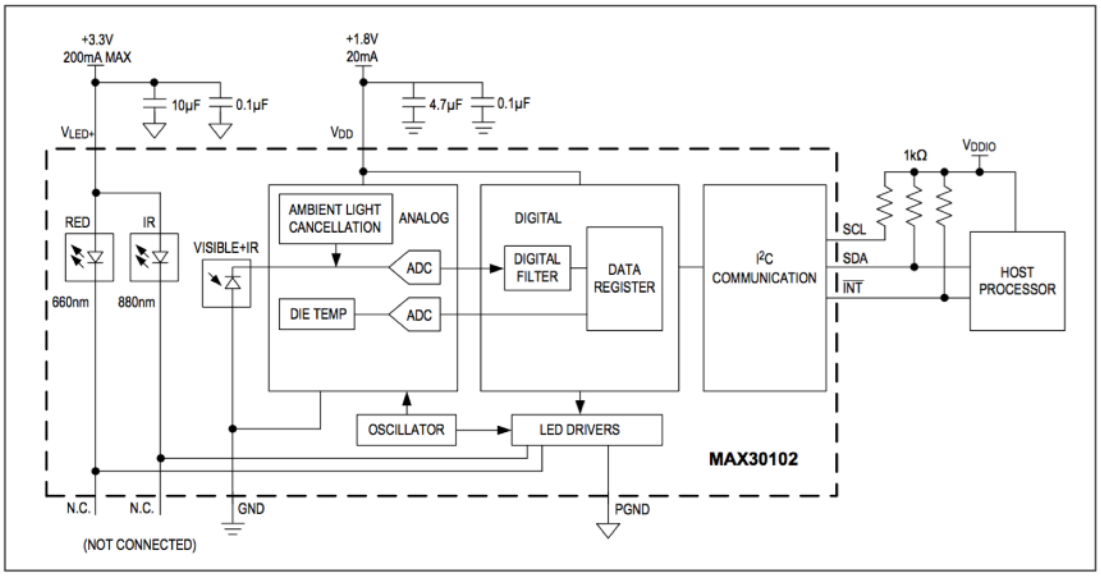


Figure 7-5 Typical Application Circuit
(Cited from MAX30102.pdf)

I think you could run **Example4_HeartBeat_Plotter.ino**. Continue to familiarize yourself with the MAX30102 by changing settings such as the sampling rate or by running other sample programs.

Looking at the datasheet (**Table 7-2**), we have set **my_HeartBeat10ms.ino**(**Figure 7-6**) with a high sampling rate and a wide measurement range (**Table 7-3**). I removed the unnecessary part of the code and used a function called **micros ()** that returns the elapsed time in microseconds to get the code working at 10 millisecond intervals.

Open **my_HeartBeat10ms.ino** from the program folder and upload it

SAMPLES PER SECOND	PULSE WIDTH (µs)			
	69	118	215	411
50	O	O	O	O
100	O	O	O	O
200	O	O	O	O
400	O	O	O	O
800	O	O	O	O
1000	O	O	O	O
1600	O	O	O	
3200	O			
Resolution (bits)	15	16	17	18

Table 7-2 MAX30102 Settings
(Cited from MAX30102.pdf)

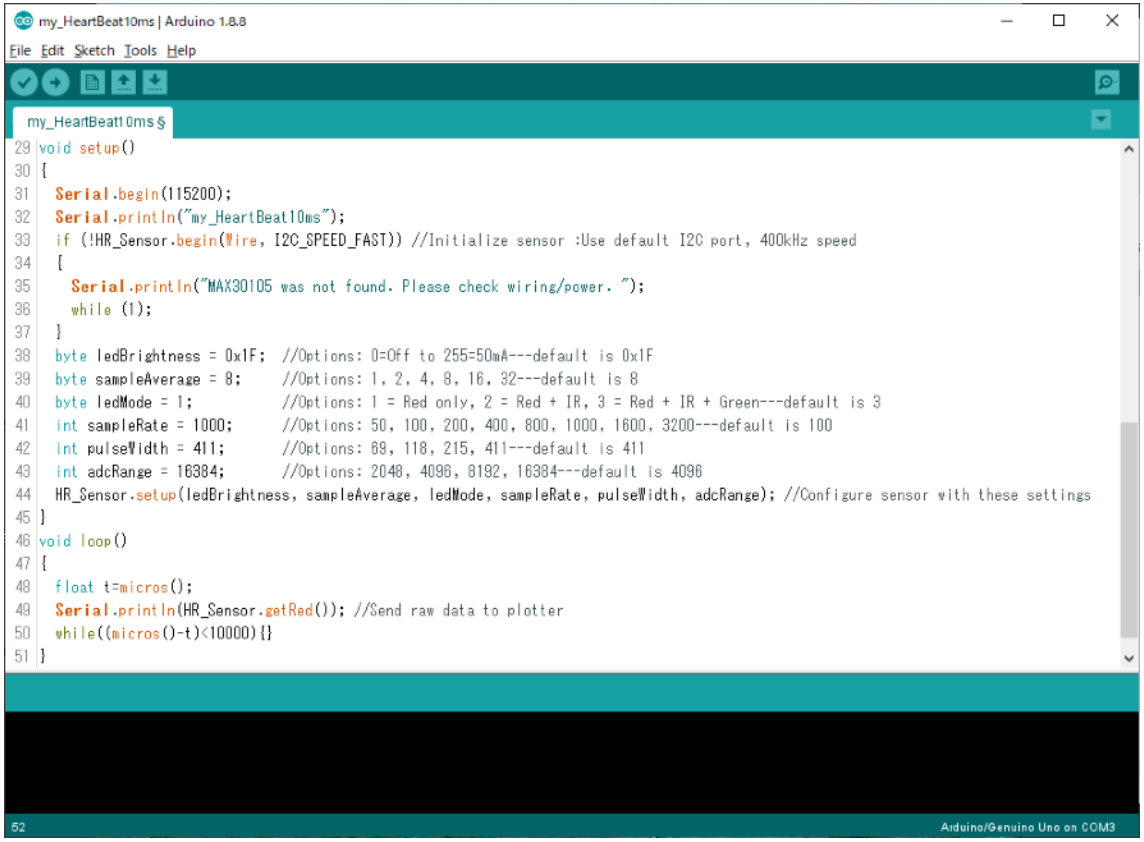


Figure 7-6 my_HeartBeat10ms.ino

to Arduino. After uploading, open the serial monitor and try receiving at 115200bps. If 6 digits are displayed one after another, we can confirm that MAX30102 sensor is connected correctly. Close Arduino IDE.

Variable name	Control register	Example4_HeartBeat_Plotter	my_HeartBeat10ms
ledBrightness	LED current	0x1F	0x1F
sampleAverage	Sample Averaging	8	8
ledMode	LED mode	3	1
sampleRate	Sample Rate Control	100	1000
pulseWidth	Pulse width Control	411	411
adcRange	ADC range full scale (nA)	4096	16384

Table 7-3 Settings of my_HeartBeat10ms.ino

7.5 Creating a LabVIEW Serial Receive Program

There are many example programs in LabVIEW, including those with features similar to the Arduino IDE serial monitor. Select **Directory Structure** in the NI Example Finder. The VI we are going to use is **Continuous Serial Write and Read.vi** under "**Instrument IO> Serial**" (**Figure 7-7**). Double click to open it and save it to the folder where you are creating the program.

