2016 Intel 中学生暑期夏令营

Introduction to Robotics Programming with VIPLE

-- Visual IoT/Robotics Programming Language Environment

Sponsored by Intel China

In cooperation with Arizona State University Beijing University of Technology

Beijing, August 7 – 9, 2016

Table of Contents

Lab 1
Introduction to VIPLE

Overview

This lab will introduce you to -- and allow you to become familiar with – ASU VIPLE, the Visual IoT/Robotics Programming Language Environment.

Lab 1 Preparation

Read this section completely before taking the pre-lab quiz

What is Visual IoT/Robotics Programming Language Environment?

There are a number of great visual programming environments for computing and engineering education. MIT App Inventor uses drag-and-drop style puzzles to construct phone applications in Android platform. Carnegie Mellon's Alice is a 3D game and movie development environment on desktop computers. It uses a drop-down list for users to select the available functions in a stepwise manner. App Inventor and Alice allow novice programmers to develop complex applications using visual composition at the workflow level. LEGO and Microsoft have developed visual robotics application development environments for beginners. LEGO's NXT and EV3 environments are for LEGO robots only. Microsoft has discontinued its VPL (Visual Programming Environment).

ASU Visual IoT/Robotics Programming Language Environment (VIPLE) is developed at ASU. It is a service-oriented software development environment used for developing IoT (Internet of Things) and robotics applications on a variety of hardware platforms. The platforms include LEGO EV3 and Intel Edison robots. VIPLE uses workflow and service-oriented technologies as its underlying foundation for creating simple and easy-to-use services in a visual programming manner. The idea is to have human (developers) to draw the flowchart (workflow) of the intended application. The development environment (tool) can convert (compile) the flowchart into executable, and thus, to make software development easier and faster. The development process is to drag and drop blocks that represent components and services, and connect them with wires/lines. This simple process makes it possible for even non-programmers to create their IoT and robotics applications in minutes.

Let us follow the Engineering Design Process:

1. Define Problem and Requirement
2. Research
3. Sketch Alternative Solutions
4. Modeling (Drawing the Flowchart)
5. Analysis
6. Simulation
7. Prototyping
8. Final Selection
9. Implementation and Testing

We apply this design process to the software development process. In the traditional software development, the flowchart is used as a conceptual model for the developers to understand problem better. In the service-oriented visual development process using VIPLE, the flowchart becomes a mathematical/logical model that can be compiled into executable, and thus eliminated, or at least reduced the burden of the implementation (coding) step that translates the flowchart into text-based code. This approach can be applied not only to robotics applications, but also to general software development. There are several workflow-based

development environments that allow to use vial programming for general software development, including IBM WebSphere, Microsoft Workflow Foundation in Visual Studio, and Oracle SOA Suite, which allow developers to draw flowchart of any type applications, e.g., an online banking system, an e-business system, or an image verification system. The compiler can directly translate the flowchart into executable.

Recall that the role of software engineers (software architects) is to understand the problem and develop a solution to solve the problem. It is not the main responsibility of software engineers to write the code to implement the solution. The advanced development tool does not reduce the need of software engineers. However, it reduces the need of programmers. According to the Occupational Outlook Handbook of the U.S. Department of Labor (http://www.bls.gov/ooh/), the need for software engineers will increase at high percentage rate in the next 10 years, while the need for programmers will decline. Now you can better understand the reason behind the statistic numbers: Coding (implementation) job could be automated by advanced software tools, but problem definition, requirement writing, modeling, and analysis jobs cannot be replaced by machines or by tools.

In addition, VIPLE includes a 3D simulation environment for users to test their programs prior to loading the programs to the hardware platforms (robots). Testing a program in a simulation environment allows the developers to identify software problems before they potentially become mixed up with hardware problems. Figure 1 shows the 3D simulation of a robot navigating in a maze. The red beams show the distance sensors measuring the distances. The simulation environment is developed using Unity game engine. The wall of the maze can be changed (added or removed) by licking the wall.

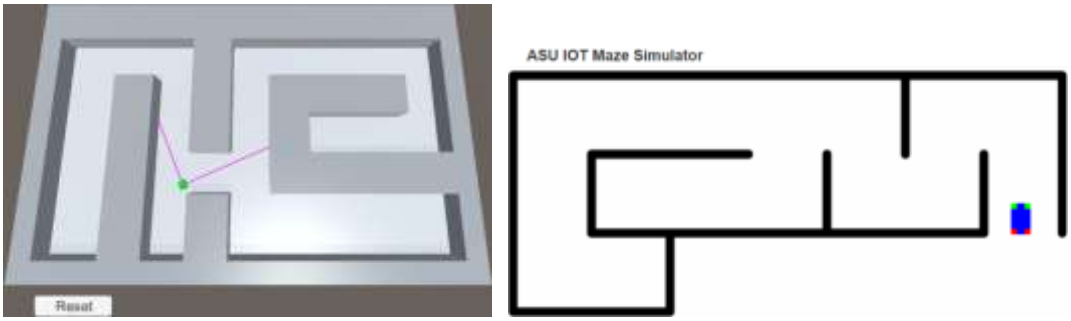


Figure 1. VIPLE 3D Unity Simulator and Web Simulator

VIPLE is an application development environment designed on a data-based programming model rather than control flow model typically found in conventional programming languages such as C++ and Java, in which a series of imperative commands executed sequentially. A data-flow program is more like a series of workers on an assembly line, who do their assigned task as the materials arrive. As a result, VIPLE is well suited to programming robotics applications, as well as a variety of concurrent or distributed applications. In our labs, we will also use VIPLE to design circuits, program vending machines, and so on.

VIPLE is targeted for novice programmers with a basic understanding of concepts like variables, data types, if / else statements, loops, and logical thinking. However, VIPLE is not limited to novices alone. The compositional nature of the programming language may appeal to more advanced programmers for rapid prototyping or code development. In addition, while its toolbox is tailored specifically for developing robotics applications, the underlying architecture is not limited to programming robots and could be applied to other applications, such a game, a complex process of manufacturing, controlling devices in a smart home, and other design processes. Its simulation environment can virtualize the physical system as an aid before the physical implementation. As a result, VIPLE may appeal to a wide audience of users including high school students, college students, enthusiasts/hobbyists, researchers, as well as web developers and professional programmers.

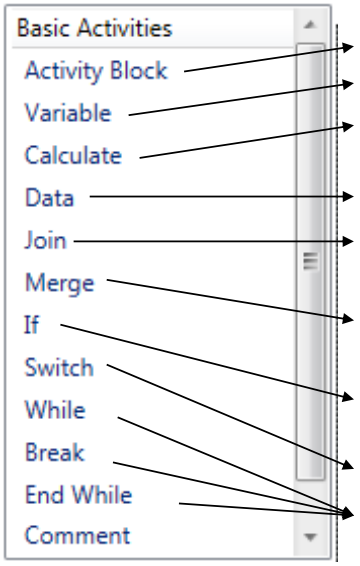
VIPLE is free and can be downloaded at:

http://neptune.fulton.ad.asu.edu/VIPLE/

You need to install .Net 4.5 Framework (or higher version) in order to run VIPLE. .Net 4.5 Framework can be downloaded from Microsoft site for free.

Now we start to learn some of the common tools that we will use in VIPLE. The "Basic Activities" toolbox window contains all of the common tools and components for forming dataflow and for creating data types and variables. The Basic Activities toolbox window contains a comment activity that allows developers to document their code. Below is a screenshot of the available components with a brief description of its purpose.

Basic Activities Toolbox Window



- **Activity**: for creating components
- **Variable**: supports basic types (Int32, Double, String, Boolean, etc.)
- **Calculate**: Calculate the value of typical expression that is supported by C++, Java
- **Data**: Introducing constant values in regular programming language
- **Join**: proceeds when all threads arrive; Can be used for parallel data or threads.
- **Merge**: proceeds when one of the data or threads arrives. It can be used for creating the return point of a loop;
- **If**: same as regular programming language construct; It allows multiple conditions.
- **Switch**: same as regular programming language construct;
- **While**: start a loop; **Break**: exits a loop, and **End While:** returns to While

Figure 2. VIPLE basic activities

The activities in Figure 2 are explained as follows.

Activity: an activity is used for creating new component, service, function, or other code modules. Simply drag and drop an Activity into the diagram, open it, and add in code into it to form a new component.

Activities can also include compositions of other of activities. This makes it possible to compose activities and reuse the composition as a building block. In this sense an application built in VIPLE is itself an activity.

An activity can be used in the current program only. An activity can be compiled into a service. A service created in one program can be based in another program.

Variable: a variable represents a memory location, where a program can store and retrieve values such as a literal string or a number.

Calculate: a calculate activity can be used to compute mathematical equations (add, subtract, divide, or multiply), as well as data extraction from other components such as from a variable or a textbox. It is similar to the assignment statement in C#, for example, x = 5 +7;

For numeric data operations, we use the following operation symbols:

+	add
-	subtract/minus
*	multiply
/	divide
%	modulo

For logical operators you can use the following symbols:

&&	AND (Both have the same meaning)

||,	OR	(Both have the same meaning)
!	NOT	(Both have the same meaning)

Data: the data activity is used to supply a simple data value to another activity or service. You can enter a value into the text. The type is automatically determined based on the value entered. VIPLE support all types in C# language. Below is a table commonly supported types.

VIPLE Type	Description
Boolean	Boolean values: true, false
Char	character
Double	double precision floating point number
Int32 (int)	32 bit signed integer
UInt32 (uint)	32 bit unsigned integer
String	character string (text)

Join: the join activity combines two (or more) data-flows (inputs). All data (inputs) from the incoming connections **must be received** before the activity can proceed to the next step. Join can be used to combine multiple inputs required by an activity.

Merge: the merge activity takes two (or more) data-flows (inputs). When the first data item arrives, the activity will proceed to the next step. No need to wait for the other data items to arrive. Merge can used to implement a loop. Merge is significantly different from Join. Merge waits for one input to arrive, while Join has to wait for all inputs to arrive.

If: The ***If*** – activity provides a choice of outputs to forward the incoming message based on a condition that was entered. If the condition is true, the first outgoing connection forwards on the incoming messages (and its data). However, if it is not true, then the else output is used. The ***If*** statement in VIPLE is similar to the if-statement in traditional languages such as Java and C#. The ***If*** activity in VIPLE can check multiple conditions in one activity block. In other words, it can combine multiple consecutive (C#) if-statements into one.

The conditional expression can use the following operators for evaluation:

= or ==	equals
!= or <>	not equals
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to

To add additional conditions to the activity, click the add (+) button on the activity block.

Switch: Similar to the "switch" statement in C#, the switch activity can be used to route messages based on the incoming messages matching the expression entered into the text box. To add additional Case branches (match conditions) to the activity block, simply click the Add (+) button on the activity block.

While: Similar to the "while" statement in C#, the while activity establishes a condition to forward the incoming message to a set of blocks, much like the If activity. The difference is that after the blocks are executed, the message and its data is then returned back to this while activity, and the condition is reevaluated. Ordinarily, and If activity would let the message continue on, but the while activity establishes a loop.

Break: This activity can be placed within a while loop and is used to prematurely exit the loop: i.e. leave the loop without the initial condition being evaluated as false.

End While: This activity marks the end of a while loop, and returns the incoming message back to the original While activity that began the loop.

Comment: the comment activity enables a user to add a block of text to the diagram as documentation.

In addition to the basic activities, VIPLE also provides a long list of built-in services for traditional input and output, as well as robotics specific services, such as sensor services, motor and drive services. Figure 3 shows a part of these services.

Figure 3. ASU VIPLE basic service, EV3 services, and Generic Services

We start to use the basic services to learn the language and start to use the robotics services later to implement our robotics project.

VIPLE menu

File

New – Allows you to create a new project.

Open – Opens a pre-existing project file.

Save – Saves the current project.

Save As – Saves the project under a specified name.

Print – Allows you to print your diagram for use in a report.

Exit – Quits VIPLE

Edit

Undo – Reverses the last edit action.

Redo – Restores the last undone action.

Run

Start – Start running the current project in the VIPLE environment.

Options

Lock Activities – Prevents the activity blocks from being moved

Help

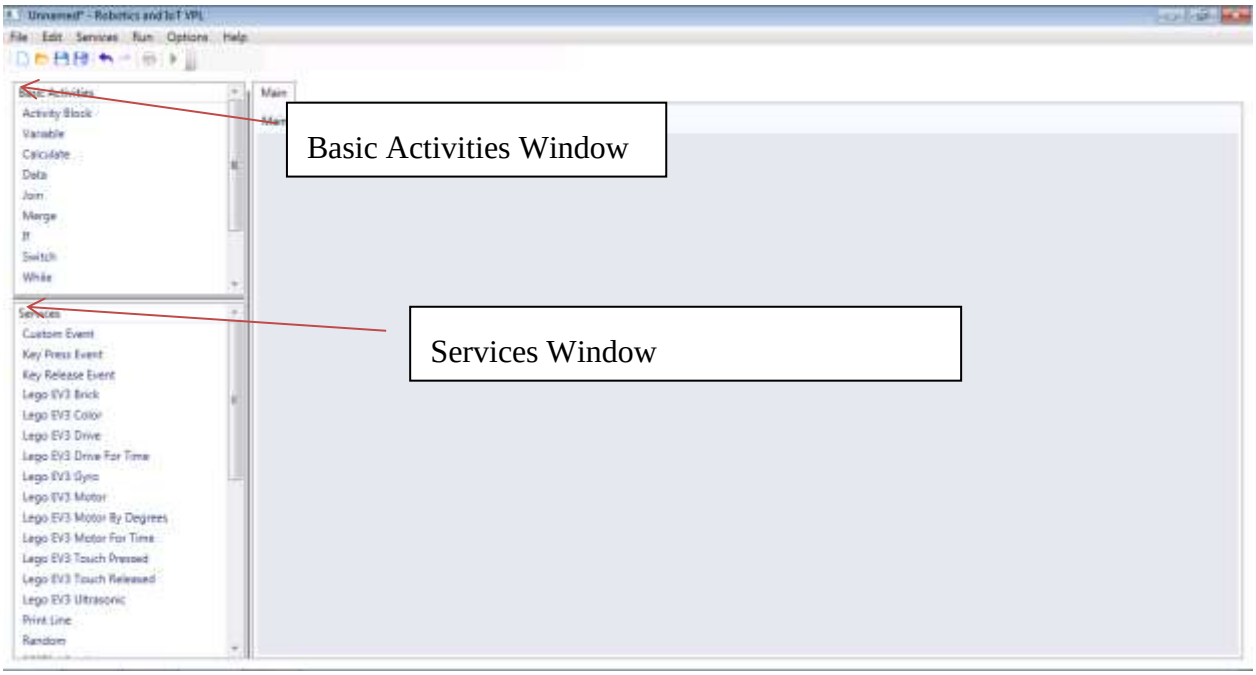
About RI-VIPLE – Displays the version of the software

Lab 1 Assignment

Lab assignment part is NOT required for preparing the pre-lab quiz. The work described here should be done in the lab as group exercises.

Getting Started

To get started, let us load up VIPLE. Please wait a minute or two while the operating system loads the application. When the application finishes loading, you will see the following screen.

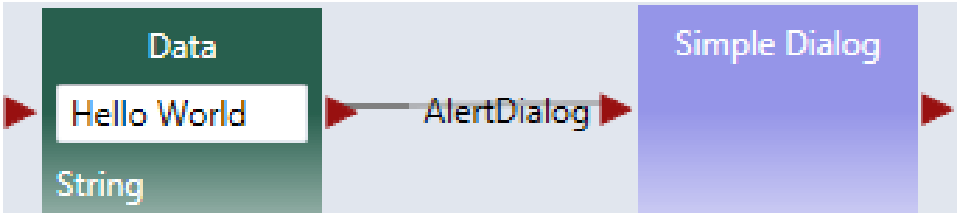


Now, your team can start to do the exercises in each of the following sections.

Exercise 1: "Hello World" in VIPLE

In this exercise, we will create our first application using VIPLE to display the words "Hello World."

1. Create a new project by selecting New from the File menu. Now insert (by drag-and-drop) a Data activity from the Basic Activities toolbox. Click in the text box of the Data block and type in "Hello World". The data type field will automatically display String.



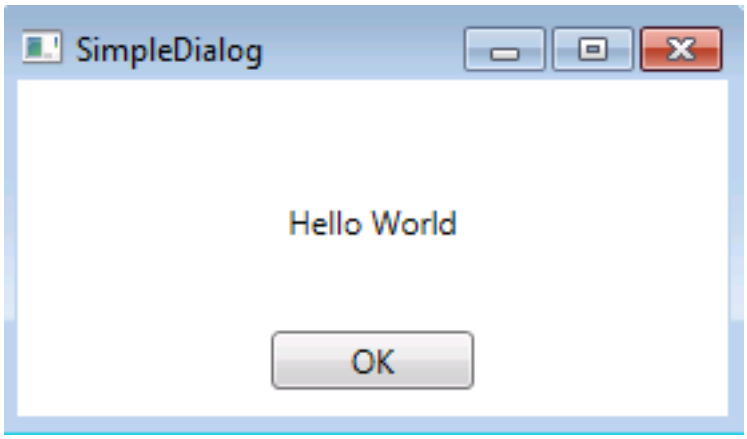
Question: In the data block, why did it display String, as opposed to Integer? (Hint: data type)

2. Insert a Simple Dialog activity, by dragging one from the services toolbox and place it to the right of the Data activity block.

3. Now drag a link starting from the output connection pin of the Data block onto the SimpleDialog block. The connections dialog box automatically opens. Select DataValue in the first list and Alert Dialog in the second list and then click OK.

Question: For each block there are normally two dots, one on each side of the block, what do they represent in terms of programming?

4. Save your program and name the project "Exercise_01". Then select the run command (or press F5) to run your program. Note, after you saved a program, you cannot double click the program to open. You need to open VIPLE and use its menu "open" to open the program. The reason for this is that we save the file as a text file, and it will be open as a text file if you double click it.



When you are done, please save the project file for later submission. At the end of the lab, you will put all the project files into a single zip file for submission.

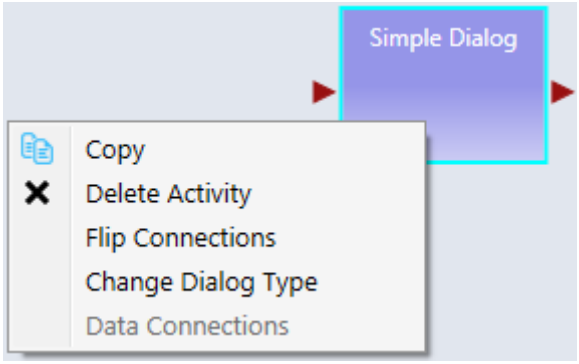
Notify your lab instructor for sign-off and change the driver.

Exercise 2: Favorite Movie

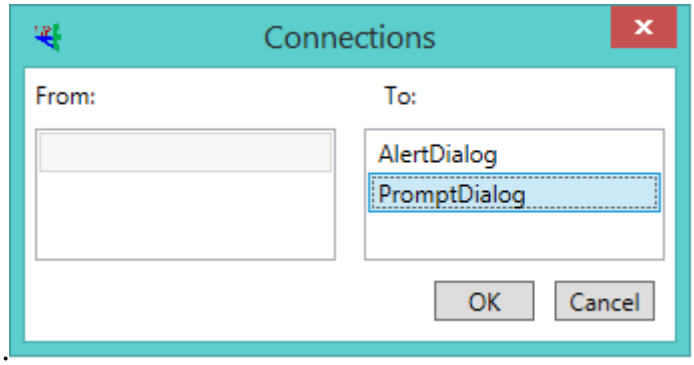
In the next exercise, we will create an application that will prompt the users for their favorite movie and respond back using the Text to Speech block.

