Workbook for Designing Distributed Control Applications using Rockwell Automation's HOLOBLOC Prototyping Software

John Fischer and Thomas O. Boucher

Working Paper No. 05-017

Introduction

A new paradigm for creating distributed control applications is currently being proposed under the IEC 61499 standard. This standard emphasizes formal methods based on Unified Modeling Language and object oriented concepts. It makes a separation between events, data, and algorithms. Algorithm execution is initiated by the arrival of events and the algorithm uses the current values of data elements available when the event occurs. The programming is done using function blocks, a modeling formalism originally proposed under the standard IEC 1131-3, but extended under the current standard. Event propagation is accomplished by connecting events among the various function blocks.

This standard is currently evolving and has not yet been used in actual factory applications. The predominant standard today, IEC 1131-3, does not distinguish between events and data. The arrival of new data (sensory input) is itself considered an event. The continuous scan of the PLC program executes the ladder logic rungs (algorithm) associated with the data when it arrives. In contrast, IEC 61499 is event driven and does not necessarily scan each function block continuously.

IEC 61499 was developed to support distributed control. Distributed control is distinguished from purely hierarchical control by the fact that the decision processes associated with an application are not running under a single processor. Rather, the decision processes are divided among several processors, each having their own thread of control. However, in order to execute the application, these processors must exchange data and state information with each other.

Figure 1 is illustrative of a distributed control development platform for IEC 61499. Components on the network are intelligent devices with their own microprocessor and network drivers. Components communicate peer-to-peer and deliver timely information for the functioning of the overall system. Control software is configurable using a standard library of function blocks (FB) that can be assembled in a plug-and-play fashion to develop the control programs that will run on the individual devices. Standard function blocks for communication among devices enable peer-to-peer communication.

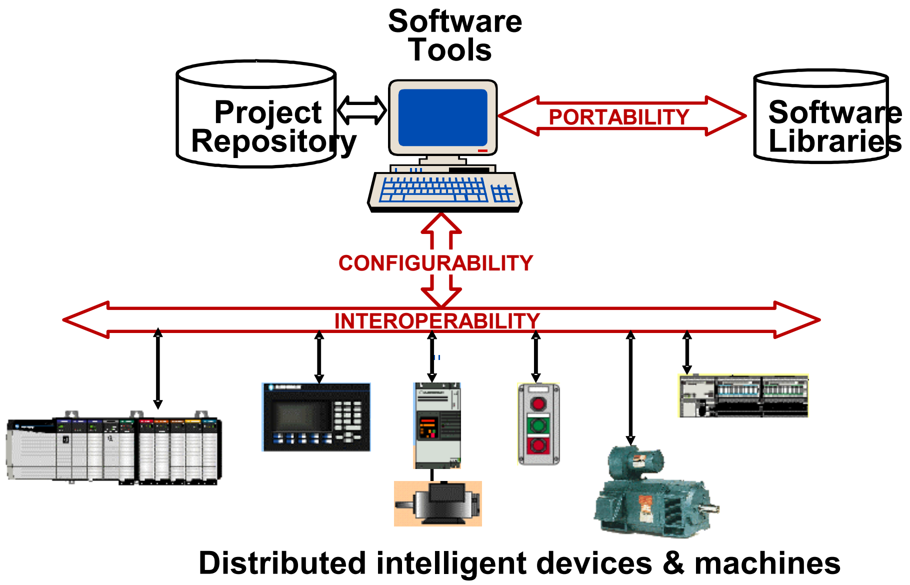


Figure 1. Components in a distributed control process

The distributed control architecture under IEC 61499 defines key elements of a distributed control system. They are application, device, and resource. An application is a related set of functions that must talk to each other in order to fulfill a control task. A device is a control unit having one or more processors. The device interfaces to the physical I/O and also communicates with other devices on the network. A resource is essentially a processor on which part of a distributed application will run. We are use to thinking about single unit operations consisting of a single PLC running all the control functions of a machine. Such a PLC usually has a single processor. The PLC, with its I/O and network interface, is the device. The processor of the PLC is the resource. The program on the processor is the application. Therefore, such a system is a device with a single resource and single application.

Figure 2 shows an example of distributed control on a single device. Here there are three resources that share the process I/O interface and communication interface of the device. The process I/O interface is for physical wiring of sensors and actuators. The communication interface is the network interface of the device over which the processors can communicate with other processors on other devices. Processors on the same device can communicate with one another over the device back plane. Application C is distributed across resources x, and y. Some of the execution code is handed by each resource and the resources must talk to one another for the application to run properly. Applications A and B extend beyond the device. They will have functions on other devices (not shown) and will communicate with them through the communication interface.

In the PLC world, a typical device usually has only one resource. For the time being we will assume that model. Therefore, distributed control consists in applications that are running on more than one PLC (device with a single resource) and communicating with each other peer-to-peer on a network. The problem of distributing an application is determining how the model of an application will be divided among the PLCs and how the PLCs will talk to one another.