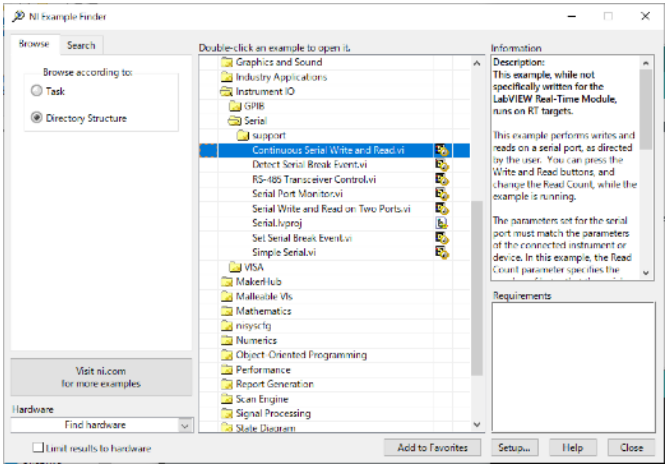


Figure 7-7 NI Example Finder

Change **VISA resource name** to the COM port number that your Arduino is using and baud rate to "115200" and press the Run button. When you press the **Read** button, the text will appear like a serial monitor (**Figure 7-8**). The block diagram looks complicated because this example VI has many features (**Figure 7-9**). Delete unused functions and save as **Continuous Serial Read.vi** (**Figure 7-10**). You can convert the 6-digit string that this VI receives to a number that you can add, subtract, multiply, divide, or display in a graph.

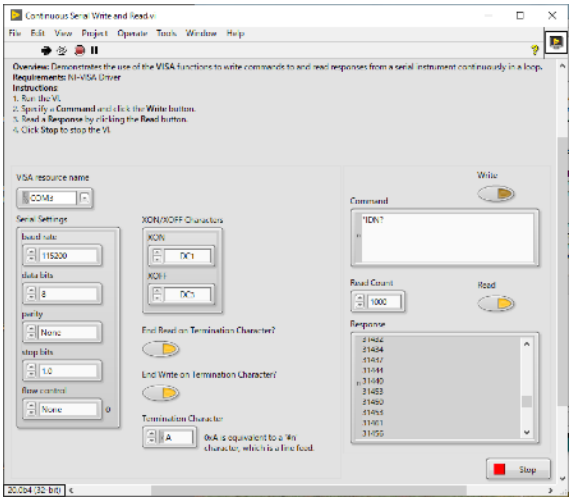


Figure 7-8 Continuous Serial Write and Read.vi

Arduino sends at most a 6-digit number and a line feed code (**CR** and **LF**), which is worth two characters, every 10

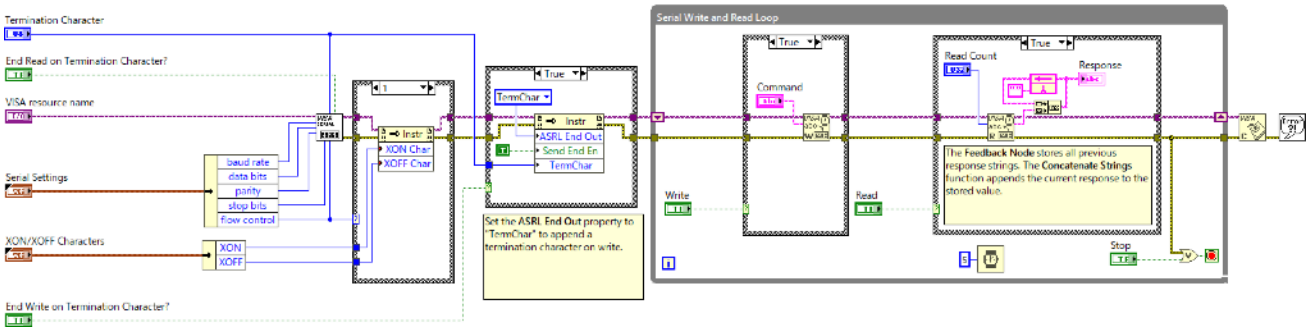


Figure 7-9 Block Diagram of Continuous Serial Write and Read.vi

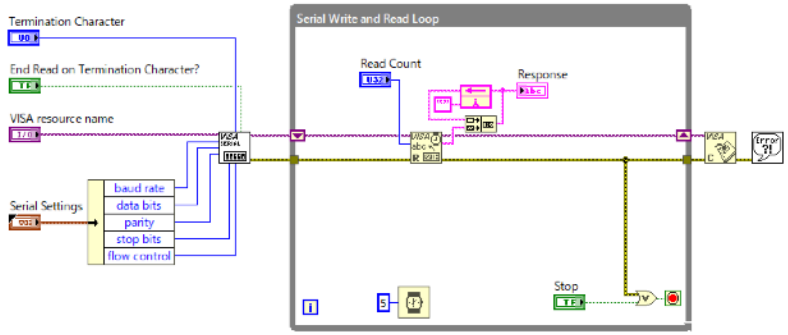


Figure 7-10 Continuous Serial Read.vi

milliseconds, so let's estimate the time required for serial communication. The calculation shown in **Table 7-4** shows that about 0.7 milliseconds is required. Here, the communication speed of serial communication is 115200bps, but in the case of 9600bps, which is often used, the communication speed is low and the margin is almost gone. If you process and save the data while receiving it, you may lose the data. In such situations, use the Producer / Consumer design pattern, a mechanism that separates data reception and data processing to allow each process to work independently.

This heart rate measurement program with the MAX30102 has some headroom, but let's use the producer / consumer design pattern. Select "**File> New ...**" from the menu. The **New** window will open. Select "**From Template> Framework> Design Patterns> Producer / Consumer design pattern (data) .vi**" and click the **OK** button. The block diagram (**Figure 7-11**) has two While Loops that queue the string received in the upper loop into the lower loop. A queue is like a line of customers in front of a popular store. Even if the Consumer Loop below is occasionally stuck, the line will just become a little longer but can be processed without omission.

This design pattern only provides some buffer to handle the unstableness of the processing time, so if processing time continues to be too slow, the buffer will eventually overflow, but it can handle temporary glitches cause by, for instance, UI processes like writing to a graph.

Save it as **max30102ChartDisplay.vi** in the folder where you are creating the program. Copy **VISA Configure Serial Port**, **VISA Read** and **VISA Close** from **Continuous Serial Read.vi** together with the controls and indicators connected to them and paste them in the loop above.