1. First, create a **new** project by selecting New from the File menu. Click on the save button and save this project as "Exercise_02"

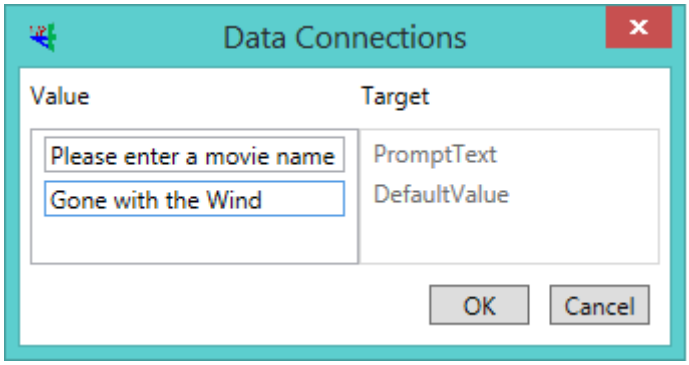
2. Now insert a Simple Dialog activity from the service toolbox. Right click the service and choose Change Dialog Type.



3. The Connections dialog box opens, Choose PromptDialog.



4. Another Data Connections dialog box opens. Type the texts as shown in the following dialog box, and then click OK. The first textbox is for the prompt test and the second textbox is for a default value.



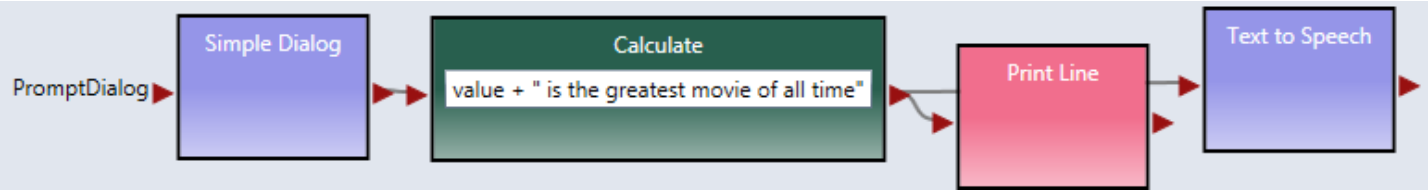
5. Now insert a Calculate activity (from the basic activities toolbox) into the diagram.

6. Create a link that connects the Simple Dialog block with the Calculate activity. The connection dialog box automatically opens.

7. Next, type the following text into the Calculate block: value + " is the greatest movie of all time!". You can achieve this by first clicking on the blank of the Calculate block, and then a dropdown list will appear. Choose "value" using either mouse or arrow keys. Finish up by typing in all strings left.

8. Next insert a Text to Speech block and a Print Line into the diagram and connect the link between the Calculate block and the two output blocks.

11. Your final diagram should look something like the screenshot below. When you are ready, run the application by clicking on the run button.

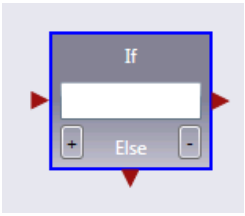


Tips: If you accidentally close the connections or data connections windows during the process or you simply want to make a change to your previous configuration, you can always reopen them by right clicking your mouse on the blocks, choosing connections or data connections to retrieve the windows you want.

Question: How do we change the text the computer is speaking? (Hint: What does the Calculate block do?)

If-Statement

The if-statement is inherited from the traditional programming paradigm. The if-statement is also known as a conditional statement. In VIPLE, the if-statement is designated by the block below.



The if-section of the statement or statement block is executed when the condition is true; if it's false, control goes to the else statement or statement block. Within the if condition, you can also apply the conditional OR (||) and conditional AND (&&) operators to combine more than one condition. Unlike most traditional programming languages, VIPLE allows the program to branch into more than just two ways using if block. To add more branches, just simply click on the "+" sign next to the else branch, type in the condition you want to check. VIPLE will evaluate the conditions in order, and your program will go into the first branch whose condition is evaluated to be true.

When you are done, please save the project file for later submission. At the end of the lab, you will put all the project files into a single zip file for submission.

Notify your lab instructor for sign-off and change the driver.

While-Loop

The while-loop also falls in the conditional loop category. The while-loop statement executes until the condition is false. It is also a pretest loop, which means it first tests if a condition is true and only continues execution if it is.

Question: Why do we need to learn how to use While-Loop?

When you are done, please save the project file for later submission. At the end of the lab, you will put all the project files into a single zip file for submission.

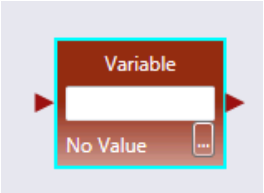
Notify your lab instructor for sign-off and change the driver.

Exercise 3: Create While Loop Block using Merge and If Activities

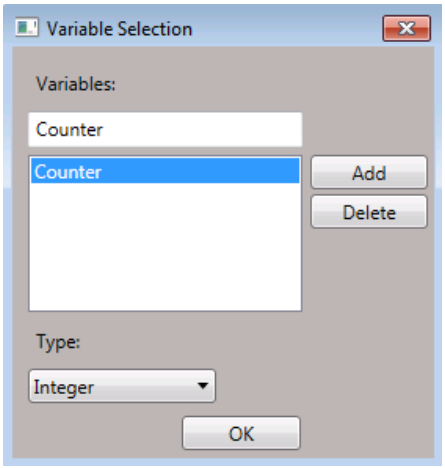
This exercise will introduce the use of While-Loop in a VIPLE program. It creates a variable, initializes it, and then counts to ten, speaking the value on each of the iteration using TTS Activity Block.

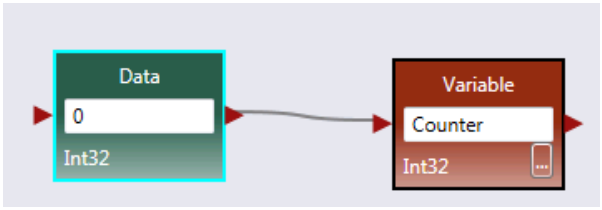
1. To begin, create a **new** project by selecting New from the File menu. Then next save the project as "Exercise_03".

2. Now, insert a Variable activity from the toolbox.



3. Next, click "…" to define the variable. In the dialog box that opens, click in the Name textbox and type in Counter as the name for your variable. Then click on the Add button. Then click on the dropdown arrow under "Type:" and select Integer from the dropdown list as the type for the variable. Click OK.

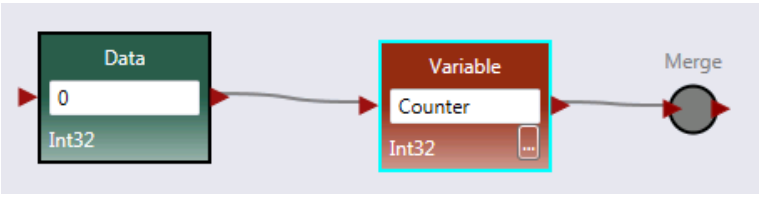




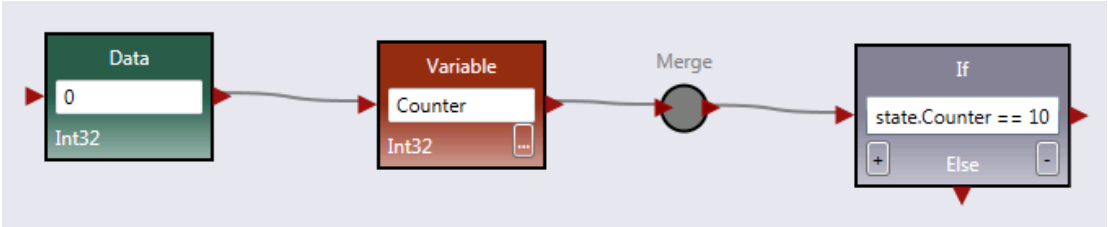
4. Now add a Data block to your diagram to the left of the Variable block and connect them by creating a link connection from the Data activity to the Variable activity block.

5. Enter 0 into the text box of the Data block, and the data type Int32 should automatically fill in. This sets both the data and its type. Its connection will then initialize the Counter to 0.

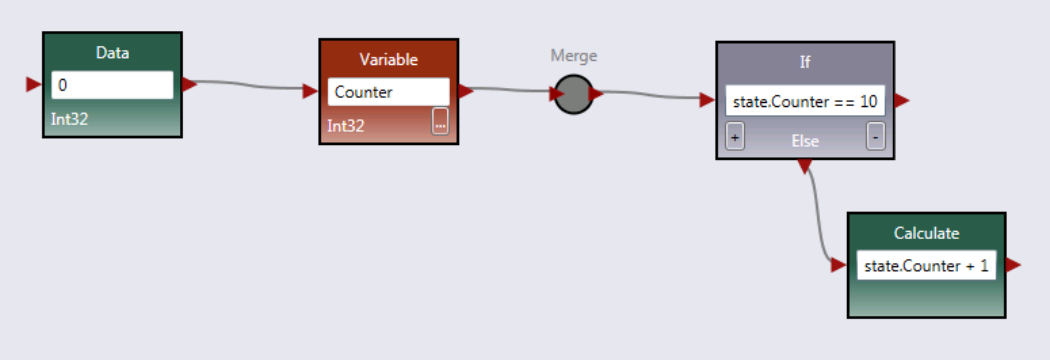
6. Now insert a Merge block to the right of your Variable block and connect the Variable block to the Merge block. The block is used to create a counting loop. *Remember: A Merge block can have multiple inputs; each input is passed on as it is received.



7. Next, add an If activity to the diagram to the right of the Merge block. Connect the Merge block with the If-Statement block. In the If-Statement block, enter state.Counter == 10.

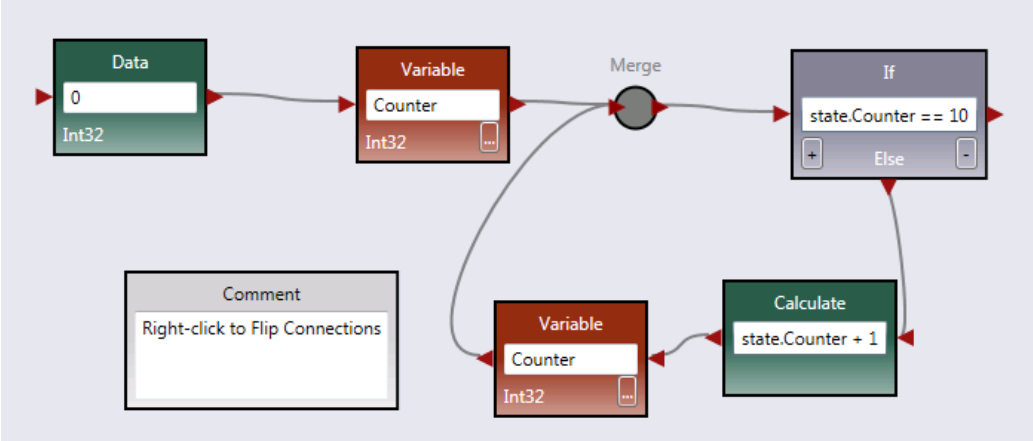


8. Now add a Calculate block and connect it to the Else connection of the If-Statement block. In the Calculate block, enter "state.Counter + 1" into its expression.



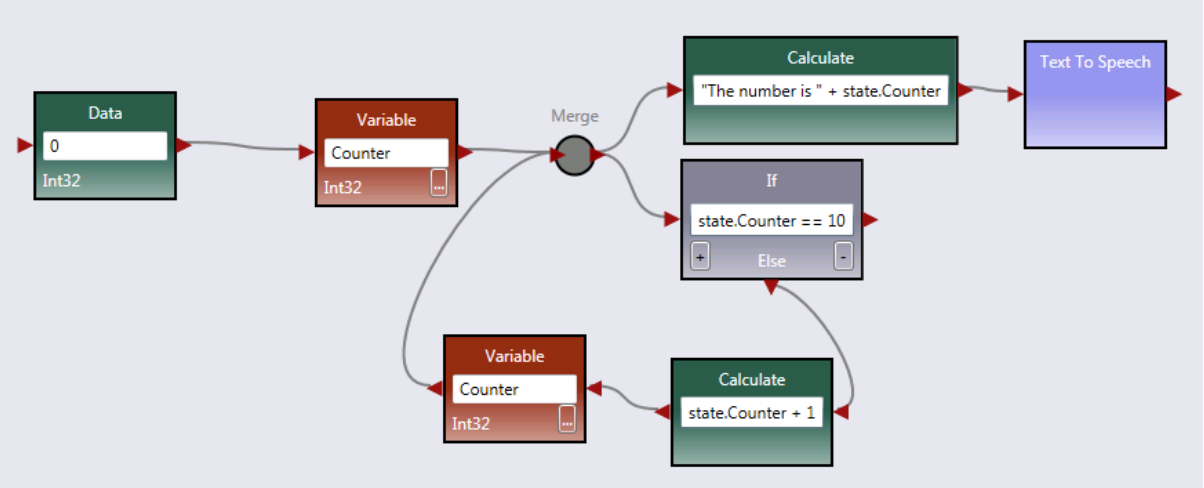
9. Now insert another variable into diagram. This new variable will take the value of Counter after it is re-calculated. Click the "…" button, and select the Counter variable. Then click OK.

10. Now connect the output connection pin of the Variable block to the Merge block. This completes your loop.



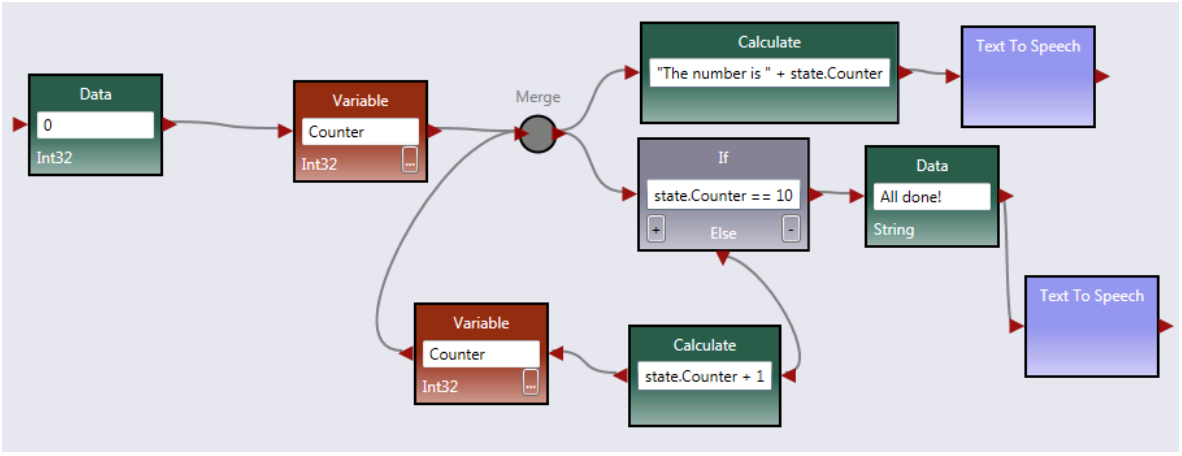
11. The next step is to insert the TTS block into the diagram to allow the program to speak out the result as it increments. So add another Calculate block into the diagram. Inside the new Calculated block enter: "The number is" + state.Counter

12. Then add a Text to Speech block and connect it to the output of the Calculate activity block.



13. To complete the program, add a Data block after the If-Statement bock and connect the If-Statement with the Data block. Within the Data block, enter "All Done!".

14. Add another TTS block and connect it with the Data block. This will allow the program to say "All Done!" when the program reaches a count of 10.



15. When you have completed the program, run your program by clicking on the Run button (or F5 on your keyboard).

When you are done, please save the project file for later submission. At the end of the lab, you will put all the project files into a single zip file for submission.

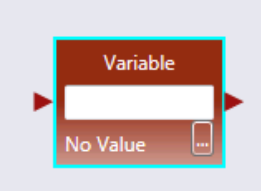
Notify your lab instructor for sign-off and change the driver.

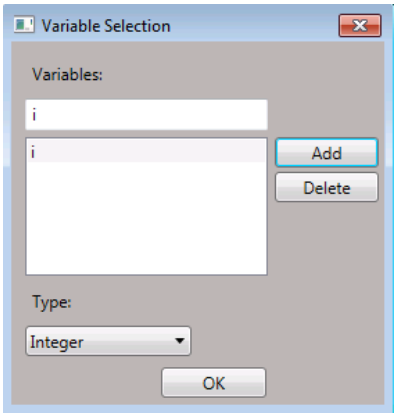
Exercise 4: Creating While-Loop Block using the While Activity

This exercise will introduce the use of While-Loop in a VIPLE program. It creates a variable, initializes it, and then counts to ten, speaking the value on each of the iteration using TTS Activity Block.

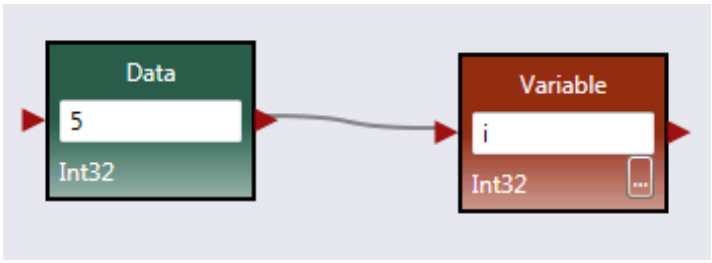
1. To begin, create a **new** project by selecting New from the File menu. Then next save the project as "Exercise_04".

2. Now, insert a Variable activity from the toolbox.





3. Next, click "…" to define the variable. In the dialog box that opens, click in the Name textbox and type in 'i' as the name for your variable. Then click on the Add button. Then click on the dropdown arrow under "Type:" and select Integer from the dropdown list as the type for the variable. Click OK.

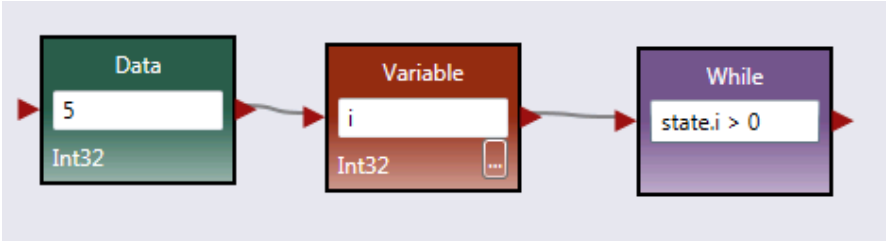


4. Now add a Data block to your diagram to the left of the Variable block and connect them by creating a link connection from the Data activity to the Variable activity block.

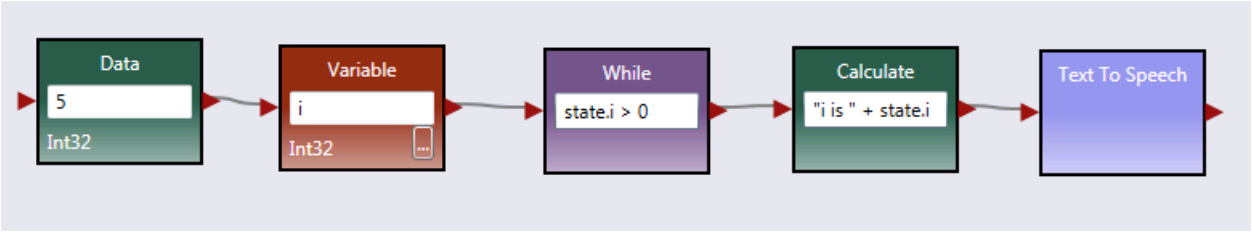
5. Enter 5 into the text box of the Data block, and the data type Int32 should automatically fill in. This sets both the data and its type. Its connection will then initialize 'i' to 5.