It should look like **Figure 7-12**. The VI can

Item	Value	Notes
Transfer string length	8 bytes	Transfer characters 6 bytes Control characters (CR+LF) 2 bytes
Number of transfer bits per character	10 bits	data bit length 8 bits start bit 1 bit stop bit 1 bit
Total number of transfer bits	80 bits	(Transfer string length) X (Number of transfer bits per character)
Data transfer rate (bps)	115200 bps	bps : bits per second
Transfer time (sec)	0.00069sec	(Total number of transfer bits) / (Data transfer rate)

Table 7-4 Estimated Time Required for Serial Communication

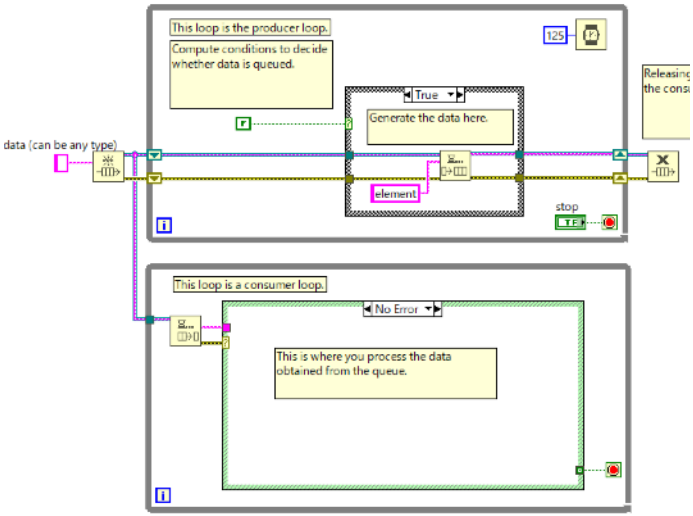


Figure 7-11 Producer/Consumer Design Pattern (Data).vi

run as is, but make the following three changes to pass the incoming string to the lower loop through the queue:

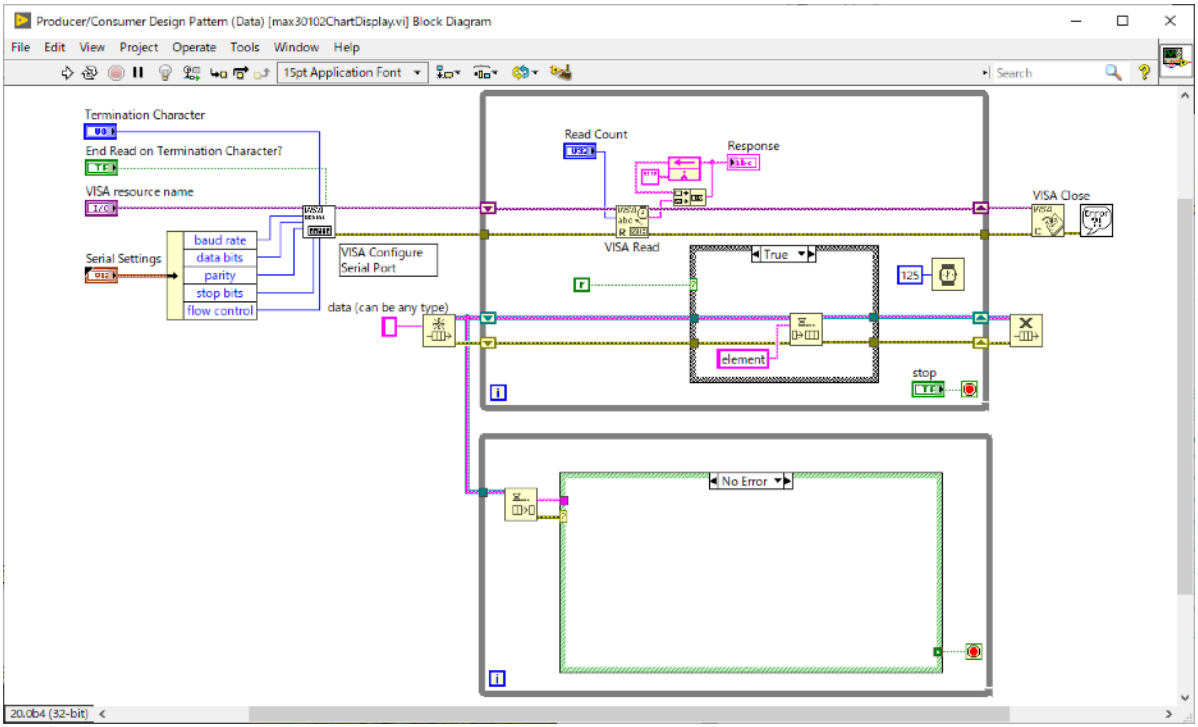


Figure 7-12 Copying VISA Functions from Continuous Serial Read.vi

1. Move the part that receives the character string output from **VISA Read** function to **No Error** subdiagram of the case structure in the Consumer Loop. Wire it to the string output from **Dequeue Element**.

2. The **Return count** output of the **VISA Read** function outputs the number of characters read, so the **Queue Element** should be executed only if the number of characters read is greater than zero. To do so, connect **Return count** to the input of **Greater Than 0?** function and the output of **Greater Than 0?** function to the case selector.
   Connect **read buffer** output of **VISA Read** function to **Queue Element**.

3. Change the constant connected to **Wait (ms)** function to "5".

It should look like **Figure 7-13**. Run the VI. Now the VI is running in Producer / Consumer design pattern. Up to this point, what is sent as a numeric character string is displayed as is.

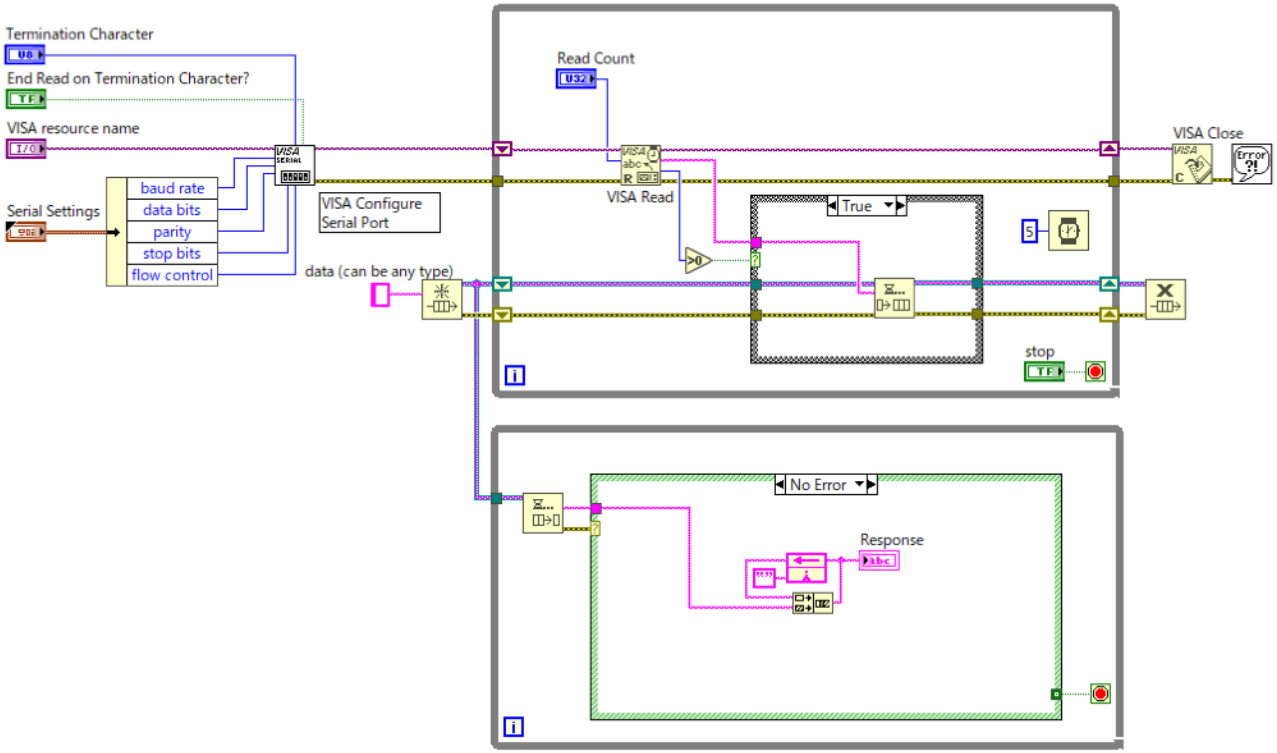


Figure 7-13 Pass Incoming String in Queue

Drag and drop **Waveform Chart** from **Graph** palette to the front panel. Use **Decimal String To Number** function in **Strings** palette to convert a numeric string to a number, as shown in **Figure 7-14**. Wire as shown in **Figure 7-15**. When executed, the blood flow pulse is displayed as shown in **Figure 7-16**.

This program will slow down after running for a while. The reason for slowing down is that there are too many characters in **Response** string indicator. Since the data is displayed on the waveform chart, **Response** is no longer needed, so delete it (**Figure 7-17**).

The producer / consumer design pattern uses two While Loops, so the stopping mechanism when the **STOP** button is pressed is tricky. Check the operation after **Execution Highlights** is set to **ON** and the **STOP** button is pressed.

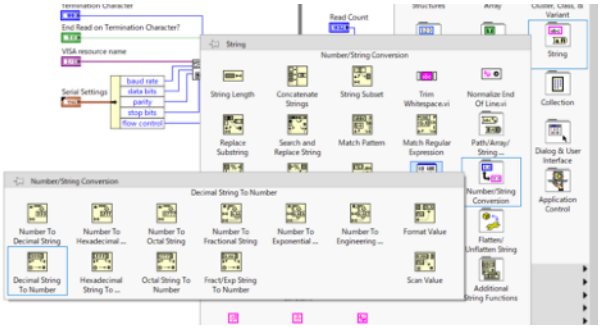


Figure 7-14 Decimal String To Number

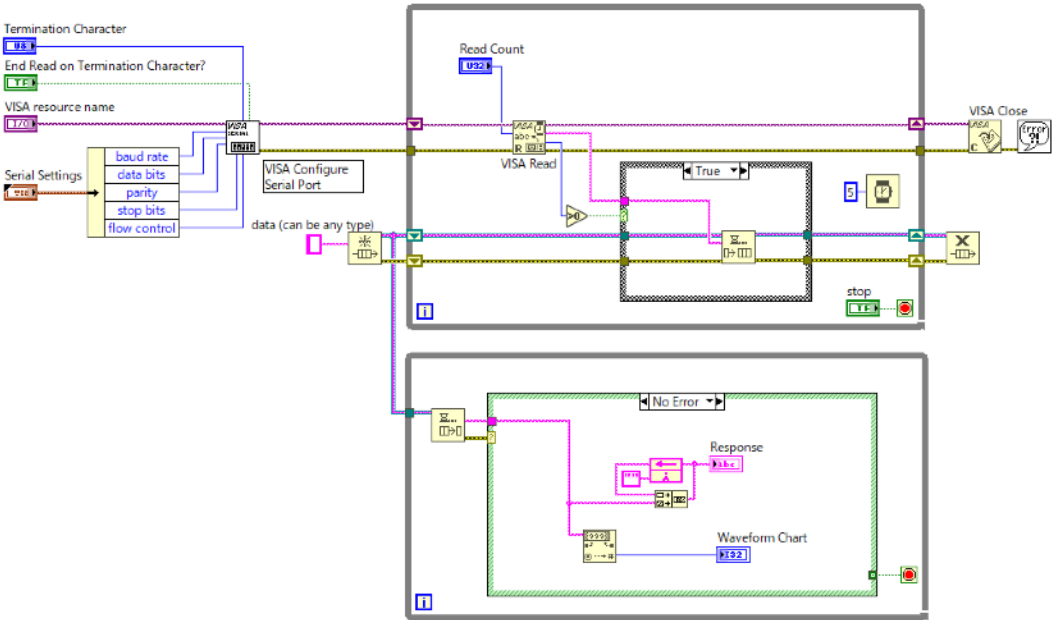


Figure 7-15 Convert String into Number and Display it in Waveform Graph

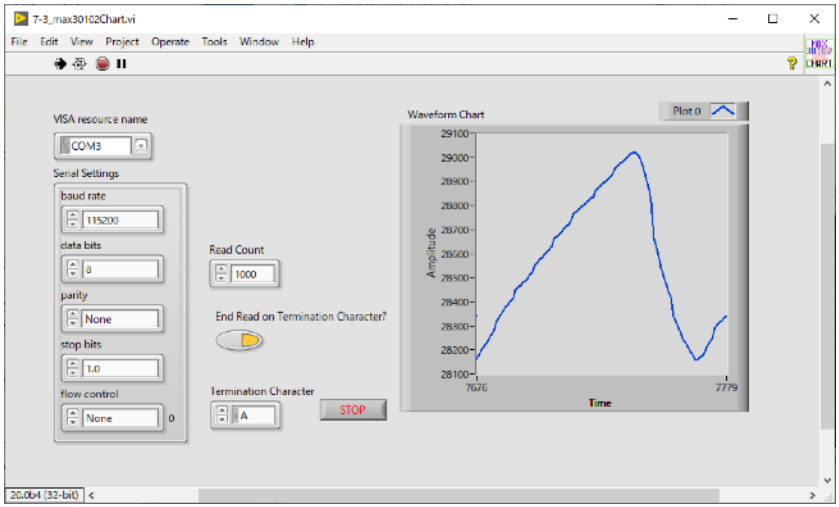


Figure 7-16 Execute max30102ChartDisplay.vi

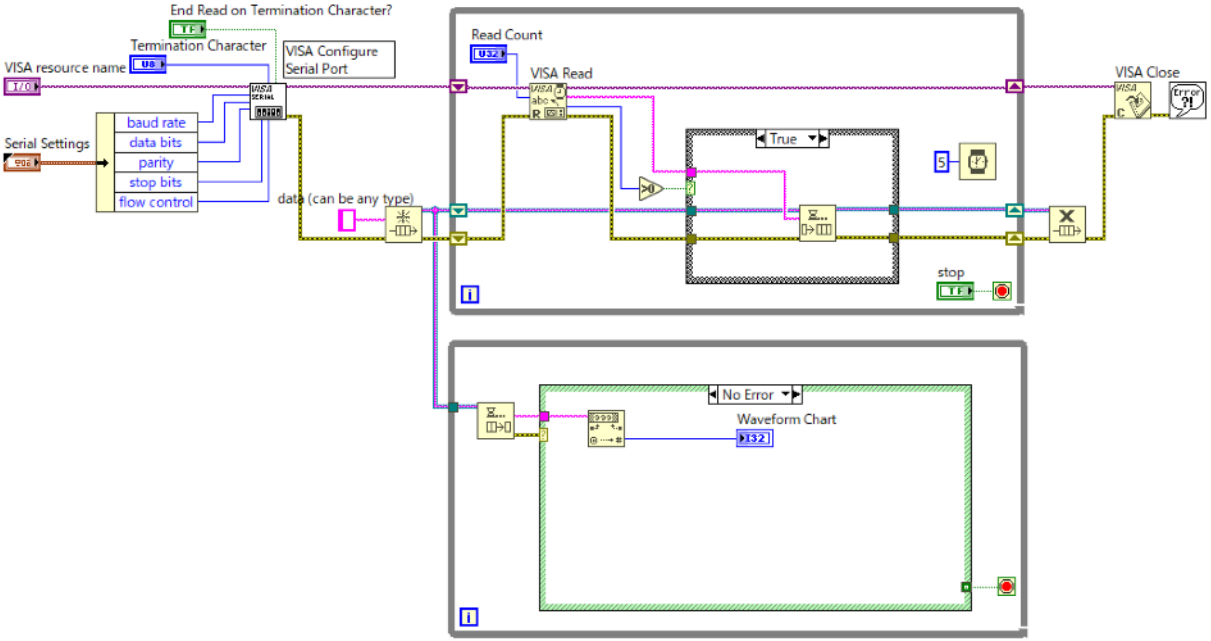


Figure 7-17 Delete Response Indicator

7.6 Create Heart Rate Measurement Program

Modify **max30102ChartDisplay.vi** to create a VI that calculates the heart rate, which is the number of heartbeats per minute. Save **max30102ChartDisplay.vi** with the name **MAX30102_Plot.vi**. This VI uses a signal processing function to estimate heart rate from 10 seconds of data.