6. Now insert a While block to the right of the Variable block and connect the Variable block to the While block.

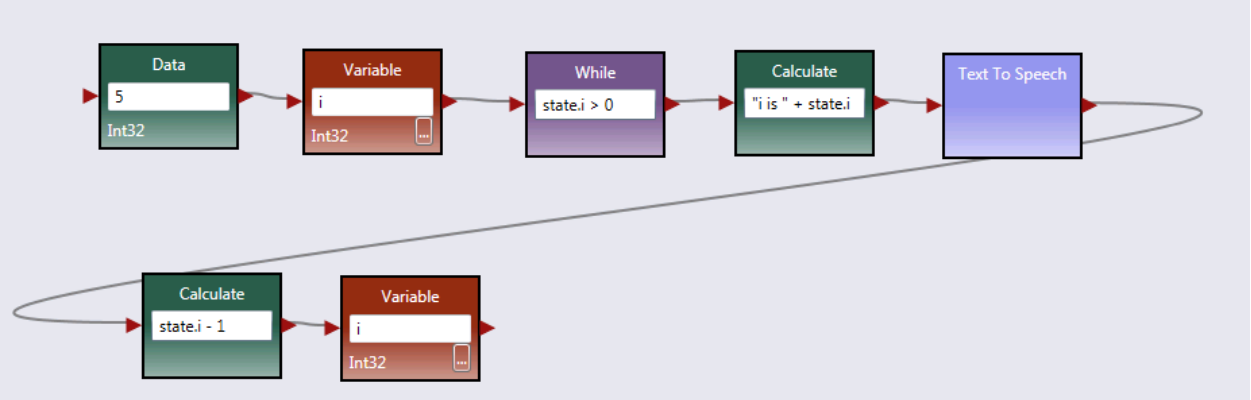
7. In the expression field for the While block, enter "state.i > 0".



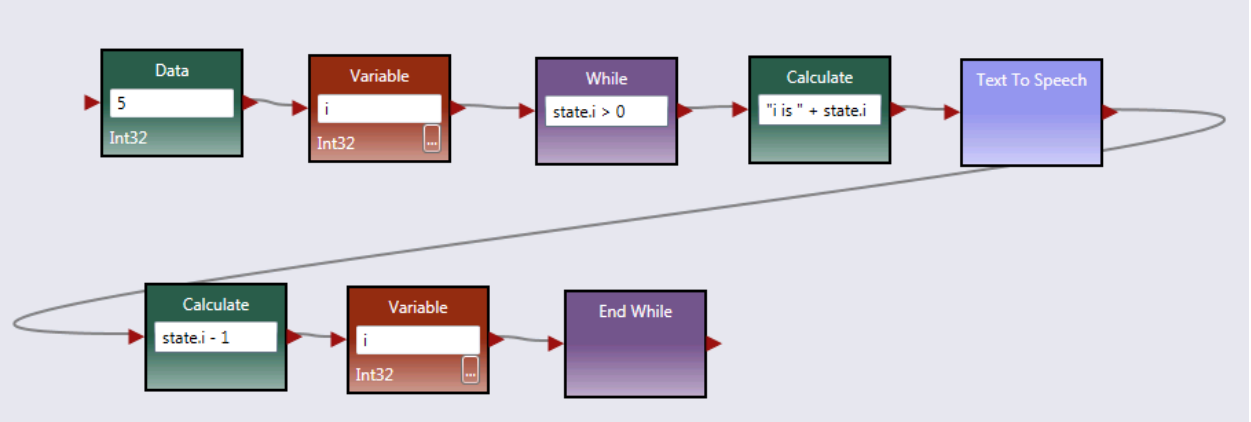
8. Now add a Calculate block and connect it to the While block. In the Calculate block, enter "i is " + state.i into its expression. Add a Text to Speech block and connect it to the output of the Calculate block.



9. Now insert another calculate block into the diagram. In its expression field, write "state.i – 1". Insert a variable block after this calculate block. This new variable will take the value of i after it is re-calculated. Click the "…" button, and select the i variable. Then click OK.

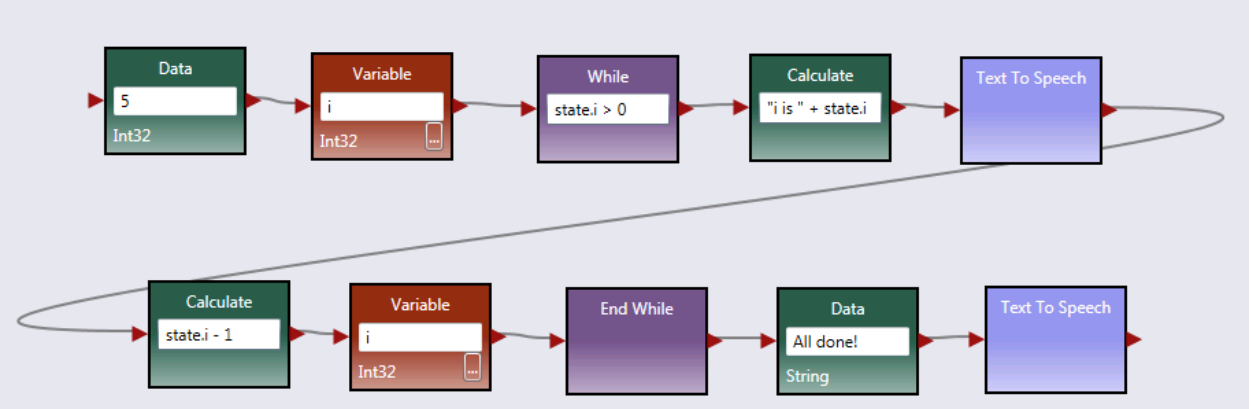


10. Now connect the output connection pin of the Variable block to an End While block. This completes your loop.



11. The next step is to insert the TTS block into the diagram to allow the program to speak out when the result is complete. So add a Data block into the diagram. Inside the new data block enter: "All done!"

12. Then add a Text to Speech block and connect it to the output of the Data activity block.



13. When you have completed the program, run your program by clicking on the Run button (or F5 on your keyboard).

When you are done, please save the project file for later submission. At the end of the lab, you will put all the project files into a single zip file for submission.

Notify your lab instructor for sign-off and change the driver.

Exercise 5: Creating Counter Activity

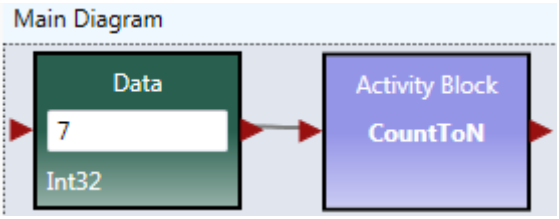
Question: Do you remember some of the reasons why we prefer using services/activities in program?

In this exercise, we will create an activity to modularize the while loop that was created a previous exercise.

Step 1. Create a new project and save this project as "Exercise_05"

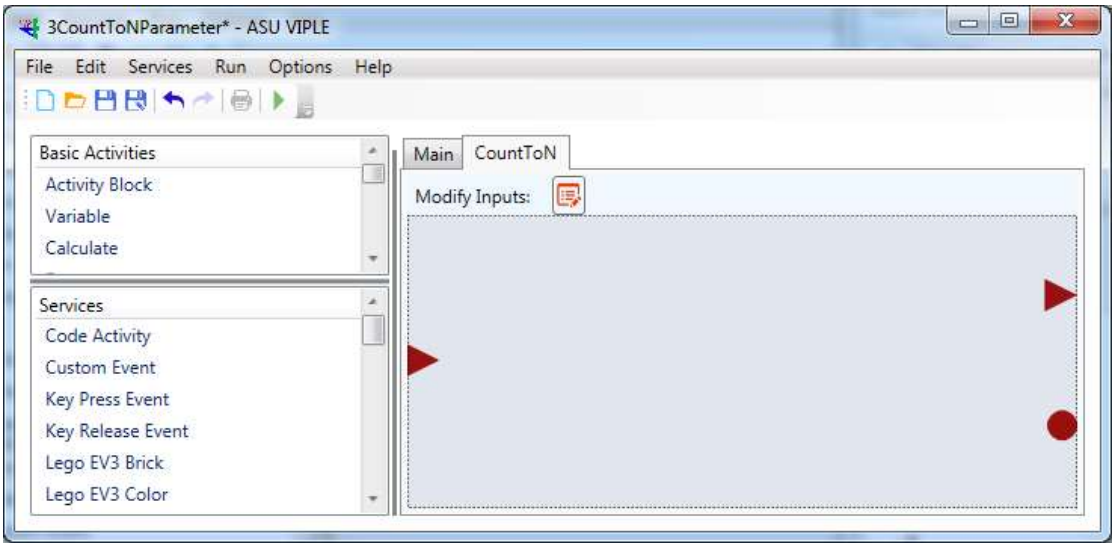
Step 2. Insert a Data activity and set the value of your new Data activity to 7 (N = 7).

Step 3. Now create a new activity by inserting an Activity block on your diagram.

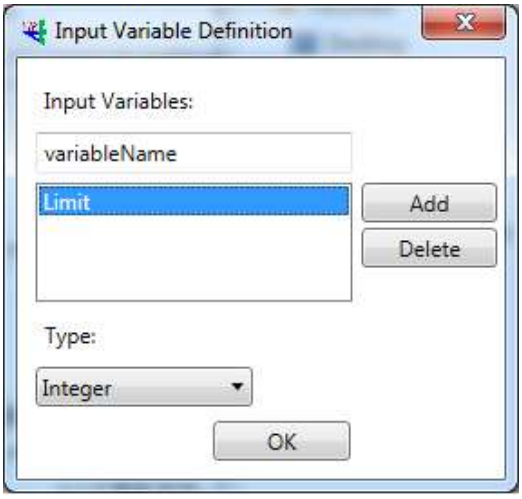


Right-click the block and rename it CountToN.

Then open it (Double click on CountToN). A new tab page appears. The triangle dots on the left side and on the right side represent data input and output, respectively. The circular dot on the right side represents an event output. We will discuss event-driven programming later.

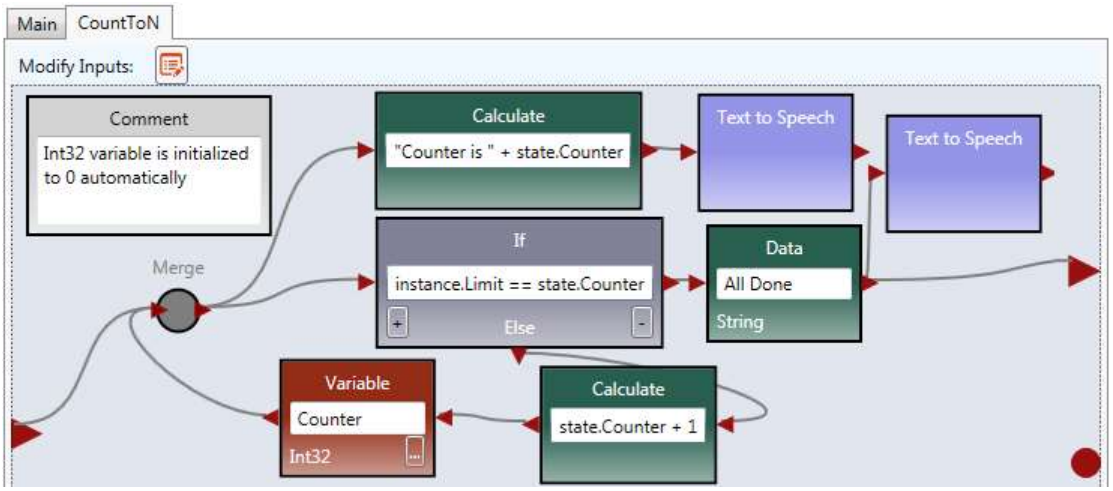


Step 4. Click the orange symbol  underneath the CountToN tab to open the window below for defining the input of the activity.



Step 5. We need one input value. We name it Limit, which will be used as the upper value for counting. Change the name of the input value to Limit and select Integer as its type. Then, click OK.

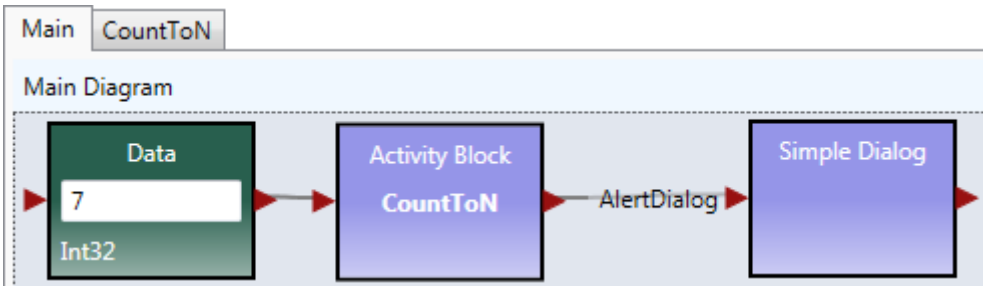
Step 6. For your activity's internal dataflow, insert the activity blocks you built in the previous counter exercise.



We use instance.Limit to access the input value from outside the activity.

Step 7. Close the activity page or click the Diagram tab to switch back to the main dataflow page (main diagram). Now connect the output of the Data activity to your new CountToN activity. The Data Connections window will open, type in "value" for the input to StartVariable.

Step 8. Finally, insert a SimpleDialog block into the main diagram and connect the output of your CountToN activity to it.



Step 9. When you are ready, go ahead and run the program. Try experimenting with a different data value other than 7.

When you are done, please notify your lab instructor and demonstrate your program for sign-off. Then change the driver to proceed to the next assignment.

Lab 2
Event-Driven Programming and Finite State Machine

Overview

In this lab, you will learn finite state machine and VIPLE's event-driven programming mechanisms. You will use them to implement various control programs based finite state machines.

Lab 2 Preparation

You need to read the preparation part carefully. The lab assignment part of this manual will be completed in the lab as a team assignment.

1. Introduction

The primary task of a robotics application is to process sensory inputs from a variety of sources and utilize a set of actuators (motors) to respond to these sensory inputs in a manner that achieves the purpose of the application. Figure 1 shows an robotics application. It has different types of sensors that feed data into the robotics application, which then controls different motors to perform different actions. Note: a robotics application can have multiple inputs (from different sensors), which are processed concurrently to generate an outcome the robot can respond to. As sensory inputs can arrive at any time and they can be considered to events. The even-triggered actions are best modeled by a finite state machine.

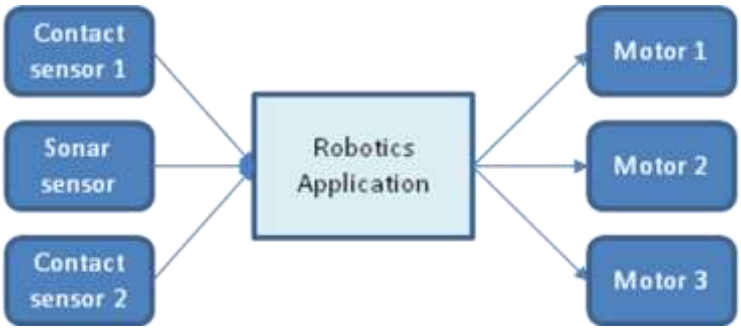


Figure 1: A robotics application has multiple inputs and they must be processed concurrently

2. Event-Driven Programming

Event-driven computing is a programming model, in which the flow of the program is determined by events (or called notifications), such as user actions (e.g., mouse clicks and key presses), sensor inputs/outputs, or messages from other threads that can occur at any time during the program execution. Event-driven computing assumes multiple computing flows (threads) exist to process data and events that are ready to be processed in parallel. Event-driven computing is based on parallel computing. On the other hand, the control flow based computing model determines its flow based on the inputs to the program. It assumes that there exists one flow only. The data that are ready to be process have to wait still the control flow arrives in order to be processed.

Consider the assembly work of robot. If there is one person to do all the jobs, we must list the jobs in certain order and get the jobs done follow the order. This is the flow control model. On the other hand, if there are multiple persons to do the jobs, some jobs can be done at the same time. The ways data are process in control flow model and event-driven model are also different. In the control flow model, the data will be lined up and processed in the order, while event-driven model does not have the lining up mechanism. As an analogy,

consider the people arriving at a bus station and waiting for a bus. The control flow model will require the people to line up waiting for the bus. In this way, people arrive first will be guaranteed to enter the bus first. On the other hand, the event-driven model will not require the people to line up. As the results, people arrive at the bus station first will not be guaranteed to enter the bus first. The control flow model works better if there is only one bus can arrive at the same time. If multiple busses can arrive at the same time and the bus capacity is greater than the number of people, the event-driven model is more efficient.

A related concept to programming model is that programming paradigm, which describes how computation is express in programing. Programming paradigms include imperative programming, functional programming, logic programming, object-oriented programming, service-oriented programming, real-time programming, and parallel programming. Almost all programming languages conveniently support flow control model, but require complex programming in order to write event-driven programs. Most programming languages support multiple programming paradigms. For example, Java and C# support both flow control and event-driven models, and they conveniently support imperative, object-oriented, and service-oriented programming paradigms. With the support of library functions, they also support parallel and event-driven programming.

Robotics applications are often event-driven, that is, the program must react to the arrivals of events. Handling sensory inputs and controlling motors (actuators) must be dealt with concurrently (in parallel). Otherwise the actuators may starve.

VIPLE conveniently supports both control flow model and event-driven model. It supports object-oriented programming using the activities to encapsulate the details into reusable components. It supports service-oriented programming by allowing activities to be converted into services. It can also call the remote services using the RESTful service in the service list.

In the previous labs, we have learned using VIPLE to program in control flow model. In this lab, we will use event-driven model to write programs. An event output can be used alone to notify specific events, for example, the completion of an action. It can also be used as an additional output with a regular return value. Event, with the regular return value can give an additional signal to the user activity, as shown in the Figure 2.

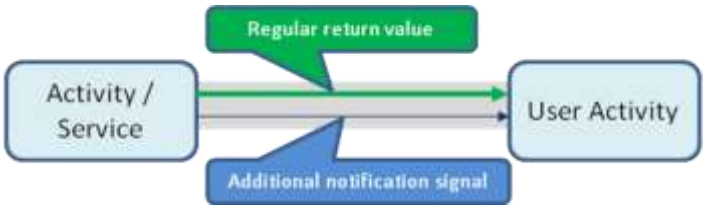


Figure 2: Notification option contains an additional signal

The additional event signal is necessary if the activity (service) is to be used multiple times, e.g., in a loop. Without the notification signal, the user activity cannot determine if a new value has arrived if the new value happens to be the same as the previous value.

When an event arrives, the user activity will know a new value has arrived, no matter if the value happens to be same as the previous value or not, as shown in Figure 3.

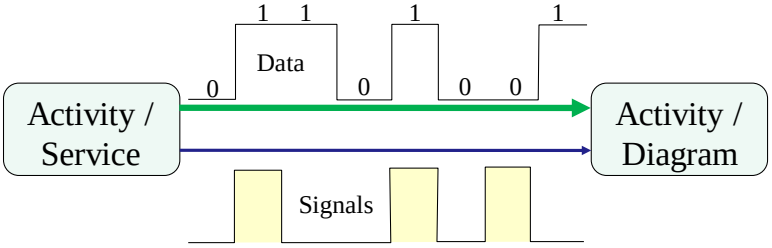


Figure 3: The return values may remain unchanged. The notification is necessary to signify the arrival of the next value

Figure 4 shows the event-driven counter activity. It defines an event by connecting the counter output value to the circular (event) output, instead of the triangular (data) output, which generates an event every time the merge activity generate a new value. This event can be used to trigger the execution of another activity. An activity generate both data output and event output.