The sampling interval of the data is important for performing frequency analysis. Since the data is sent from Arduino every 10ms, the sampling interval **dt** is 0.01s. The number of data in 10 seconds is 1000, so prepare an array of 1000 elements. **Figure 7-18** shows an array of 1000 elements created and connected to a shift register. Not shown, but you should also wire the **Error** subdiagram. New data will be received every 10msec, so the new data will always be updated at the end of the array. **Rotate 1D Array.vi** is a function that moves elements in sequence, thinking that the arrays are connected at the beginning and end like a ring.

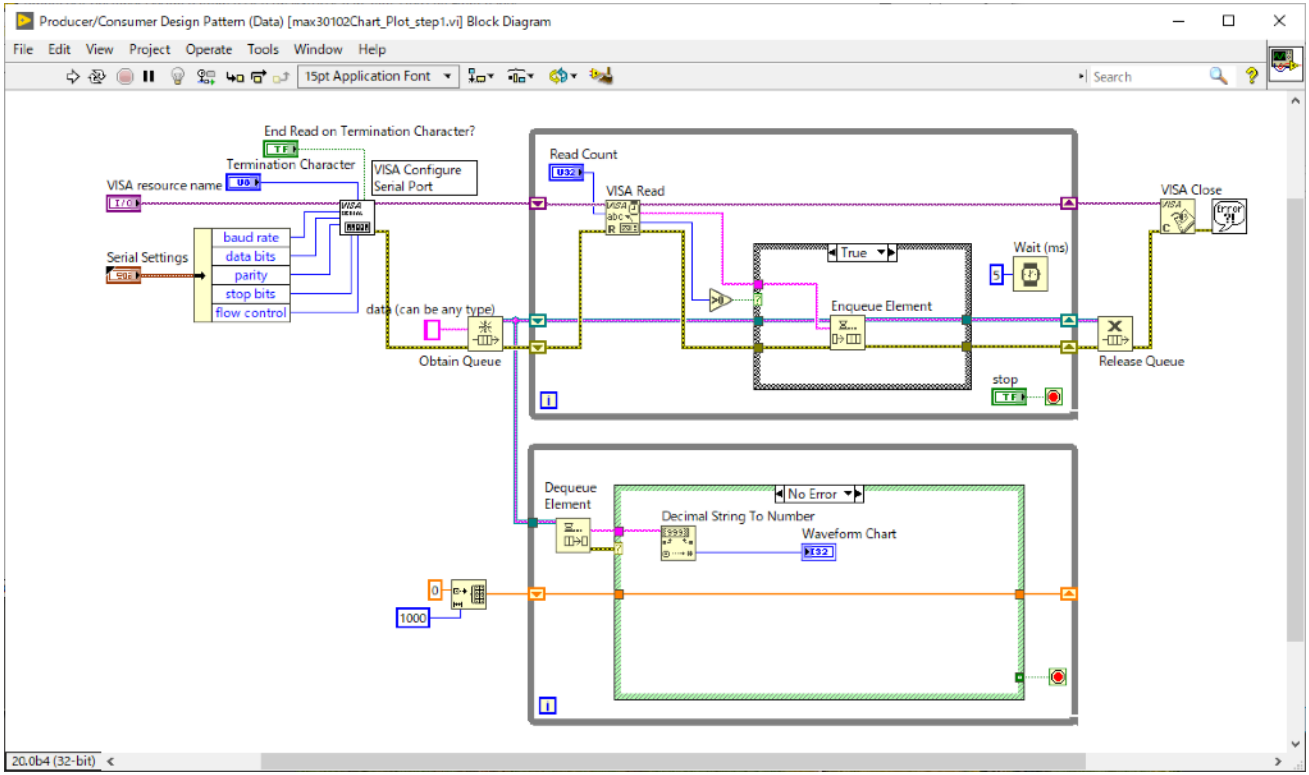


Figure 7-18 Create Array and Shift Register

In **Rotate 1D Array.vi**, the number of movements **n** can be specified. Entering "1" moves the last element at index 999 to index 0 and the element at index 0 to index 1. This function icon is a good example of how a function can be imagined just by looking at the icon.

Entering "-1" moves index 0 to index 999 (last). The element at index 999 is moved to index 998. The element with index 1 moves to index 0. When you rotate the array element by "-1" like this, the index 999 becomes the oldest data, so replace it with the new data that came in. Now you can update the new data so that it is always at the end of the array.

The usage of such an array is called a ring buffer (**Figure 7-19**).

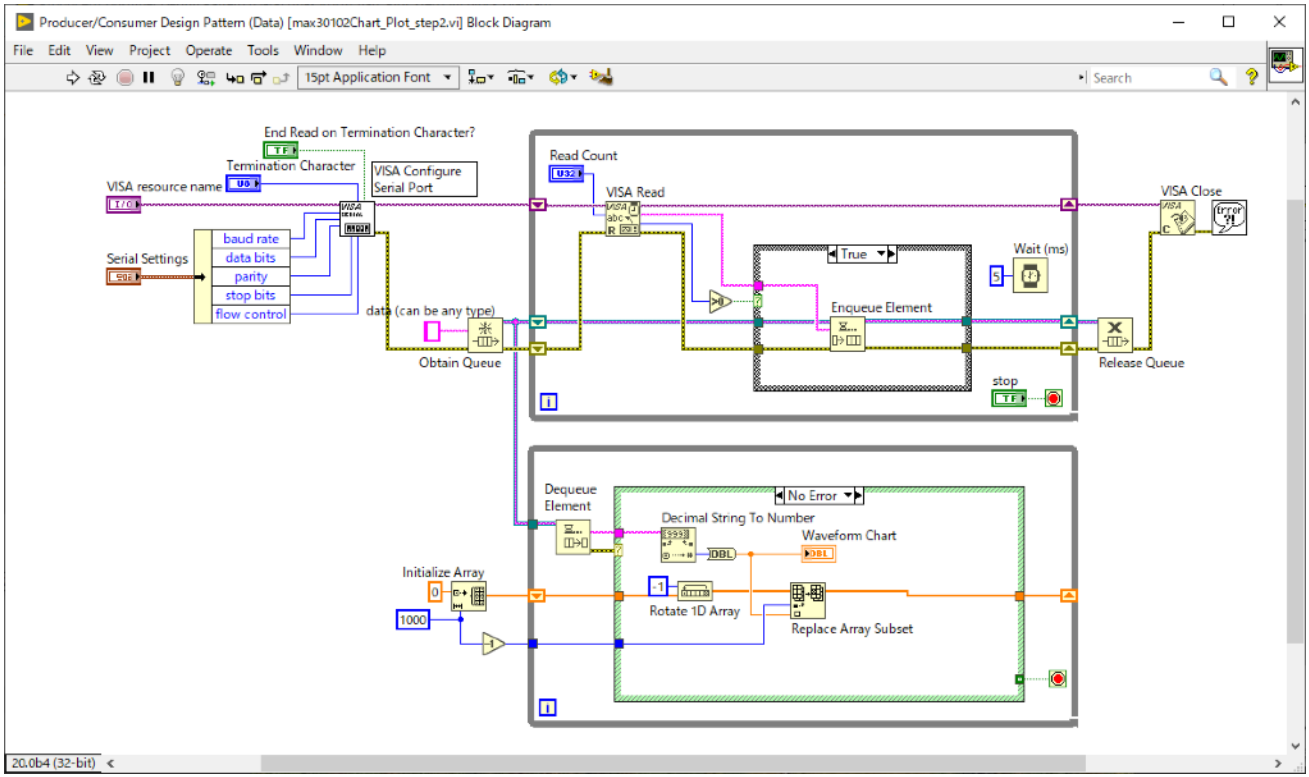


Figure 7-19 Create Ring Buffer

In **Figure 7-20**, the waveform data is created from the sampling interval and the data array and connected to the **waveform graph**.

When you run the VI, it should look like **Figure 7-21**.

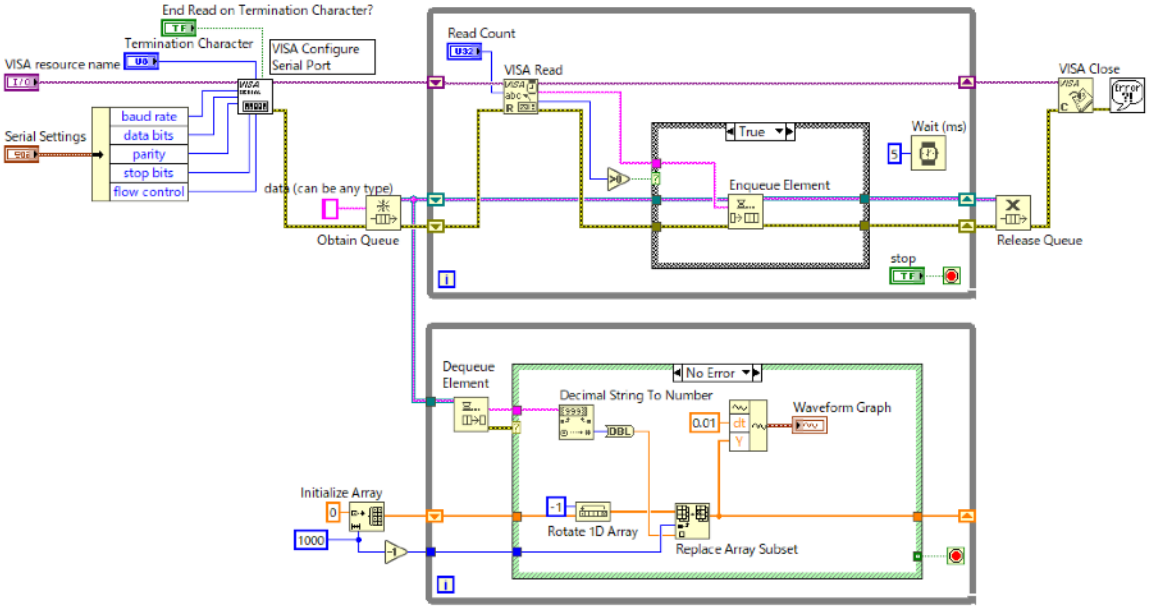


Figure 7-20 Create Waveform Data and Connect to Waveform Graph

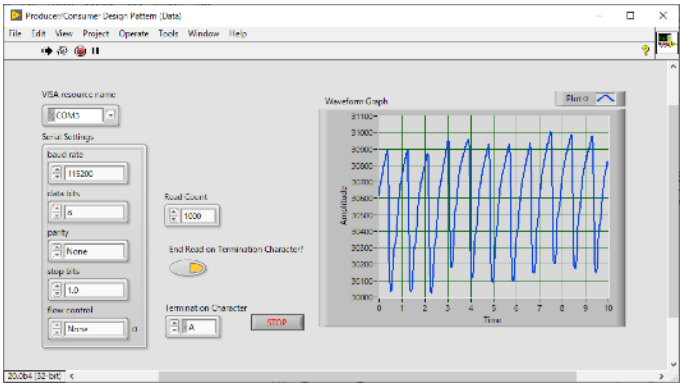


Figure 7-21 Display of Heartbeat Data on Waveform Graph

When you input the waveform data into **Extract Single Tone Information.vi**, frequency analysis is performed and the most prominent frequency is output. Multiply by 60 to get your 1-minute heart rate (**Figure 7-22**).

Figure 7-23 shows the display screen.