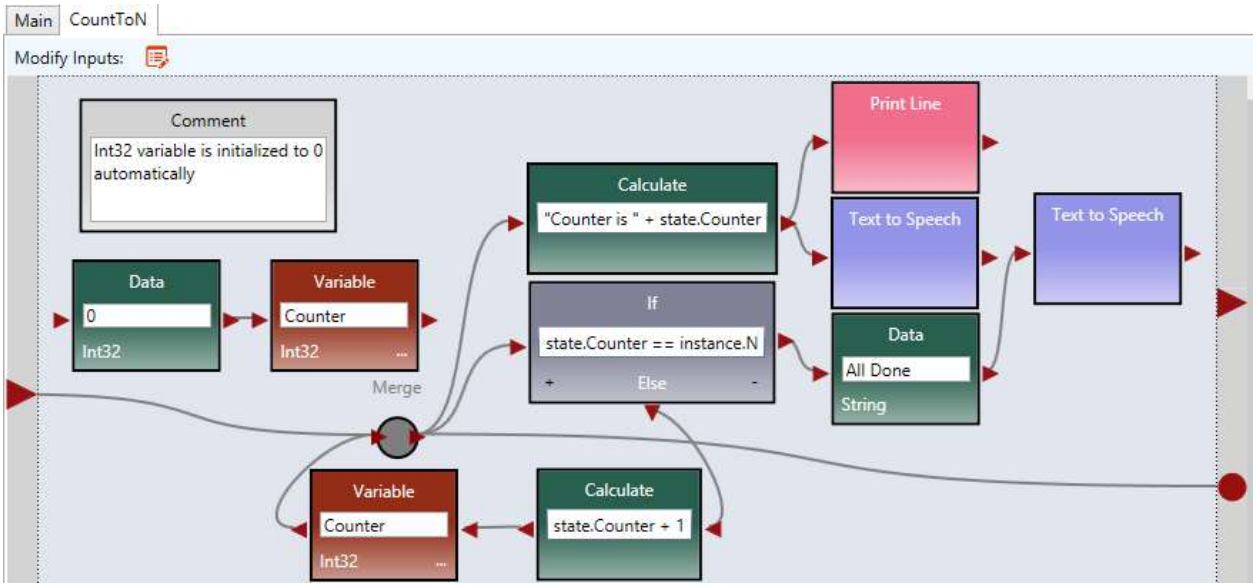


Figure 4: The activity generating an event output

In the program, we defined a parameter N to take the input value. To define input parameters, click "Modify inputs:" button in the activity window.

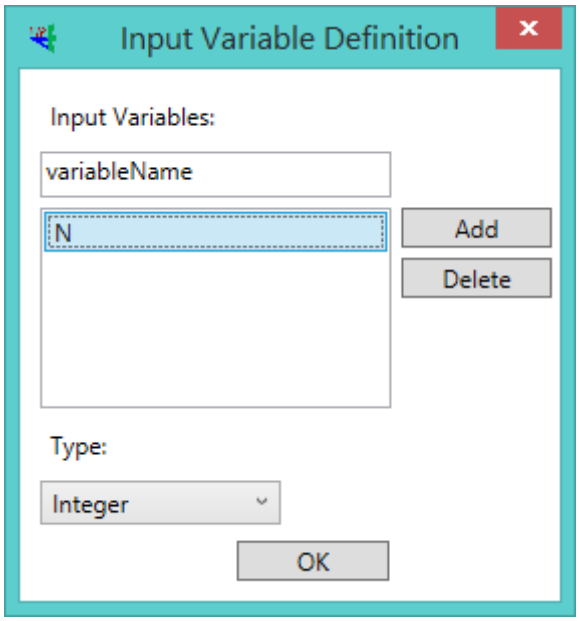


Figure 5: Define a parameter for an activity

Once an event output is defined, the event will be available in the "Custom Event" service, as shown in Figure 6. VIPLE has two types of events: built-in events and custom events. The built-in events include Key Press, Key Release Event, and the senors that generate event output. Each activity in which an event is defined will be added into the Custon Event set.

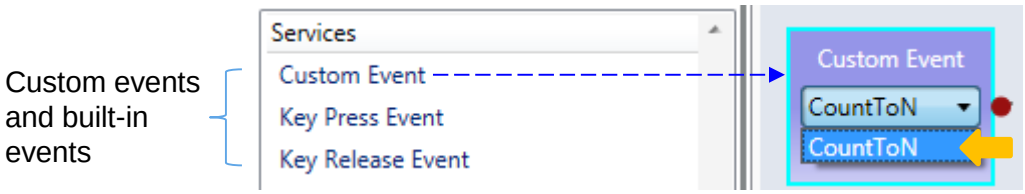


Figure 6: VIPLE built-in events and custom events

Creating events in your activities for other activities to use is one part of event-driven programming. Using the event output as input in your program is the other part of the event-driven programming. Figure 6 shows using the event in the diagram for generating the test cases. Notice that the CountToN activity is not connected to the Print Line service.

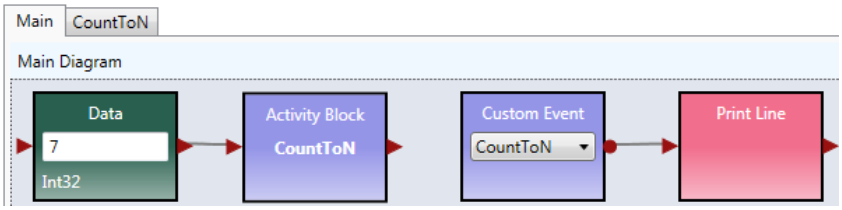
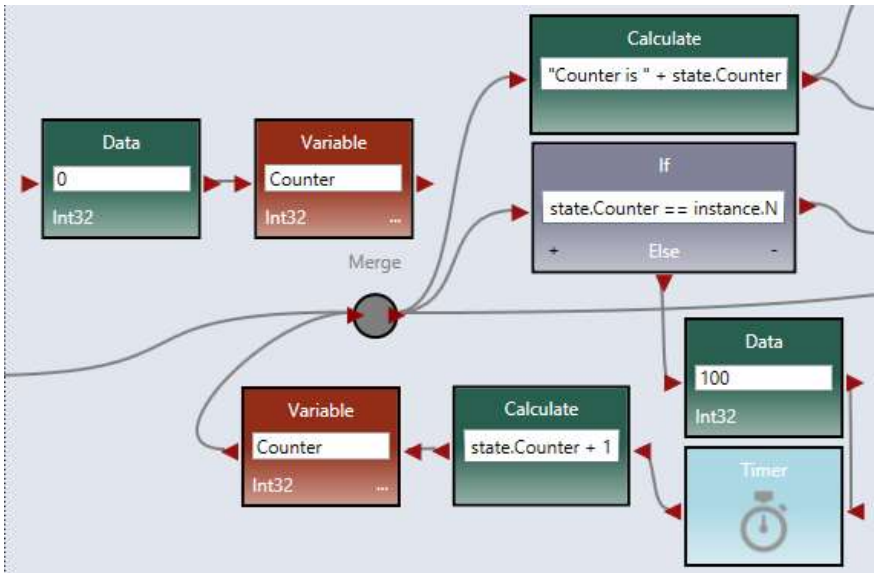


Figure 6: The activity generating a notification output

The numbers may be generated too fast for the Text to Speech to read. You can add a delay into the Counter activity to slow down the number generation process, as shown in the following diagram.



The event-driven programming model is best described by finite state machines consisting of states and transitions between the states. The transitions are triggered by events.

3. Finite State Machine

A **finite state machine** (FSM) diagram, also called a **state diagram** or **state transition diagram**, is a model of behavior composed of a finite number of **states**, **transitions** between those states, and **actions**. A finite state machine is an abstract model of a machine with internal memory [http://en.wikipedia.org/wiki/Finite_state_machine]. A current state is determined by past states of the system. As such, a finite state machine can record information about the past, i.e., it reflects the input changes from when the system starts to the present moment. A transition indicates a state change and is described by a condition that would need to be fulfilled to enable the transition. An action is a description of an activity that is to be performed at a given moment. The FSM that we discussed here is a **deterministic finite state machine**. It accepts finite strings of symbols as input (http://en.wikipedia.org/wiki/Deterministic_finite-state_machine).

A FSM can be used to specify (to model) different systems in which memory (variables) are involved, such as

- a vending machine that takes a sequence of coins or bills as inputs;
- a parity (even or odd) checker that counts the number of 1s in a binary number (a string of 0s and 1s);
- a string filter (processor) that reads the string as a sequence of characters and processes the characters in the process of reading; and

- a robot's behavior, for example, using the information of its current and past states to determine the next action.

In a later course, you will learn that there are two kinds of FSMs: (1) Pure FSM that uses states as the only memory, and it does not use additional variable; (2) FSM that uses additional variables. First, let us take a look of a pure FSM.

We assume that a vending machine sells soda and has the following four inputs (events):

- Deposit quarter (25)
- Deposit dollar (100)
- Push button to get soda (soda)
- Push return button to get money back

Because all information must be stored in a finite number of states, we have to assume a maximum amount that the machine can hold. For example, we can assume that the machine can hold at most 100 cents ($1 or four quarters). The finite state machine that specifies the state, state transitions, input, and output is given in the Figure 7.

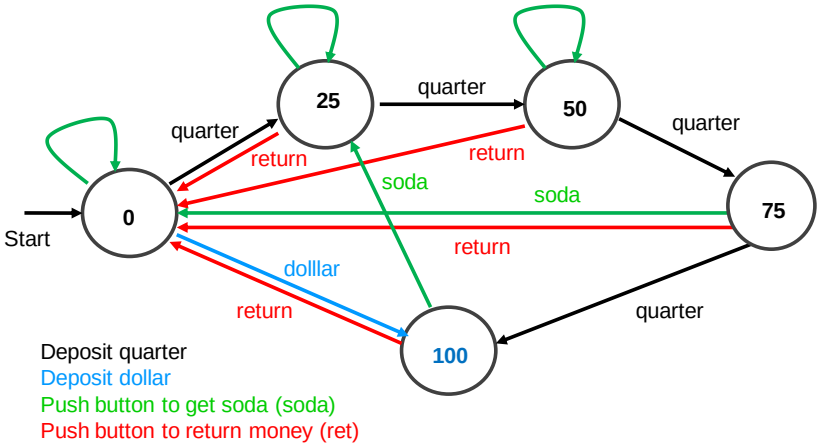


Figure 7: Pure deterministic finite state machine

If we can use additional memory, we can eliminate the upper bound of the amount that can be accepted, and also reduce the number of states. Figure 8 shows the FSM with an additional variable "Sum", which keeps track of the total amount deposited into the vending machine.

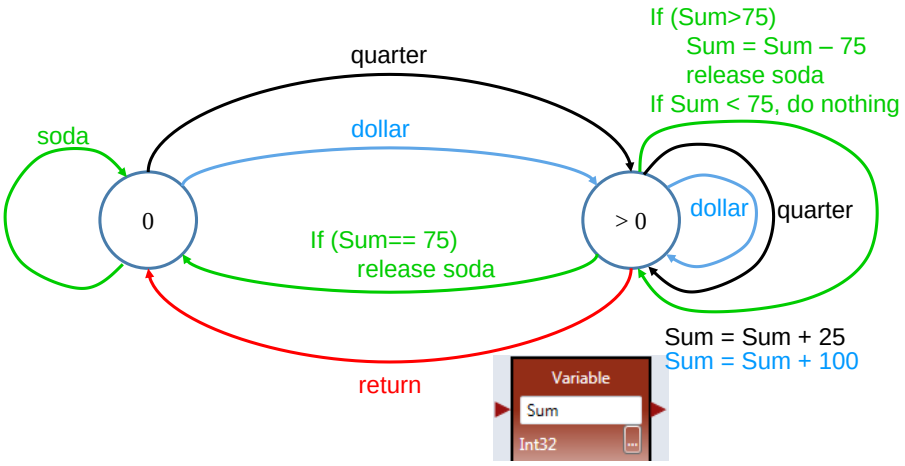


Figure 8: A deterministic finite state machine with additional memory

Lab 2 Assignment

Although ASU VIPLE can be used as a general programming language, as we have used in the previous labs, its strength is in event-driven programming that can response to a sequence of events. The event-driven applications are best described by finite state machines consisting of states and transitions between the states. The transitions are triggered by events. We will start to use ASU VIPLE for solving event-driven problems.

Exercise 1: Building an event-driven counter

Follow Figures 4, 5, and 6 in the preparation part to build an event-driven counter. Compare and contrast the counter that you built in the previous lab: what remains the same and what has been changed?

When you are done, please notify your lab instructor and demonstrate your program for sign-off. Then change the driver to proceed to the next assignment.

Exercise 2: Implement a Vending Machine

Given a Finite State Machine (FSM) in Figure 8, we first implement the requirement of the vending machine without using event-driven programming.

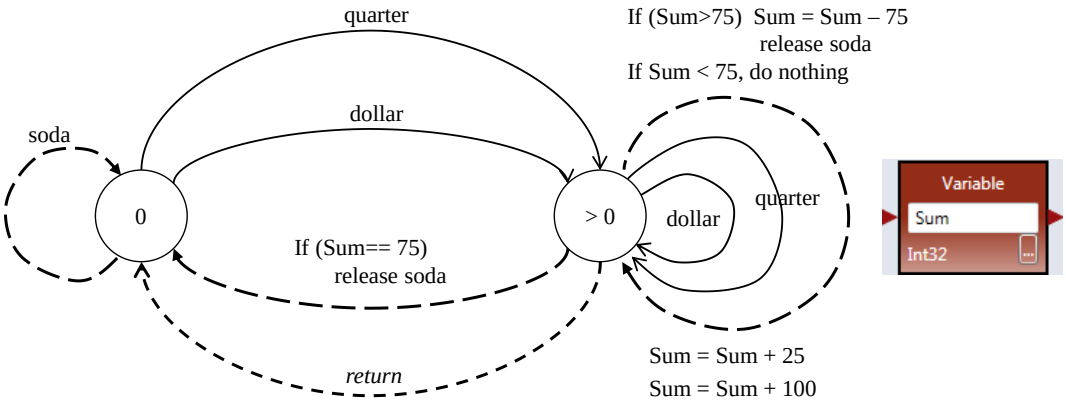


Figure 8. Finite State Machine of a vending machine

A sample VIPLE diagram is shown in Figure 9. We use Simple Dialog to take input. The user can enter one of the input values: quarter, dollar, return, and soda. Based on the input value, the program generates the required output.

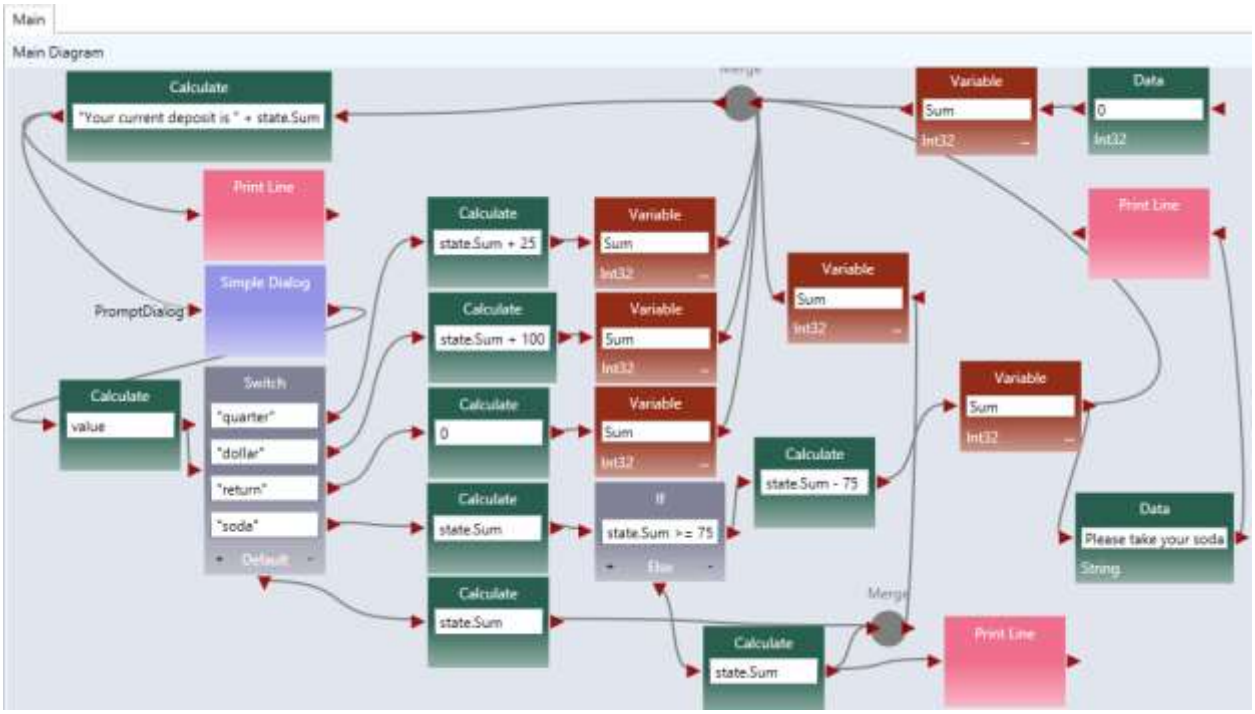


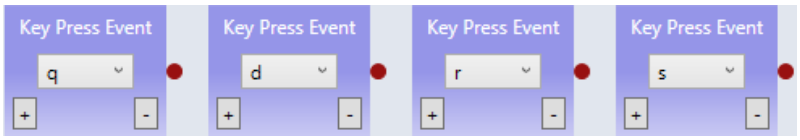
Figure 9. ASU VIPLE program of the vending machine

When you are done, please notify your lab instructor and demonstrate your program for sign-off. Then change the driver to proceed to the next assignment.

Exercise 3: Implement the Vending Machine Using Events

Change the program from using SimpleDialog to using the Key Press Event as the inputs (triggers) of the finite state machine. Use the keys

- q for quarter
- d for dollar
- r for return
- s for soda



You can connect each Key Press Event to a Calculate activity without using the Simple Dialog and the Switch.

Test you program and make sure it works as expected.

When you are done, please notify your lab instructor and demonstrate your program for sign-off. Then change the driver to proceed to the next assignment.

Exercise 4. Garage Door Opener (part 1 and part 2)

Given a Finite State Machine (FSM) in Figure 10, implement a simulated garage door control logic in ASU VIPLE.

At this time, ASU VIPLE has not been connected to sensors and actuators (motors), we will use key press event and Print Line to simulate the sensors and motors.

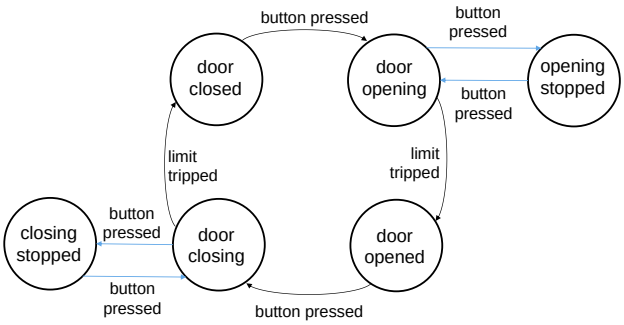


Figure 10. Finite state machine of a garage door opener

The remote controller is a touch sensor or a key press event, and the limit sensor is a build-in sensor in the motor. When the door stops, the limit sensor will generate a notification. You can start your program similar to the maze navigation program we learned in Microsoft VPL, as shown in Figure 11. In the diagram, Ctrl key is used to simulate the remote controller. You may use another key, e.g., the "Space" key.