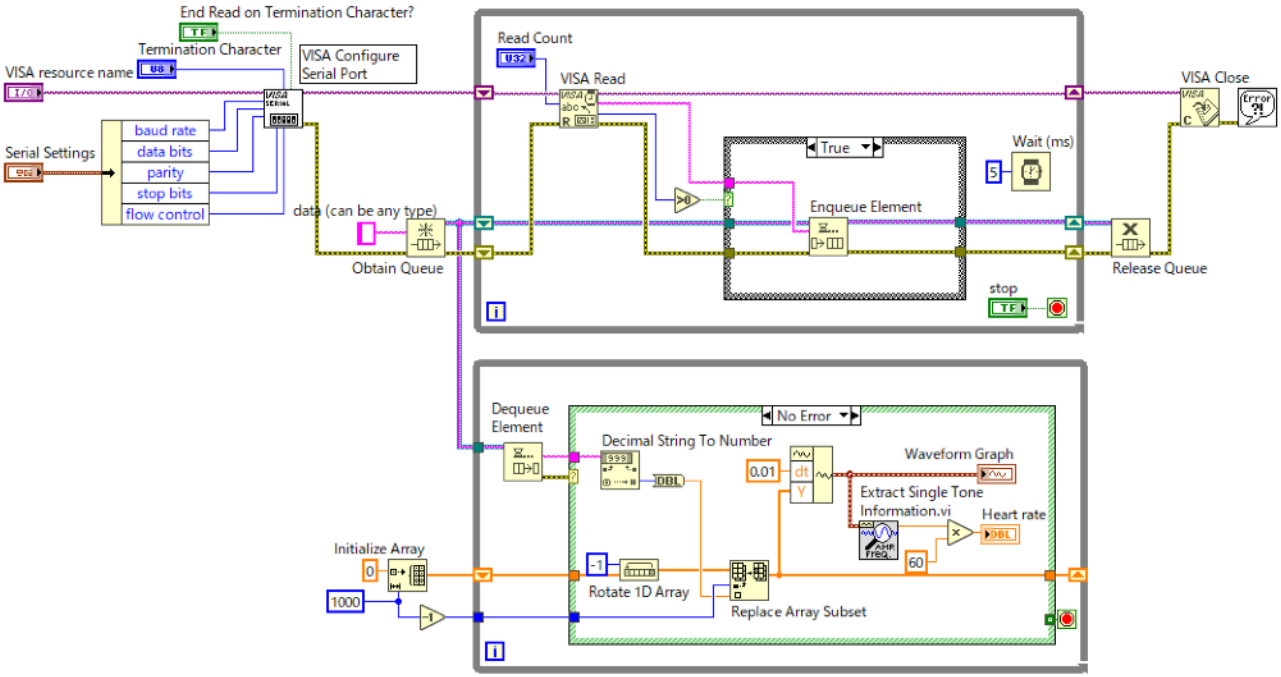


Figure 7-22 Frequency Analysis with Extract Single Tone Information.vi

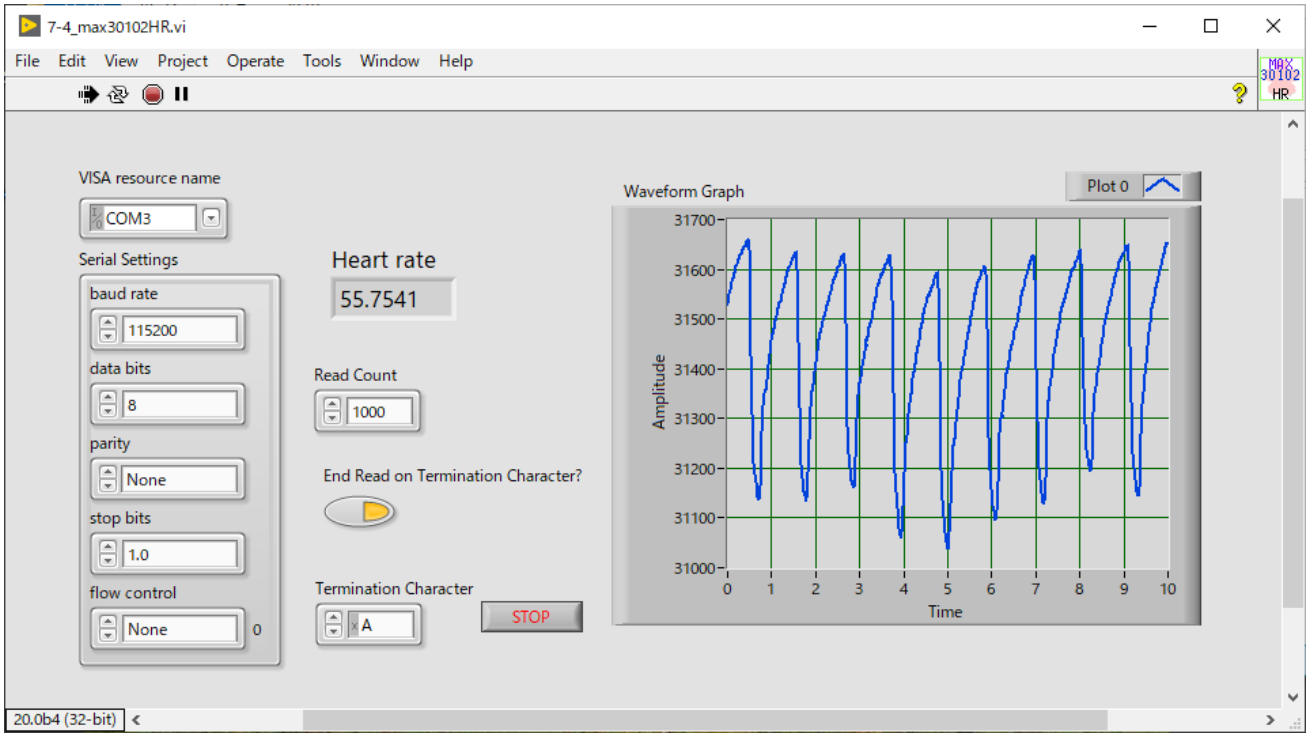
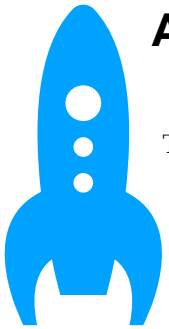


Figure 7-23 Heart Rate Indicator



Article 7 What LabVIEW NXG Aims for

Technology changes very fast, and the latest technology up to yesterday may be outdated today. Technology is becoming more complex, and at the same time, new products and technologies need to be introduced more quickly. LabVIEW NXG is designed to help you quickly achieve the measurement and control you need for product development. As we already explained in the previous column, NXG allows you to do **Data Acquisition** or **Analysis** before programming.

Although LabVIEW NXG is still under development, it is expected to have all the features that LabVIEW has, plus many features that LabVIEW does not have. Familiarize yourself with LabVIEW now. In the future, when LabVIEW NXG goes beyond LabVIEW, consider switching to LabVIEW NXG.