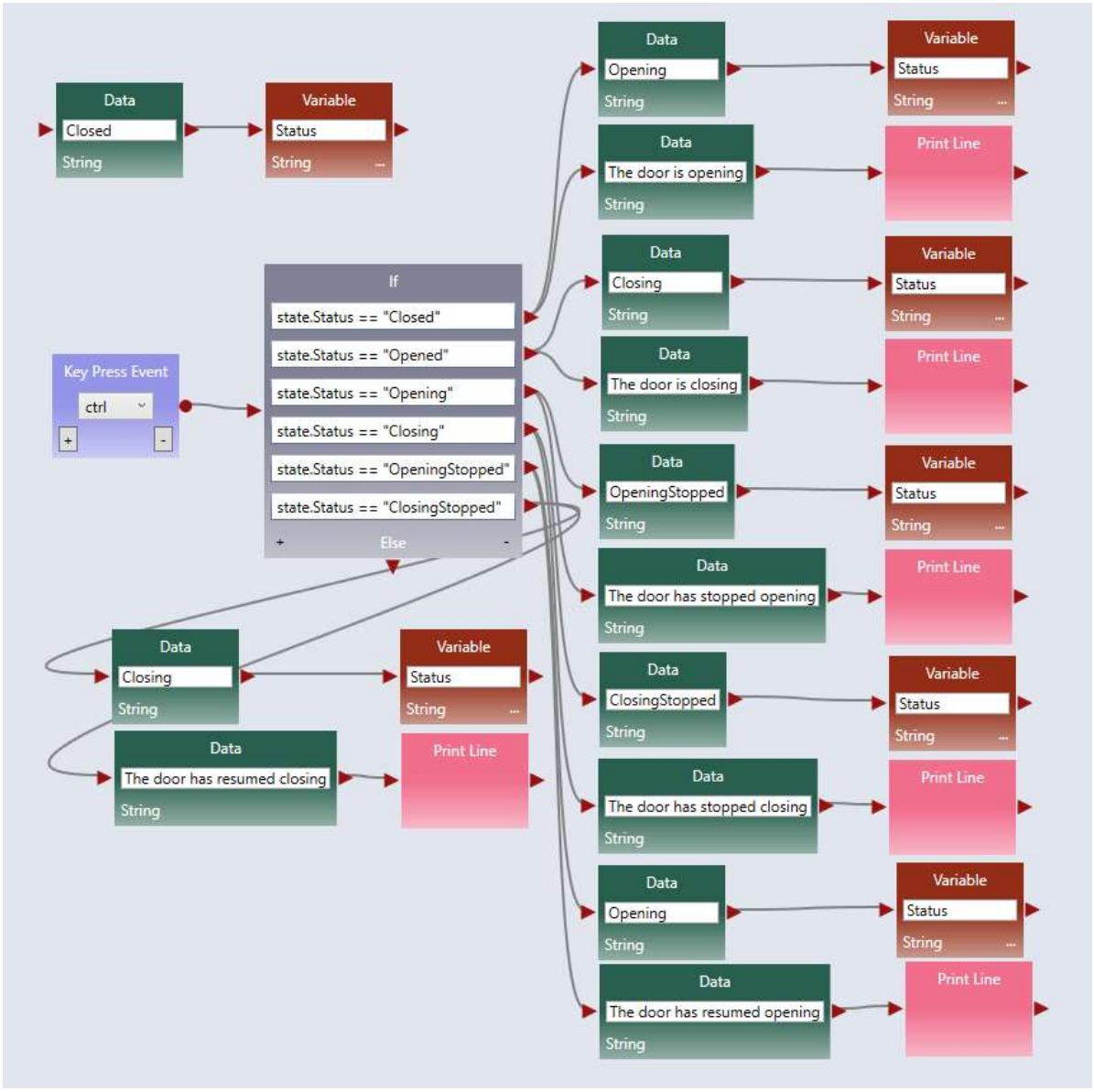


Figure 11. ASU VIPLE program (part 1) of the garage door opener

Test you program and make sure it works as expected.

When you are done, please notify your lab instructor and demonstrate your program for sign-off. Then change the driver to proceed to the next assignment.

The diagram implemented in Figure 11 is incomplete. The limit sensor built-in the motor can be simulated using a Timer event. After the ctrl key is pressed, the timer starts to count for 3 seconds. The state will change after the timer expires, as shown in Figure 12.

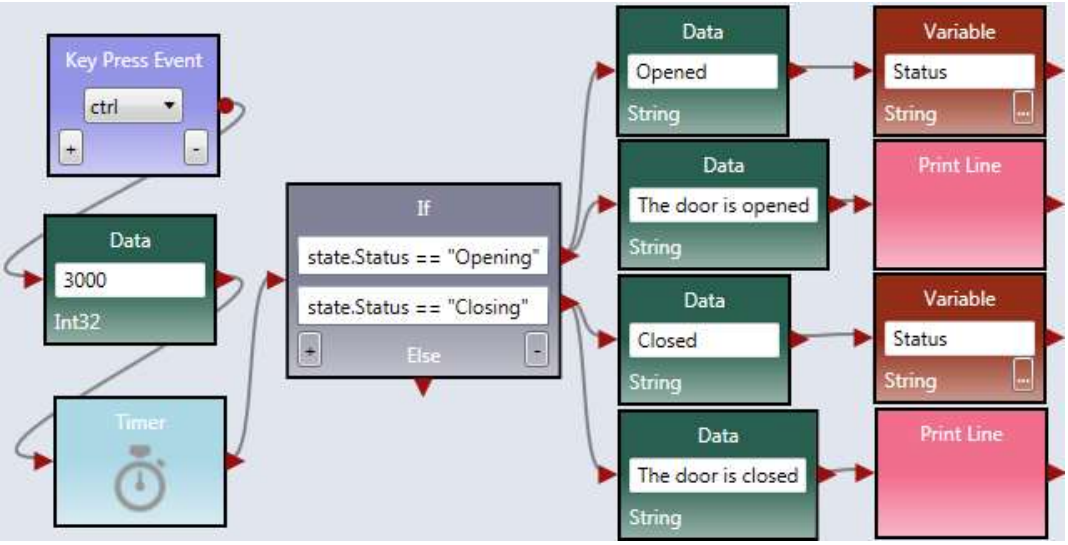


Figure 12. ASU VIPLE program (part 2) of the garage door opener

Test you program and make sure it works as expected. Make sure you press the ctrl key before the timer expires, which test if you can stop the door before the limit sensor is touched.

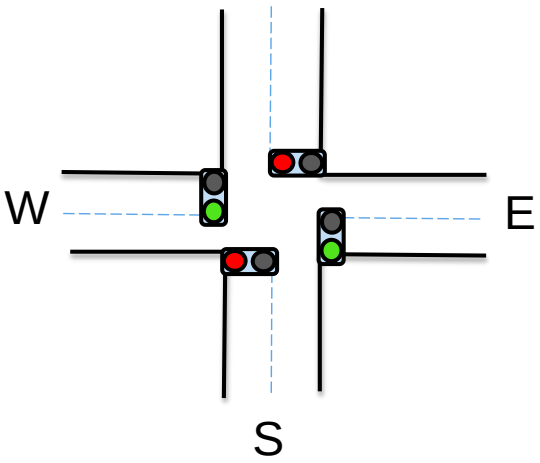
When you are done, please notify your lab instructor and demonstrate your program for sign-off. Then change the driver to proceed to the next assignment.

Exercise 5. Traffic Control with Emergency Inputs

Given a Finite State Machine (FSM) and sample code in Figure 13, implement the simulated traffic controller in ASU VIPLE. Improve the printed information, shat human can better understand the printed message. For example:

Now North-South is red and East-West is green

Now East-West is red and North-South is green



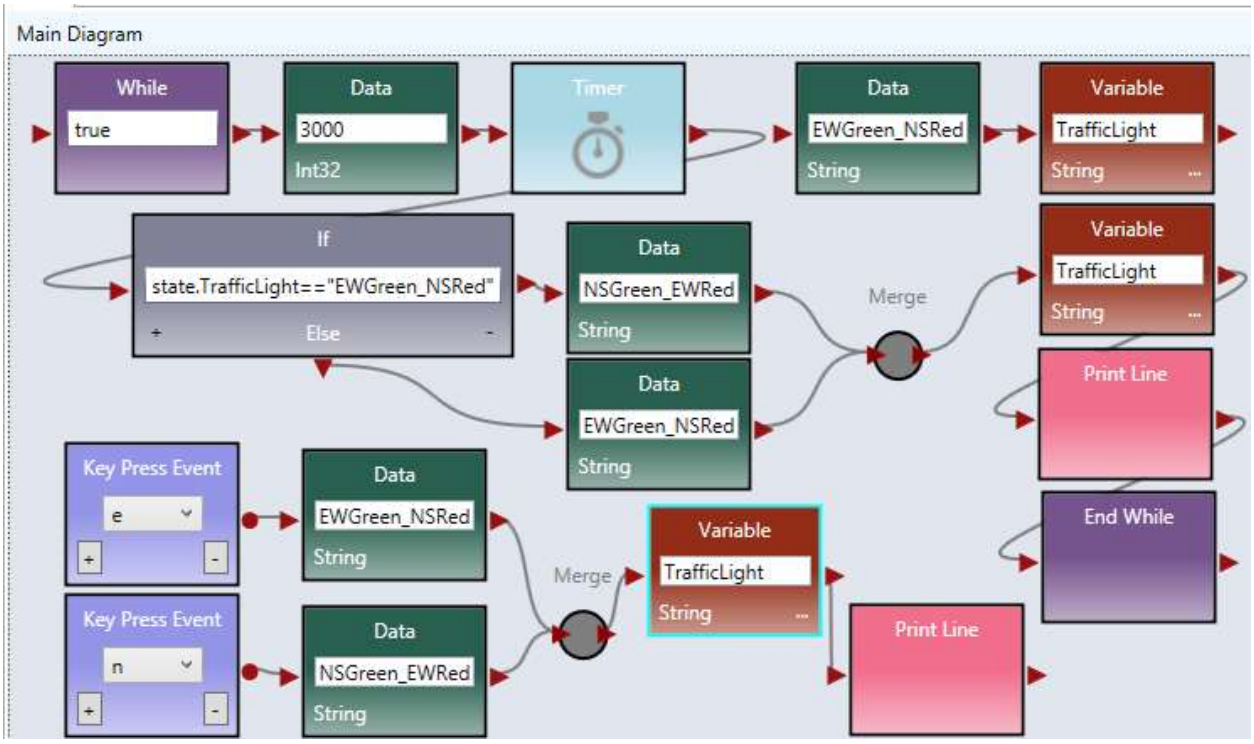


Figure 13. Finite state machine for Traffic Control with Emergency (keyboard) Inputs

Test you program and make sure it works as expected. When you are done, please notify your lab instructor and demonstrate your program for sign-off. Then change the driver to proceed to the next assignment.

Exercise 6. Drive-by-wire Simulation in Unity Simulator

We have been using VIPLE for general programming in flow control style and in event-driven style. These preparations are necessary for fulfilling our main purpose of programming IoT and robotics applications. In the rest of the section, we will focus on robotics application development using both simulated and physical robots. We start with the drive-by-wire program that controls the robot using the keyboard of the computer. Then, we will discuss the autonomous programs that control robots to navigate through the maze without any human intervention.

A number of robot services are implemented in VIPLE to facilitate different robots. Two simulated robots and environments are implemented: Unity Simulator and Web Simulator. We start with using the Unity Simulator, and the Web Simulator will be discussed later.

Step 1: First, drag and drop the service "Robot" in the diagram. Right click the robot to use the following configurations (1) In Set TCP Port: set port number to 1350, In Properties: choose localhost, and in Connection Type: choose Wi-Fi, as shown in Figure 14.

Figure 14. Configuration of the Robot service for simulation

Step 2: Now, we can write the drive-by-wire code as shown in Figure 15. You can follow the comments to sect the data connection values. You can find the services in the VIPLE service list.

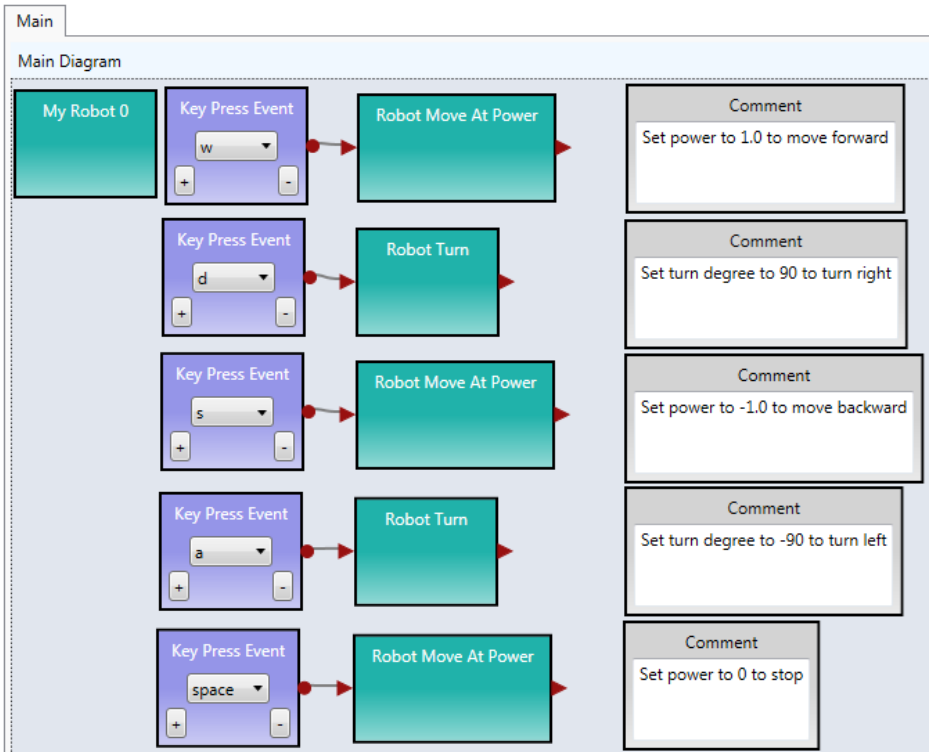


Figure 15. Basic drive-by-wire diagram

You must right click each Robot Move or Robot Turn activity and choose "My Robot 0" as the partner.

When you start the code, you will not see anything happening. You need a simulator or a real robot to see the robot controlled by the code. We will a simulator at this time and use a real robot later.

Step 3: Start the Unity simulator in VIPLE by choosing the VIPLE menu Run → Start Unity Simulator. Figure 16 shows the VIPLE start command and the simulated maze environment and the robot. Now, you can drive your robot using the five keys: w, d, a, s, and space. You can also change the maze by mouse-clicking the maze area to add and remove bricks.

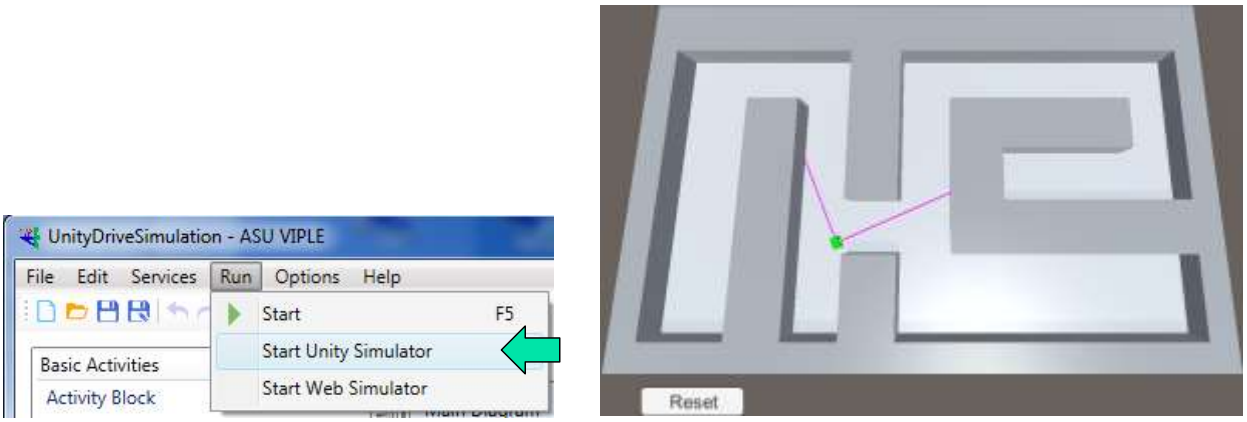


Figure 16. The simulation environment

Note, in order for the VIPLE code to take the keyboard input, you need to keep the "Run Window" on the top of your screen. In order to see the maze and keep the Run Window on top, you need to reduce the VIPLE code window, so that it does not block the maze.

When you are done, please notify your lab instructor and demonstrate your program for sign-off. Then change the driver to proceed to the next assignment.

Exercise 7: Autonomous Maze Navigation in Unity Simulator

Read the VIPLE diagram given in Figure 17 and answer the following questions.

1. What algorithm does this VIPLE diagram implement?
2. What states does this diagram use?

   Hint, there two types of variables: (1) variables with a fixed (finite) number of possible values and (2) variables with unlimited number of values, such as integer or double. The finite state machine uses those variable's values as states that have fixed (finite) numbers of possible values.

3. Draw the finite state machine of this diagram.

   Hint: You first decide the states based on the values of the variable isBusy. Then, you add transitions and the input and output related to each transition to form the finite state machine.

4. What values should be given to the Robot Move and Robot Turn services in order to complete the algorithm?

When you are done, please notify your lab instructor and demonstrate your program for sign-off. Then change the driver to proceed to the next assignment.

We start to use the Unity simulator.

Enter the Diagram into VIPLE. Make sure you configure the Robot, sensor, and motor services with the following values.

Right click the robot to use the following configurations: (1) In Set TCP Port: set port number to 1350; In Properties: choose localhost; and in Connection Type: choose Wi-Fi.

Right click each Move service and choose "My Robot 0" as the partner.

Right click each Distance Sensor service, choose "My Robot 0" as the partner. Set Port to 1 for right sensor and set to 2 for the front sensor.

Start the Simulator and run the diagram. Adjust the values given to the Robot Move and Robot Turn services to make the program work.

Follow Figure 21 to change the maze and test the effectiveness of the algorithm.

When you are done, please notify your lab instructor and demonstrate your program for sign-off. Then change the driver to proceed to the next assignment.

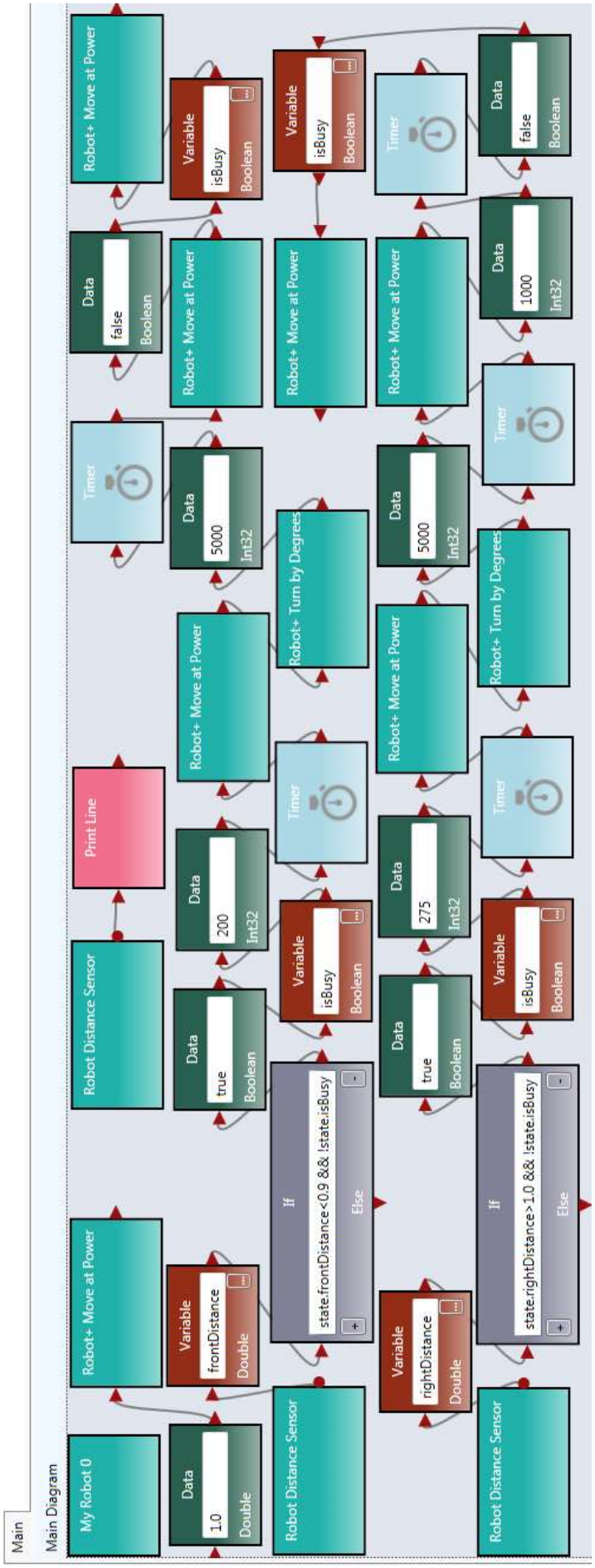


Figure 17. Main diagram of a maze navigation algorithm

In this previous exercise, we implemented a wall-following algorithm. In this exercise, you will implement the two-distance-local-best algorithm in Unity Simulator. Its finite state machine is shown in Figure 18.