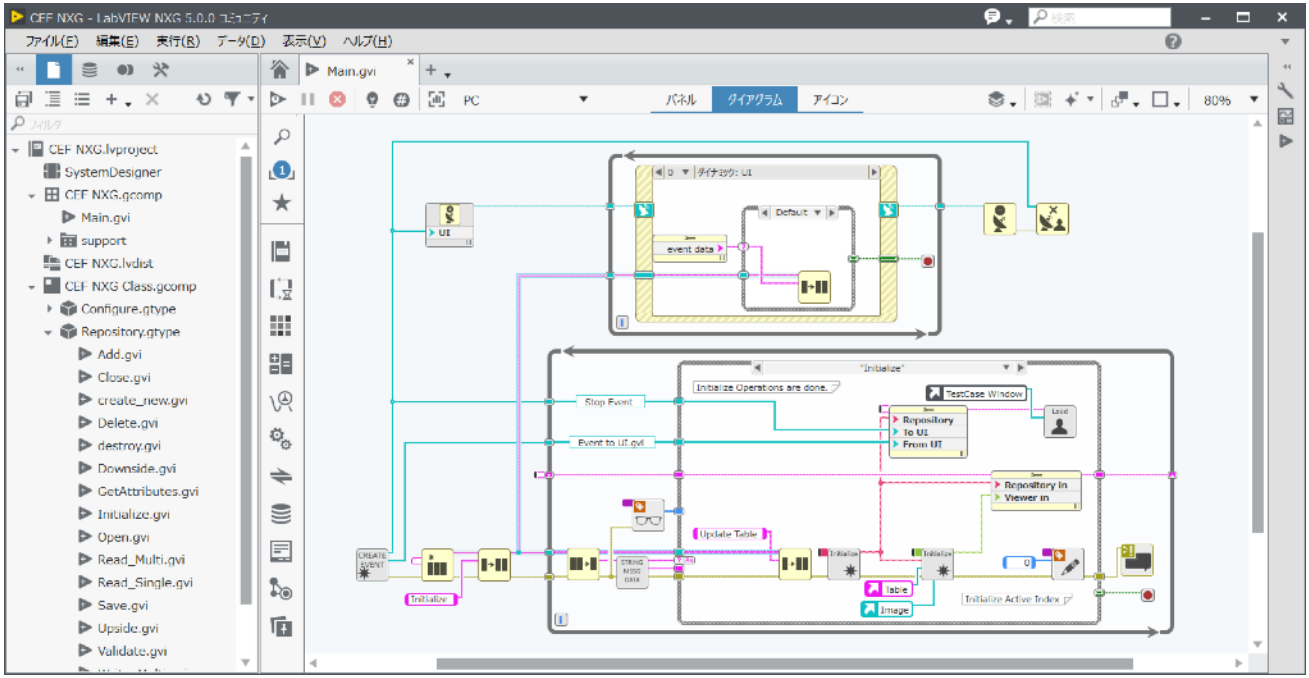


Figure C 7-1 LabVIEW NXG

List of Example VIs

Chapter	Program Name	Description
3	3-1 LED Simulator.vi	This program calculates the LED current for the supply voltage and the current limiting resistor with the specified LED characteristics (forward voltage and maximum current).
3	3-2 my_LED_step1.vi	This program calculates the LED current from the supply voltage, forward voltage and current limiting resistor.
3	3-3 my_LED_step2.vi	This program was set up so that you cannot enter negative numbers into the "forward voltage" control.
3	3-4 my_LED_step3.vi	In this program, if the LED current is higher than the "maximum current", the "LED is damaged?" Indicator lights up.
3	3-5 my_LED_step4.vi	This program uses a While Loop, so it runs continuously.
4	4-1 SoundVIEW.vi	This program records the audio signal from a microphone and a PC sound source, analyzes the frequency during playback and displays a spectrogram.
4	4-2 AvailableDevice.vi	This program displays the sound source recognized by LabVIEW.
4	4-3 Finite Sound Input.vi	This is a sample program provided by NI for capturing audio signals. It will be changed and used in Chapter 4.
4	4-4 parrot.vi	It is a program that takes in an audio signal, waits a short time and plays it back.
4	4-5 reverse.vi	This program plays the recorded signal in reverse.
4	4-6 ForLoopSum.vi	This program calculates the sum of 1 to N.
4	4-7 PrimeNumber.vi	This program finds all prime numbers less than the specified positive integer.
4	4-8 2D_array_Display.vi	This program introduces various display methods for 2D arrays.
4	4-9 PiPoPa.vi	It is a program that generates the tone dial used when making a call.

Chapter	Program Name	Description
4	4-10 easyRecorder.vi	It is a program that can record and play back using the state machine design pattern.
5	5-1_PushCounter.vi	It is a counter program that uses Arduino and LINX, and it increases by 1 when the switch is pressed, and returns to 0 when it exceeds 10.
5	5-2_FanControl.vi	This program controls the fan speed with a switch counter.
5	5-3_FanControlWith LongPushStop.vi	It is a program that stops the fan by holding down the switch.
5	5-4_FanControlWith LongPushImmediate Stop.vi	This program will stop the fan as soon as you hold down the switch. need improvement.
5	5-5_ Push Counter (LatchWhenReleased).vi	This is a counter program created using another algorithm that works the same as 5-2.
5	5-6_Fan Control with Long Push Stop (LatchWhenReleased).vi	This is a VI that behaves like 5-4 created with another algorithm. If you keep pressing the switch, the fan will stop immediately.
6	6-1_LINX - Analog Read N Channels.vi	This is a sample VI provided by MakerHub as a programming example for analog input in LINX. Change and use in Chapter 6.
6	6-2_LED VI Curve.vi	This is a program that performs an experiment to measure the voltage-current characteristics of an LED with a simple circuit and perform regression analysis on the obtained data.
7	7-1_Continuous Serial Write and Read.vi	This is a program provided by NI as an example of a serial communication program. Modified and used in Chapter 7.
7	7-2_Continuous Serial Read.vi	This is a serial communication reception program modified based on the example of serial communication program.
7	7-3_max30102Chart.vi	This program uses the producer / consumer design pattern to receive and display the heartbeat signal from the Arduino.
7	7-4_max30102HR.vi	This program uses a producer / consumer design pattern to receive a heart rate signal from the Arduino and display the heart rate.
7	Example4_HeartBeat_ Plotter	This program comes with a library provided by SparkFun for using the MAX30102 heart rate sensor with Arduino.

Chapter	Program Name	Description
7	my_HeartBeat10ms	This Arduino program uses serial communication to send heartbeat data at 10 millisecond intervals.

Afterword

LabVIEW does not rank in the so-called "programming language ranking." This is because the subject of the survey are programmers. LabVIEW was invented for non-programmers such as scientists and engineers. It is very useful when they want to make programs and devices that automate their tasks to improve the work efficiency. Can you imagine how much fun it would be if you could borrow a little bit of help from the computer to make your hobby more convenient? But can you also imagine how difficult and daunting it would be to learn a language for programmers just for this little bit? LabVIEW can help you here. "Automatic devices" are needed in various industries, and there are many people who design, program, and maintain them. LabVIEW is the most popular tool among such professionals.

The Japan LabVIEW Users Group is comprised of professionals who love LabVIEW, and members use LabVIEW every day to "enjoy" their work and hobbies. Furthermore, we wish that more people could get a hand on this incredible tool. However, it is truly regrettable that LabVIEW is not so widespread, perhaps because it is for the professionals and the price is not cheap.

One day, we obtained the information that "LabVIEW Professional Edition will be available for free." To this we thought, “This may drastically increase the number of our LabVIEW peers!” As beta testing began and our expectations grew, we decided to create a "document for users who interact with LabVIEW for the first time in the Community Edition." Publishing this for free. Targeting the content for junior and senior high school students for ease of understanding. “Physical computing,” where the computer and the real world are physically connected, should be the theme to allow the readers to experience the joy of programming the most. Announcing the publication in May when the official release is scheduled. The project was started with such plans at the end of January 2020. Volunteers presented outlines, drafted text for each part, and created example VIs. Another member conducted the experiment by following the text to check the reproducibility. Another member translated the text from Japanese to English. This book was created through proofreading, writing additional columns, designing drawings, and setting layouts. We hope you can feel our love for LabVIEW.

We hope you become fond of LabVIEW. And if you make a new VI that doesn't appear in this document, or if you wrote an article about a perspective or an idea in your area of expertise, please let us and other users know. Let's enjoy together.

Hirotake Watashima
Chairman, Japan LabVIEW Users Group

Publication :
Volunteer members of Japan LabVIEW Users Group

Author : Yusuke Tochigi (Introduction, Chapter 1 to Chapter 3)
Author : Koji Ohashi (Chapter 4 to Chapter 7)
Author : Hirotake Watashima (Chapter 6 Appendix, Afterword)
Translator : anonymous (Introduction)
Translator : York Kitajima (Chapter 1 to Chapter 3 , Chapter 6)
Translator : Tamon Minami (Chapter 5)
Translator : Shogo Shinohara (Chapter 5)
Translator : Yusuke Tochigi (Figures)
Translator : Hirotake Watashima (Chapter 6 Appendix, Afterword, Figures)
Translator : Koji Ohashi (Chapter 4, Chapter 7, Figures)

Cover Design / Photo : Koji Ohashi
Planning : Yusuke Tochigi
Facility Management : Yusuke Tochigi
Editor : Koji Ohashi
Proofreader (J) : Hirotake Watashima, Hidetoshi Emura
Proofreader (E) : York Kitajima, Tamon Minami, Yusuke Tochigi

Sample VI : Koji Ohashi
Sample VI (Improvement) : Hirotake Watashima
（See VI Properties > Documentation）

Operation Verification：Hirotake Watashima

Operation Verification：Hidetoshi Emura

Operational Cooperation : Atsushi Kamoshida
Account Manager | Frontier Science Region
National Instruments Japan Corporation

Publication date : April 28th 2020 (Version 0.9.0)
June 24th 2020 (Version 1.0.0)
June 25th 2020 (Version 1.0.1)