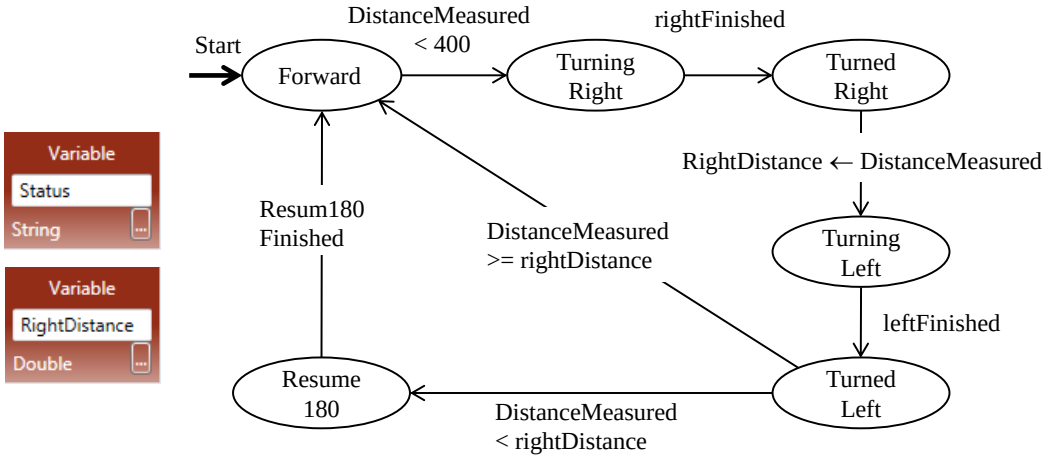


Figure 18. Two-distance-local-best algorithm

Exercise 8: Maze Navigation in Two-Distance-Local-Best Algorithm

You will use modularized code to implement the algorithm. First, you create the activities needed, as shown in Figure 19.

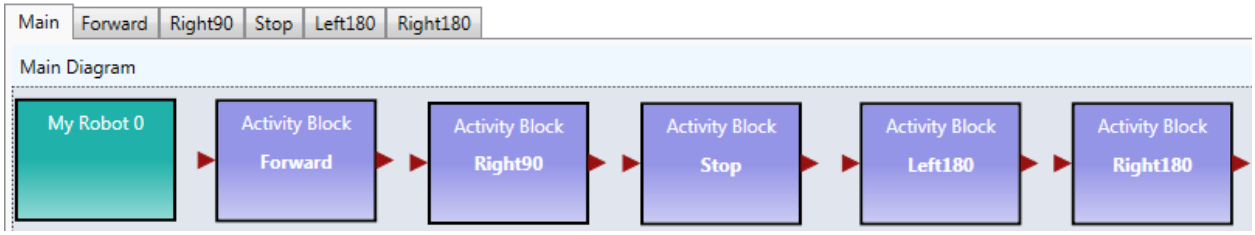


Figure 19. Activities needed in the Main diagram

The codes for the activities are given in Figure 20. The codes for turning Left180 and for Right180 are the same, except the turning degree is set to -180.0 and 180.0, respectively. The timer is used for delay the next operation for stability.

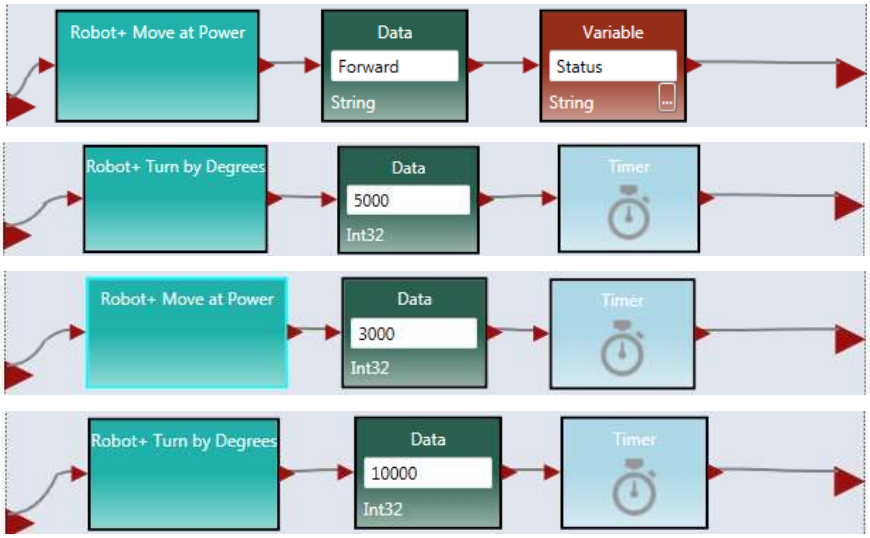


Figure 20. The code of the activities

Right click the Robot service to use the following configurations: (1) In Set TCP Port: set port number to 1350; In Properties: choose localhost; and in Connection Type: choose Wi-Fi.

Right click each Move and Sensor service and choose “My Robot 0” as the partner.

When you are done, please notify your lab instructor and demonstrate your program for sign-off. Then change the driver to proceed to the next assignment.

Following the finite state machine, the first part of the Main diagram is given in Figure 21.

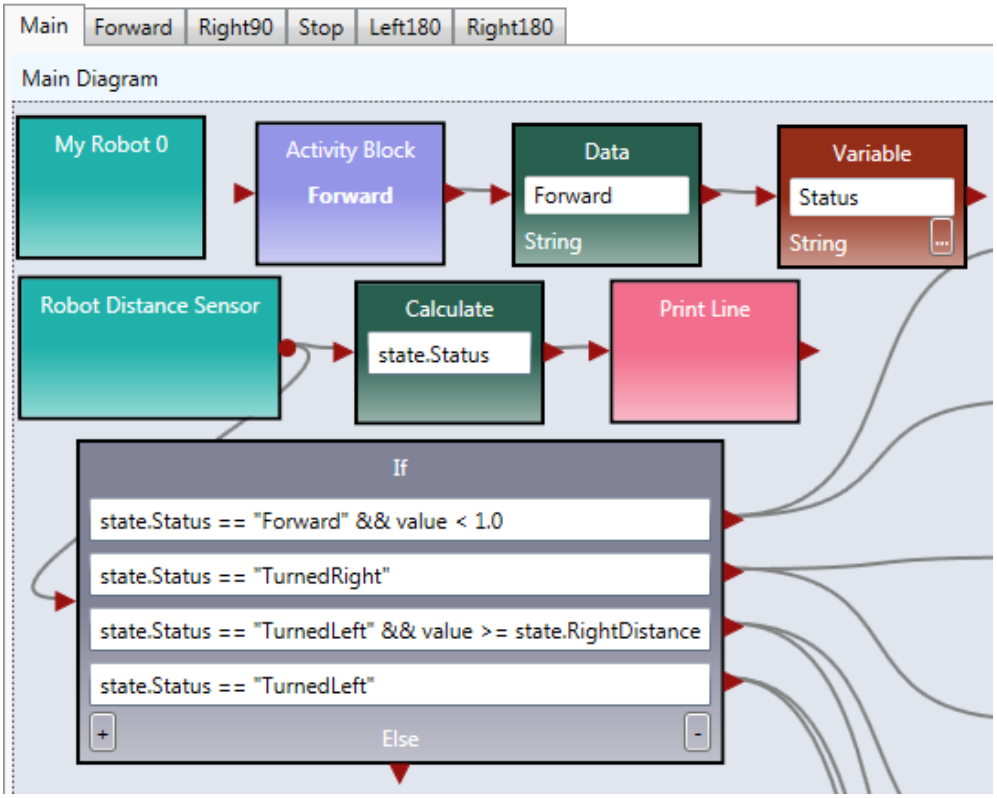


Figure 21. The first part of the Main diagram

The second part that is connected to the first part is given in Figure 22.

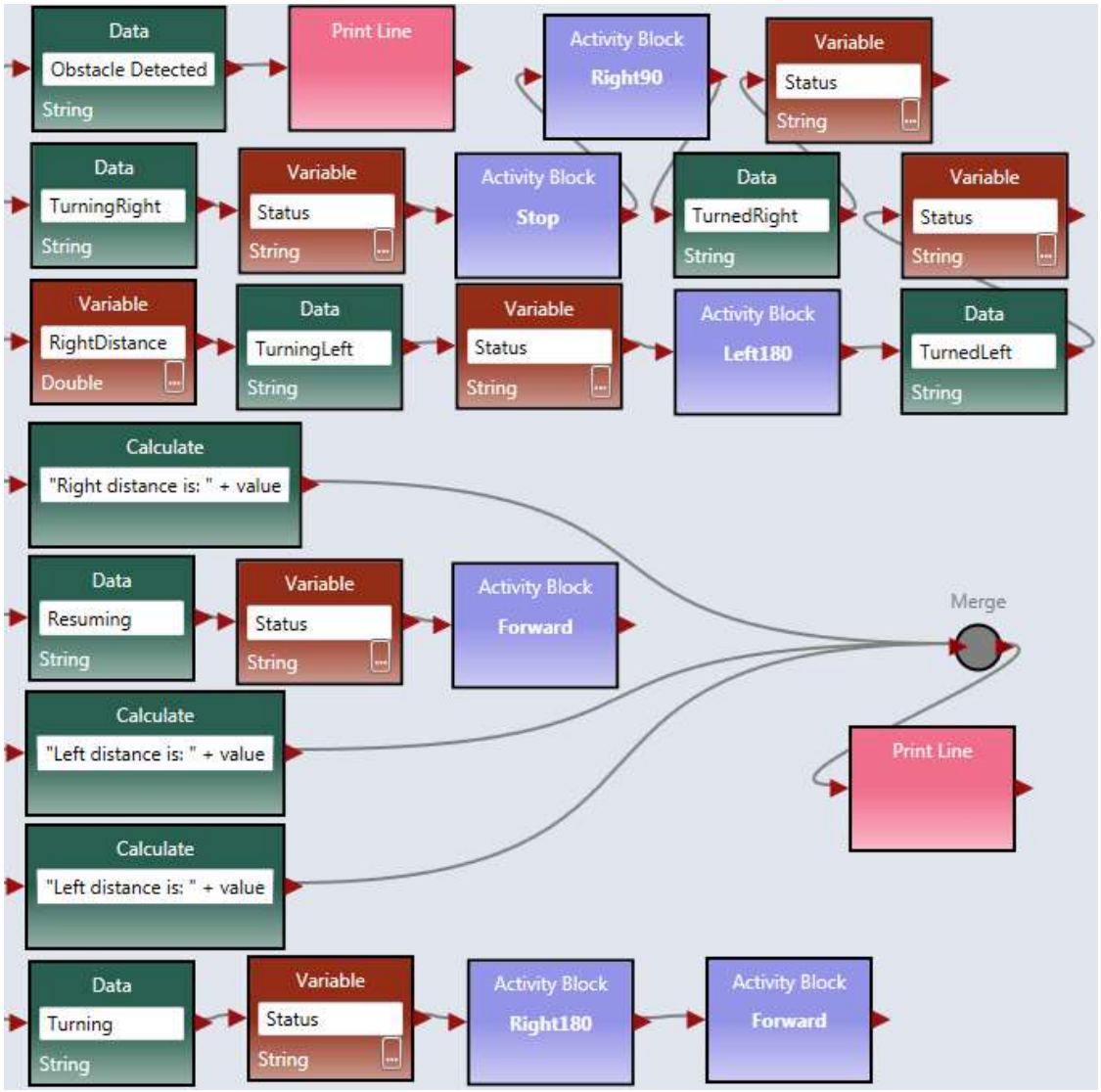


Figure 22. The second part of the Main diagram

Start the Simulator and run the diagram.

Follow Figure 6 to change the maze and test the effectiveness of the algorithm. Is this algorithm more efficient than the wall-following algorithm?

When you are done, please notify your lab instructor and demonstrate your program for sign-off. Then change the driver to proceed to the next assignment.

Lab 3
Physical Robot and Maze Navigation

Overview

In this lab, you will use finite state machines to define different maze navigation algorithms and evaluate the performance of different algorithms. You will use VIPLE's robot services and the simulation environment to solve the problems.

The pre-lab quiz will be based on the lab preparation part. You need to read all sections in the preparation part carefully before you can answer the questions in the pre-lab quiz. You will do the lab assignment part with your group during the lab.

Lab 3 Preparation

You need to read the preparation part carefully. The lab assignment part of this manual will be completed in the lab as a team assignment.

1. VIPLE Robot Services

VIPLE is a programming environment with general-purpose functions, as well as IoT/Robotic specific functions. Three sets of services are implemented: General computing services, generic robotic services, and vendor-specific robotic services.

- General computing services: include input/output services (Simple Dialog, Print Line, Text To Speech, and Random), Event services (Key Press Event, Key Release Event, Custom Event, and Timer), and component services (CodeActivity, RESTful service and WSDL service), as shown in the first part of Figure 1. CodeActivity will creased a text window that allow to use any C# code to form an activity. WSDL service is not showing in the list. It will be created from menu command. Most of the basic library functions supported in C# can be called in the data box of Calculate activity. Timer service takes an integer i as input, and it will hold data flow for i milliseconds. Timer service is frequently used in the robotic applications.
- Generic robotic services: VIPLE offers a set of standard communication interfaces, including Wi-Fi, TCP, Bluetooth, USB, localhost, and WebSocket interfaces. The data format between VIPLE and the IoT/Robotic devices is defined as a standard JSON (JavaScript Object Notation) object. Any robot that can be programmed to support one of the communication types and can process JSON object can communicate with VIPLE and be programmed in VIPLE. As shown in the second part of Figure 1, all VIPLE services that start with Robot are generic robotic services. We will use these services to program our simulated robots and custom-built physical robots.
- Vendor-specific services: Some robots, such as LEGO robots and iRobots, do not offer an open communication and programming interfaces. In this case, we can offer built-in services in VIPLE to access these robots without requiring any programming efforts on the device side. Currently, the services for accessing LEGO EV3 robots are implemented, so that VIPLE can read all EV3 sensors and control EV3 drive-motors and arm-motor, as shown in the third part in Figure 1. For those who do not want to build their own robots can simple use VIPLE and EV3 combination.

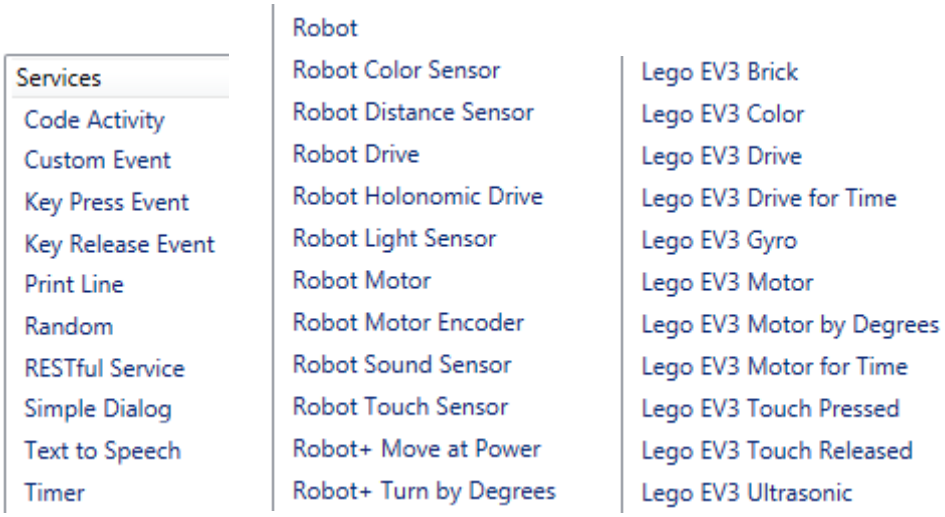


Figure 1. List of general services, generic robotic services, and EV3 services in VIPLE

In this lab assignment, we will use generic robotic services to program simulated robots and Intel-based physical robots. These services are explained as follows.

- **Robot** service is used for defining the connection types, connection port, and connection addresses. Multiple Robot services can be used in one application to control multiple robots. For each motor service and sensor service used, a partner Robot need to be selected.
- Robot Motor and Drive services. A number of services are defined for controlling different types of motors defined on the devices. Which services to use are determined by the physical device that are programmed to connect to VIPLE, and they should be specified in device hardware manual.
    - **Robot Motor**: It controls a single motor. It requires to set up a partner Robot, a motor port number, and a drive power value between 0 and ±1.0. The bigger the value, the faster the motor rotates. Positive and negative values are allowed, which will cause the motor to rotate in opposite directions.
    - **Robot Motor Encoder**: It is the same as Robot Motor, but it is of motor encoder type.
    - **Robot Drive**: Control two motors at the same time for driving purposes. It requires to set up a partner Robot, two motor port numbers, and two drive power values. The bigger the values, the faster the motors rotate. Positive and negative values are allowed. If two identified positive values are given, the robot moves forward. If two identified negative values are given, the robot moves backward. If one bigger value and one smaller value, or one positive value and one negative value are given, the robot turns left or right.
    - **Robot Holonomic Drive**: Control four motors at the same time for holonomic driving purposes, such as controlling a drone. It requires to set up a partner Robot, four motor port numbers, and three drive values for X component, Y component, and rotation.
    - **Robot+ Move at Power**: It requires to set up a partner Robot and a drive power value between 0 and ±1.0 to move both wheels of robot forward (positive value) or backward (negative value). The motor port numbers are not required to specify (hard coded in the device). This service can be used in the **simulated robot**.
    - **Robot+ Turn by Degree**: It requires to set up a partner Robot and a degree value between 0 and ±360. The motor port numbers are not required to specify (hard coded in the device). This service can be used in a robot with gyro sensor to measure the turning degree, or in a simulated where the turning degree can be easily controlled. We will use this service in the Web Simulator.

- **Robot sensors**: A number of sensor services are defined for reading data from the device, including color sensor, distance sensor, light sensor, sound sensor, and touch sensor. Each sensor requires to set up a partner Robot and a port number.

2. Maze Navigation Algorithms

To program a robot to navigate through a maze, we need to choose or design an algorithm first. Different algorithms exist. They have different complexity and performance. The complexity of a computational algorithm is typically measured by the number of steps the algorithm needs to perform in the worst case. However, the maze navigation algorithms involve robot's movements and turns. These operations involve mechanical operations, which are orders of magnitude slower than program execution steps. Thus, the main criteria for evaluating the maze navigation algorithms are the arc degree of robot's turning and the distance of robot's movement.

In the previous lab, we implemented two simple algorithms in the eRobotic simulation environment. Right-wall-following algorithm and farthest distance algorithm. In the given maze, it takes longer time for the right-wall-following algorithm to find the exit of maze.

The right-wall-following algorithm can be described in the following steps:

1. Robot moves forward;
2. If sensor.right distance > 100, delay and then turn 90 degree to the right;
3. Else sensor.forward distance < 50, turn 90 degree to the left;
4. Return to step 1.

The simulated farthest distance algorithm can be described in the following steps:

1. Robot move forward;
2. If sensor.forward > 50, continue to move forward;
3. If sensor.left distance < sensor.right distance, then turn 90 degree to the right;
4. If sensor.left distance > sensor.right distance, then turn 90 degree to the left;
5. Return to step 1.

We can also easily implement a one-distance-first-working-solution algorithm as follows:

1. Robot move forward
2. If sensor.forward > 50, continue to move forward;
3. If sensor.right distance > 100, then turn 90 degree to the right;
4. If sensor.right distance < 50, then turn 90 degree to the left;
5. Return to step 1.

These algorithms are so simple that we do not need to use any states or variables. The reasons for the simplicity are as follows:

- The eRobot in Figure 2 has three distance sensors in front, left and right. It can make the decision based on three distances. If there is one distance in the front only, the robot will have to rotate to measure the distances on the other sides. Then, it will need to have a variable to save the previous distance.
- The eRobot in Figure 2 can move forwards and turns with 100% accuracy. If the motors are not accurate in moving straight or turning, the algorithm must consider the adjustment and compensation to correct the errors, which will requires states and variables.

In the reality, the motors are not accurate and not consistent. They cannot move in straight line and cannot turn the desired degrees. The drop-down list language in the eRobotics environment does not support states and variables, and it is not capable of simulating the physical robots that we will use in our experiments.

The Unity simulator is a more powerful simulation environment that can simulate more conditions. It works with VIPLE, which is a much more powerful programming language supporting variables, data, calculation,

and various control structures. In this lab, we will use VIPLE and Unity simulator to implement various maze navigation algorithms that mimic more realistic situations.

3. Maze Navigation Algorithms Using Finite State Machine

Since our physical robot will have one distance sensor installed, we will define the algorithms in this section use one distance sensor only. The sensor is assumed to be installed in the front of the robot. We will use finite state machines to describe the maze navigation algorithms.

First-Working-Solution Algorithm

The first-working-solution algorithm is an algorithm that instructs a robot to move in the first direction that has a distance great than a given constant. Figure 3 shows the finite state machine of this algorithm. The finite state machine consists of four states. The robot starts with "Forward". If the front distance becomes less than a given value, the robot starts "Turning Left" 90 degrees. After "Turned Left", the robot compares the distance sensor value. If it is big enough, the robot enter "Forward" state. Otherwise, it spins 180 degree back to the other direction and then moves forward.

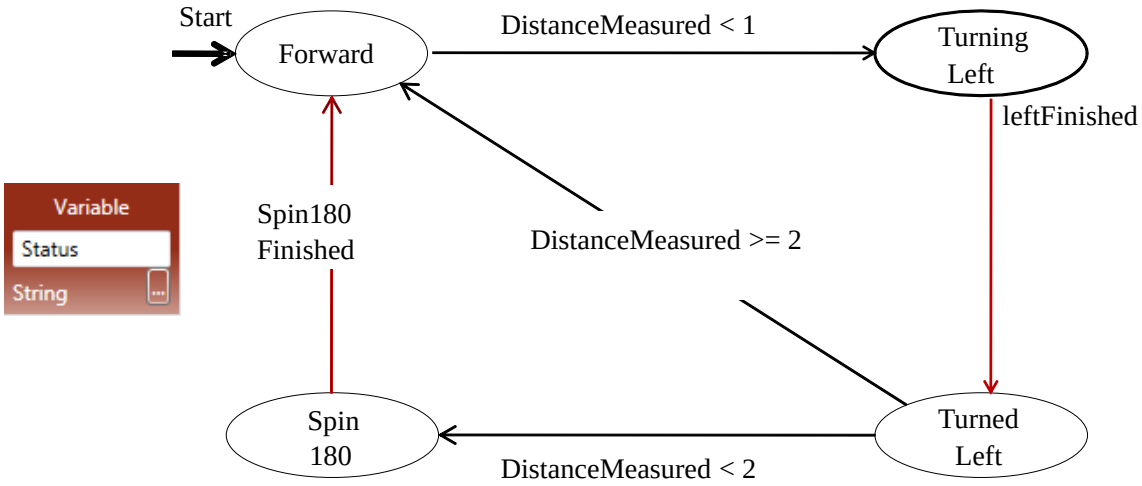


Figure 3. Finite state machine for the first-working-solution algorithm

Two-Distance-Local-Best Algorithm

The first-working-solution algorithm may not perform well in certain mazes. Figure 4 shows the two-distance-local-best (farthest distance) algorithm.

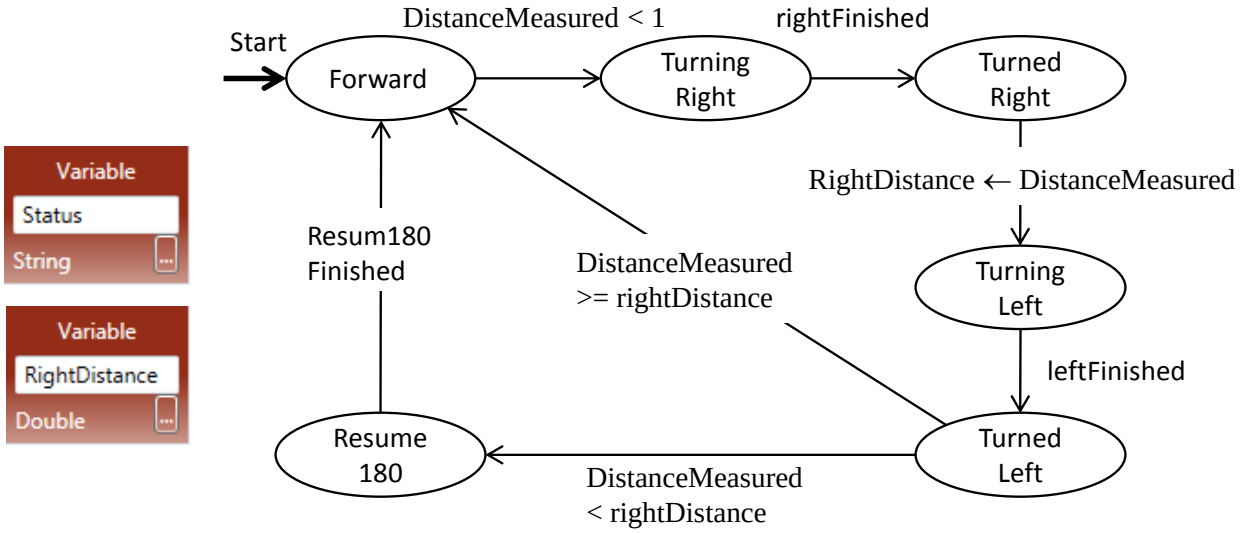


Figure 4. Finite state machine for the two-distance-local-best algorithm

Instead of comparing the left-side distance with a constant, it compares the left-side distance with right-side distance, and then moves to the side with farther distance. This finite state machine adds two states to include "Turning Right" and "Turned Right". It also uses a variable to hold the RightDistance. Recall that the robot is assumed to have one distance sensor only. It has to store the right-side distance before it measures the left-side distance.

Self-Adjusting Right-Wall-Following Algorithm

Figure 5 shows the finite state machine for the self-adjusting right-wall-following algorithm. It assumes that there are two distance sensors, one in the front and one on the right side. The front sensor could be replaced by a touch sensor. The finite state machine uses two variables: Status and BaseDistance to the right wall. BaseDistance is initialized to a desired value to keep the robot in the middle of the road.

The robot starts with moving forward. It keeps the base distance with the right wall. If the distance to the right wall is too big (base distance+5), it turns one degree to right to move closer to the wall. If the distance to the right wall is too small (base distance -5), it turns one degree to left to move away from the wall.

If the right distance suddenly become very big (base distance+400), it implies that the right side is open and the robot should turn right 90 degree to follow the right wall.

If the front distance becomes too small (<200), it implies that no way in the front and no way on the right, thus, the robot has to turn left 90 degree.

This algorithm is necessary for the physical robot, as the robot cannot move 100 straight and cannot turn precisely 90 degree.

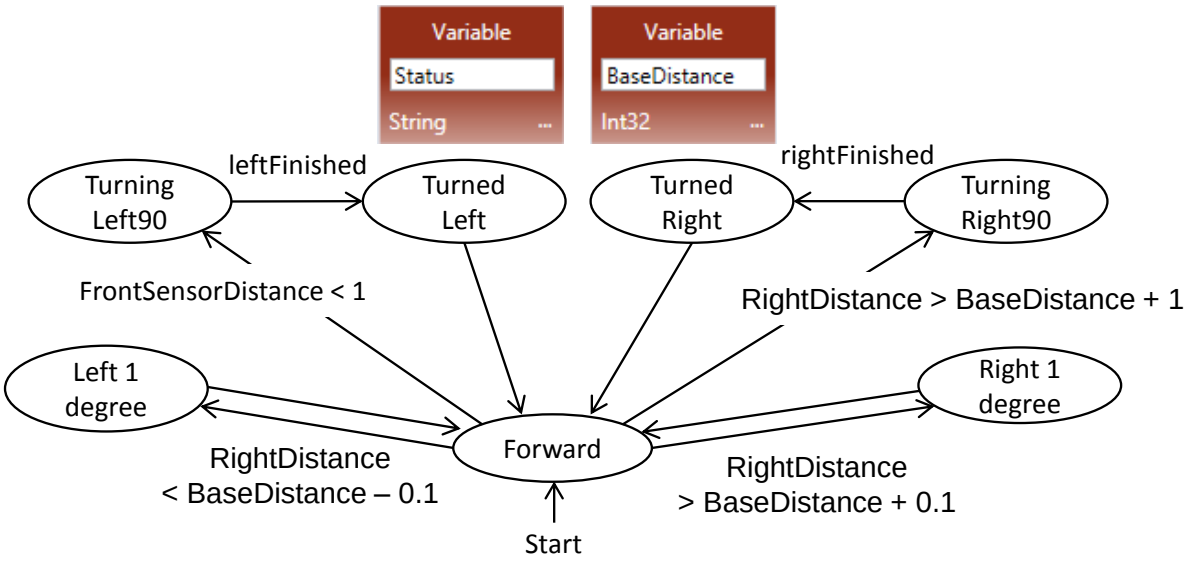


Figure 5. Finite state machine for the self-adjusted right-wall-following algorithm

The performance of maze these navigation algorithms depends on the maze too. If we apply these three algorithms in the mazes shown in Figure 6, which algorithm will perform the best?

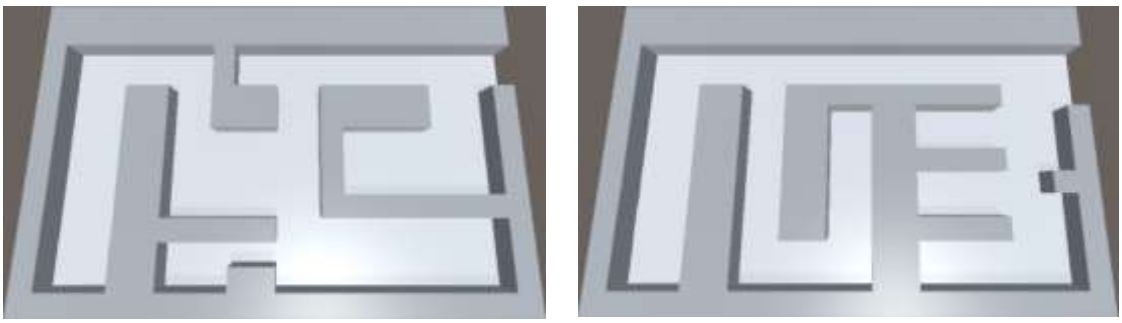


Figure 6. Different mazes can impact the performance of navigation algorithms

Lab 3 Assignment

In this lab, you will start with implementing autonomous maze navigation algorithms in two VIPLE platforms: (1) Unity Simulator and (2) physical robot.

The Unity simulator is simple and easy to get started. It uses the Robot+ Turn by Degrees service and two distance sensors to measure the distances in the front and at the right.

To implement Robot+ Turn by Degrees service in a physical robot, an accurate gyro sensor is required.

For the physical robots, we can use drive power and timing to control the turning degree of a robot. Robot Drive service is used the physical robot, which take two separate power for the left and right wheels.

Exercise 1: Testing Sensors and Motors

In the VIPLE program, we need to identify each sensor and each servo (motor) through device configuration, which include the main Robot, each sensor and each motor. Figure 7 shows the VIPLE code for testing the sensors connected to a robot.

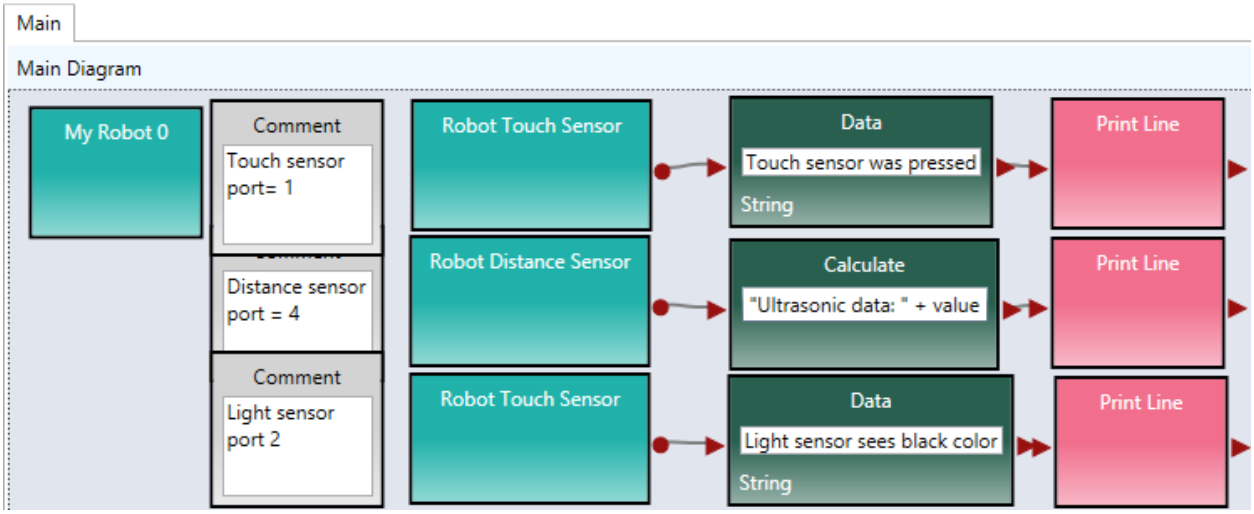


Figure 7. VIPLE code testing sensors

In order for the main robot, the sensors, and the motors to communicate with VIPLE properly, we need to configure the partnership between the main robot and its devices, the IP address (physical or virtual), and the ports. For Galileo robots that we use in the course, the configuration are as follows:

- TCP Port: 9999
- IP Address: 192.168.0.1 (password: bjut2016)
- Partner: My Robot 0
- Left Wheel: 6
- Right Wheel: 3
- Distance Sensor Port: 4
- Touch Sensor Port: 1
- Light Sensor Port: 2 (For line-following algorithm). As the light sensor outputs 1 when it sees black color and no output when sees white color. It works like a touch sensor. When programming the line-following program, we will choose touch sensor to connect to the light sensor.

Figure 8 shows an example of the configuration of the three devices: main robot, drive (motors) and the distance sensor. Notice that the numbers may differ for different robot configuration. The IP address is from the robot, which will be a fixed virtual IP address (as hotspot) or displayed on the LED panel on the robot, once the robot is started. If virtual address is used, the display panel is not necessary.

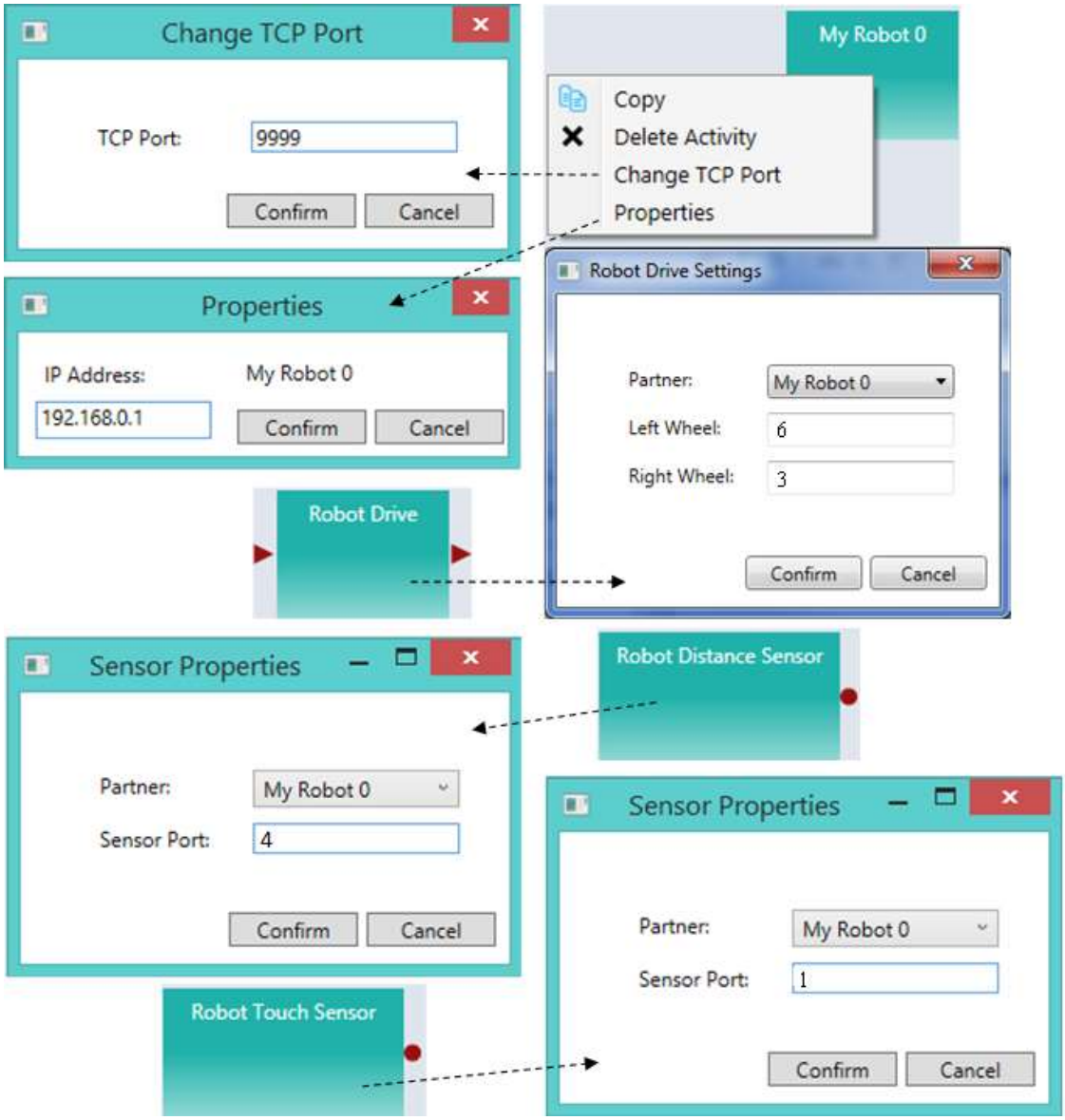


Figure 8. Configuration of IoT device ports

Now, we can write a simple program to control the robot by keyboard. Figure 9 shows the VIPLE code for testing the motors connected to a robot.

You can add "space" key to stop the robot by giving 0 power to both wheels.

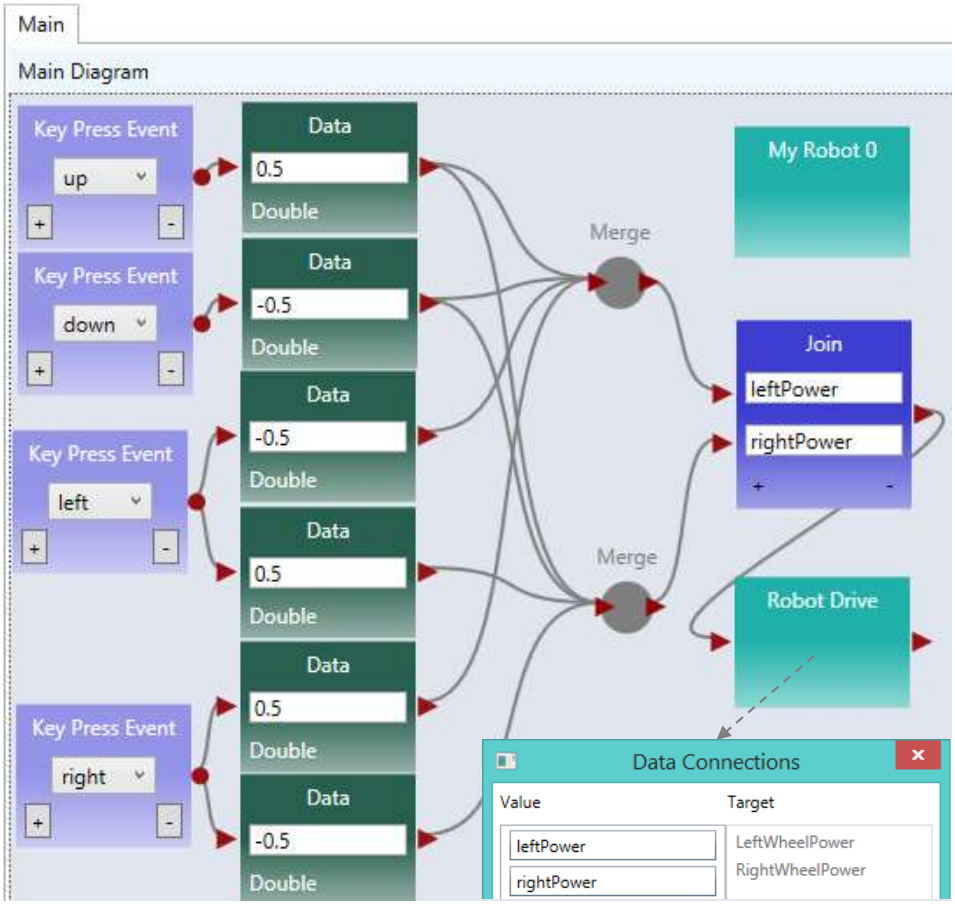


Figure 9. VIPLE code testing motors

Now, you will use your programs written for the Unity Simulator as the basis to implement different maze navigation algorithms.

Exercise 2: Implement Wall-Following Algorithm without the Self Adjustment

In this exercise, you will implement the following finite state machine that represents the wall-following algorithm without self-adjustment. This algorithm assumes that the robot can turn 90 degree precisely and can move forward in straight line. Figure 10 shows the finite state machine for this algorithm.

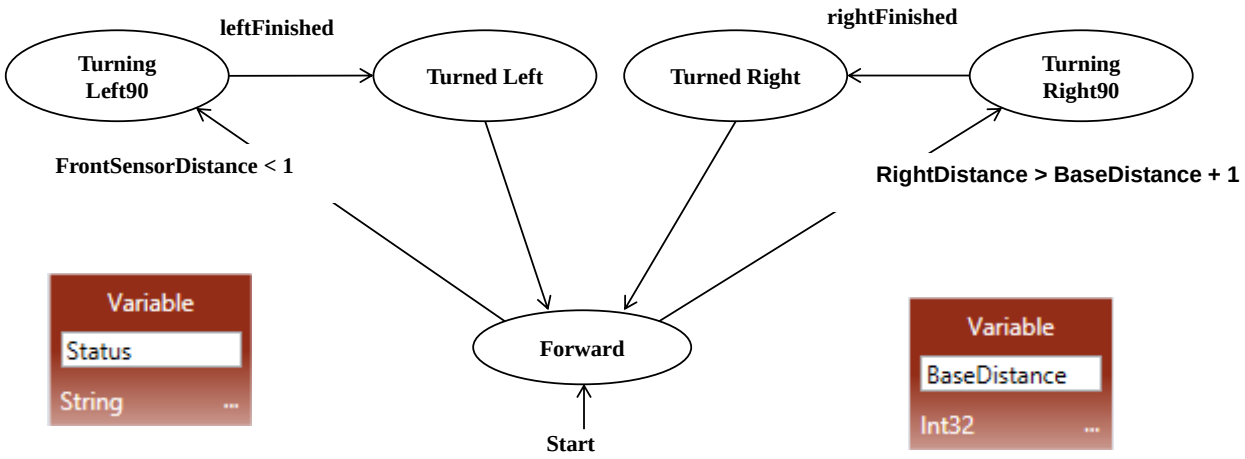


Figure 10. Finite state machine for the simple right-wall-following algorithm

We first implement the right-wall-following algorithm. The Main diagram is given in Figure 11.

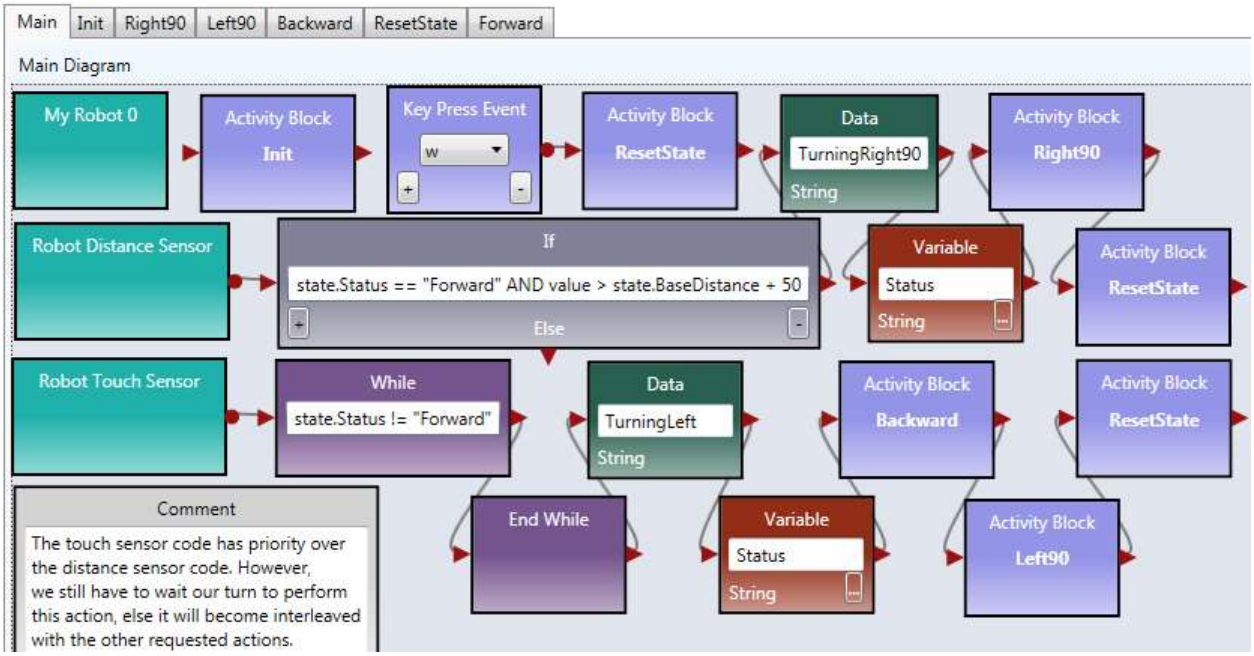


Figure 11. Main diagram implementing the right-wall-following algorithm

When you are done, please notify your lab instructor and demonstrate your program for sign-off. Then change the driver to proceed to the next assignment.

Now, we need to implement the Activity diagrams to be called in the Main diagram.

Step 1: Figure 12 shows the code for Init activity, which initializes the two variables and set the robot moves forward. There is no output from this activity.

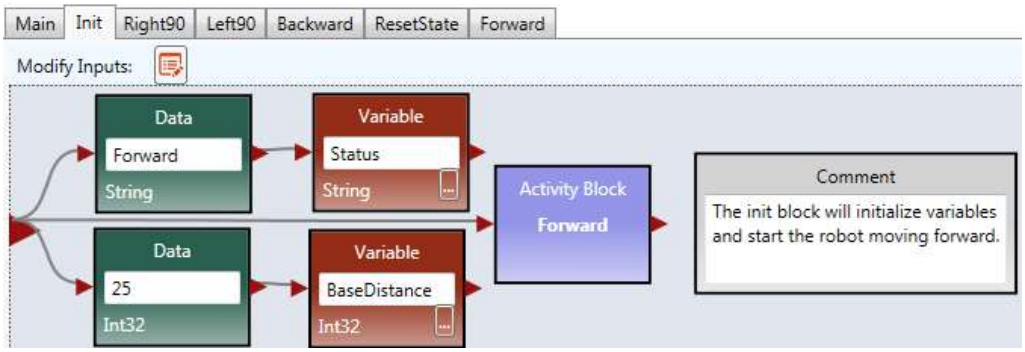


Figure 12. The Init Activity

Step 2: Figure 13 shows the implementation of the Right90 activity. Right click the motor and select the data connection. The data connection values for the two drive services are shown in lower part of the figure. The first set of values cause the robot to turn right and the second set of values cause the robot to stop.

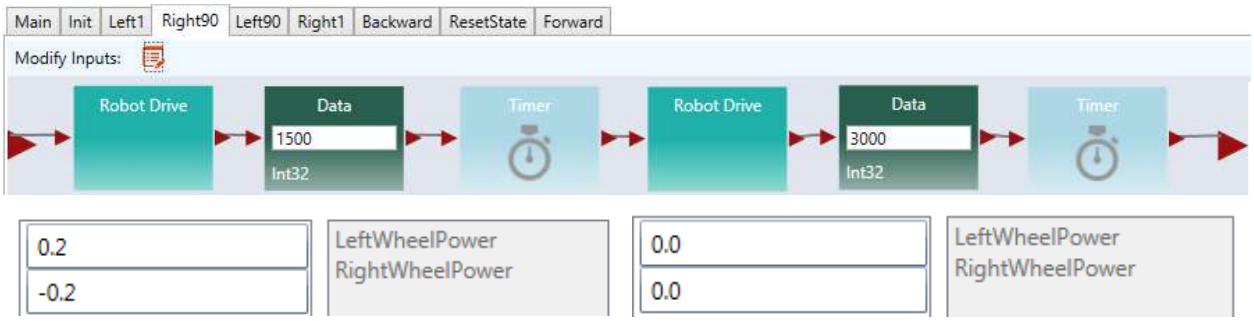


Figure 13. Right90 Activity and Data Connection

Step 3: You can follow the code for Right90 to implement Left90 by reversing the power on the wheels.

Step 4: Figure 14 shows the implementation of the Backward and Forward activities. For the Backward activity, the drive power can be set to -0.3 for both wheels. For the Forward activity, the drive power can be set to 0.5 for both wheels.

Figure 14. Backward and Forward Activities

Step 5: Figure 15 shows the code of for ResetState.

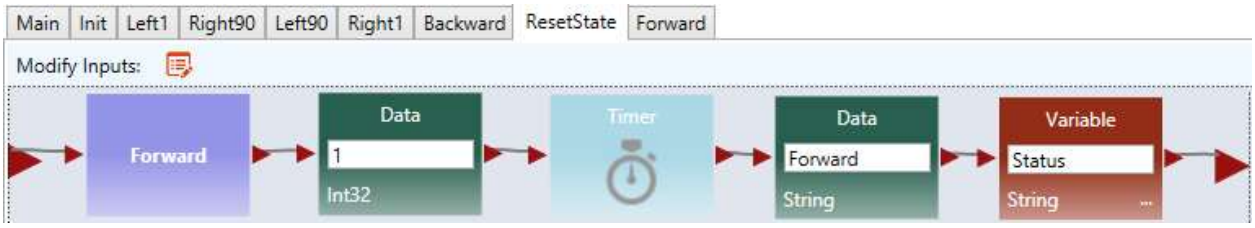


Figure 15. ResetState Activity

If you use a Key Press event to star the move, you need to click the VIPLE's Run window and use the key to star the move.

When you are done, please notify your lab instructor and demonstrate your program for sign-off. Then change the driver to proceed to the next assignment.

Exercise 3: Implement Wall-Following Algorithm with the Adjustment

In this exercise, you will implement the finite state machine in Figure 16 that represents the wall-following algorithm with self-adjustment. This algorithm allows the robot that does not turn 90 degree precisely and cannot move forward in straight line. It measures the distance to the right wall. If the distance is greater base distance + 0.1, it turns a small degree to the right. If the distance is less base distance - 0.1, it turns a small degree to the left.

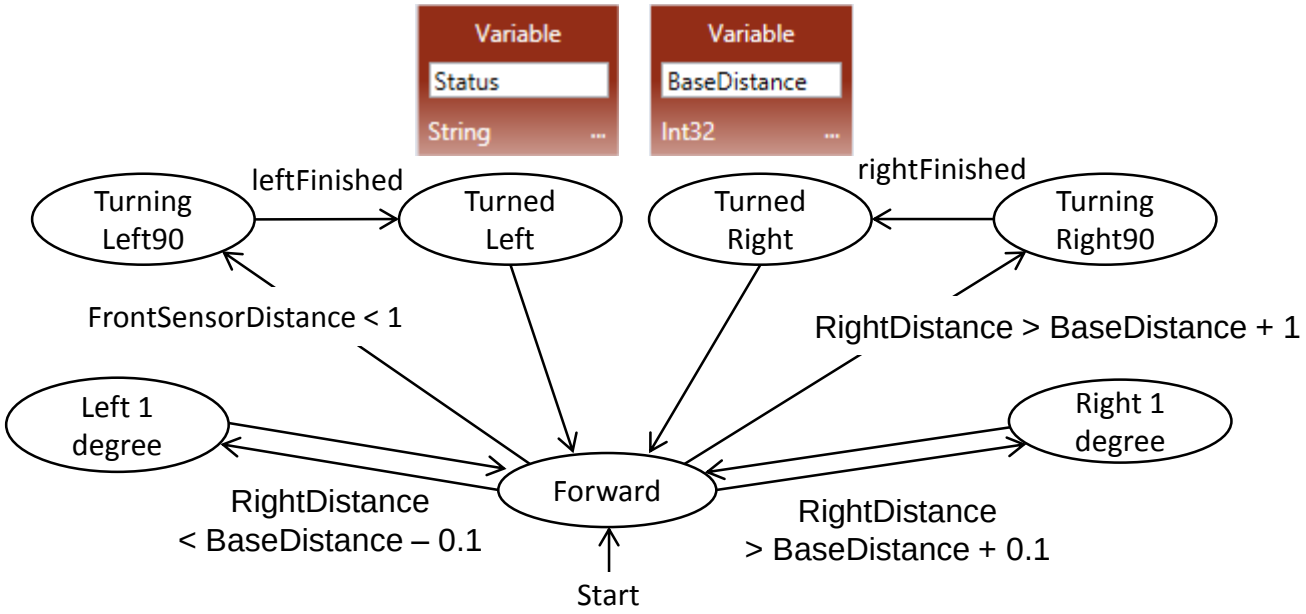


Figure 16. Finite state machine for the self-adjusted right-wall-following algorithm

Exercise 4: Implement Two-Distance-Local-Best Algorithm: Main

The Two-Distance-Local-Best Algorithm uses one distance sensor in the front of the robot, and thus, it need to turn the body of the robot to measure the distance on the right and then on the left. The finite state machine is given in Figure 17.

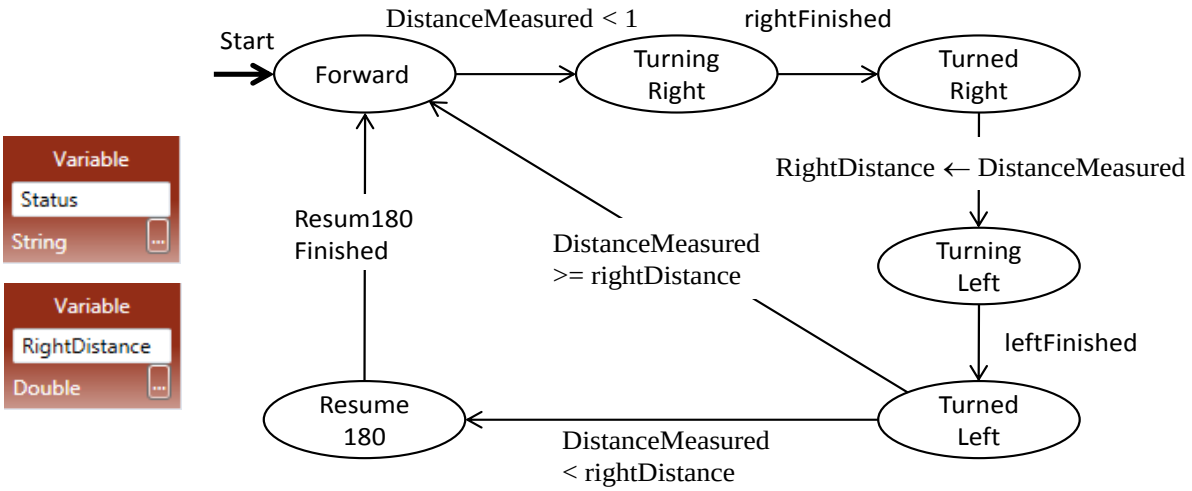


Figure 17. Finite State Machine for Two-Distance-Local-Best Algorithm

The main diagram will be similar to the code for the Unity Simulator. The differences are in the services used and the parameter values. Figure 18 shows the first of the Main diagram.

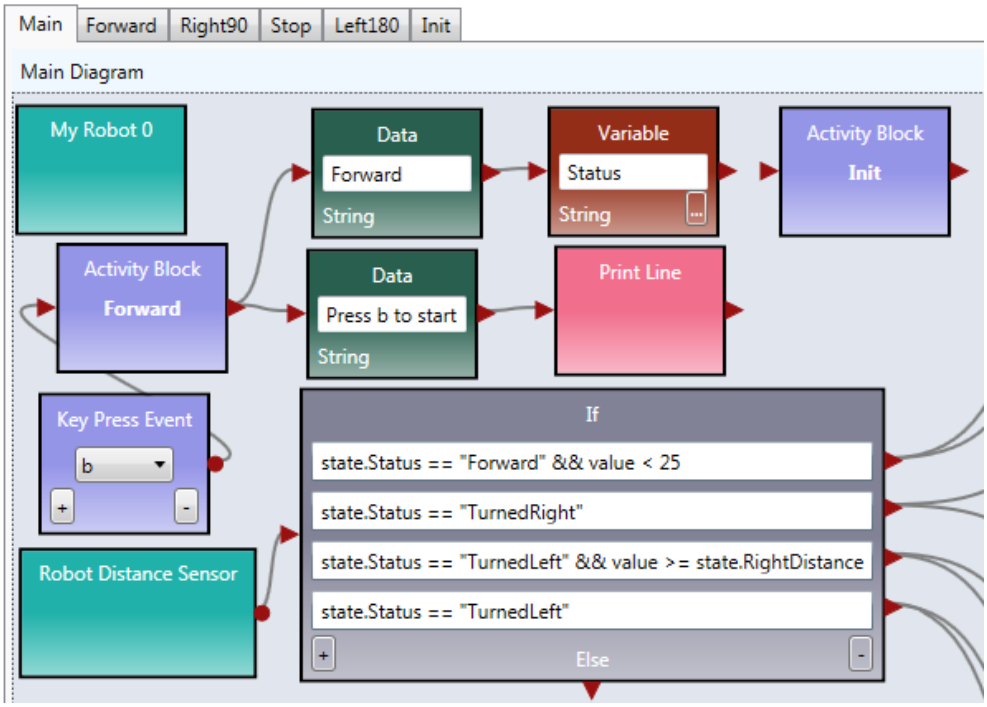


Figure 22. The first part of the Main diagram

Next, you will implement the following finite state machine. You can write the code of the activities in the same way as the code for the wall-following program. For the Init activity, you do not have the baseDistance variable to initialize. For the Left180 activity, you can start from Left90, and use longer time to make 180 degree.

Follow the same process to test the code, as you did in testing the wall-following algorithm.

When you are done, please notify your lab instructor and demonstrate your program for sign-off. Then change the driver to proceed to the next assignment.