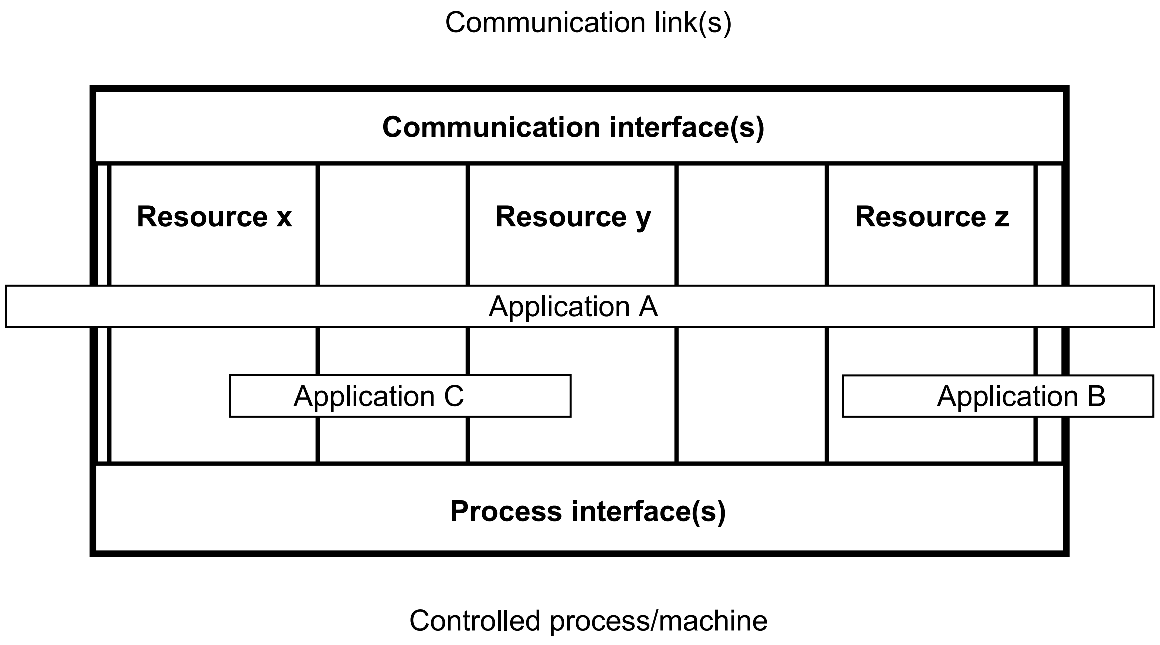


Figure 2. A device with three resources and three applications

The function block is the unit of programming. There are three types of function blocks: basic function blocks, composite function blocks, and service interface function blocks. A *basic function block* executes an elemental control function, such as reading a sensor or setting the state of an actuator. Basic function blocks may be combined together to encapsulate a higher-level control function. Such a combination is called a *composite function block.* The *service interface function block* provides the communication services among devices.

Figure 3 shows a model of the components of a basic function block. The model distinguishes between events, data, and algorithms. The upper quadrant of Figure 3 shows the event stream, which initiates the execution of the function block code. The arrival of an event executes one or more function block algorithms. Each event input will trigger a different execution scenario (combination of algorithms). After the execution of the scenario, the function block will typically enable an output event. Input events come from other function blocks and output events are sent to other function blocks in the application.

The data flow is shown in the lower quadrant of the FB in figure 3. Data inputs are provided either from physical devices or from other function blocks. At the time that an input event occurs, the data input values are read and the appropriate algorithm is executed using the current data values. Typically this results in a calculation or validation that yields data outputs of a function block. When the output event is triggered, the data output is made available to another function block that continues the execution of the application.

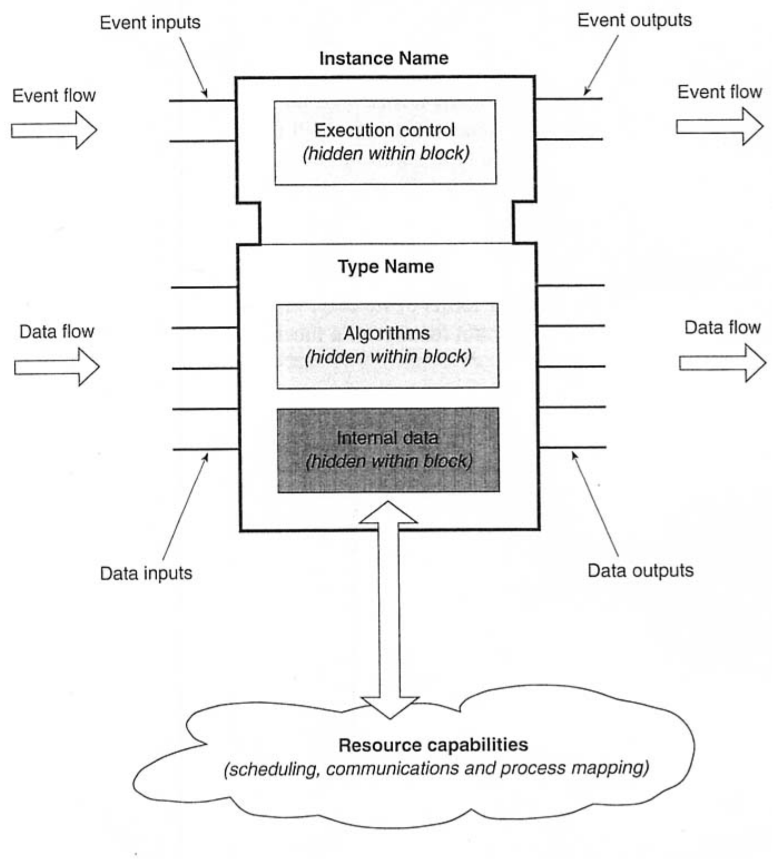


Figure 3 Components of a Function Block

A common example used to illustrate a function block application is PID control as shown in figure 4. The IO_Reader is a basic function block that scans the value of an analog sensor each time the input event RSP is triggered. The data from the sensor is read at the data input address SD_1. The algorithm of the IO_Reader simply takes the value at SD_1 and transfers it to RD_1. When the transfer is complete, the output event IND is triggered, providing the input event RUN to the PID algorithm function block. When the PID function block reads the process variable (PV) data input, it executes a standard PID control algorithm and outputs the value of the control variable (CV). IO_Writer is a basic function block for writing a value to an actuator. When the PID FB executes OK, the event REQ begins the IO_Writer algorithm, which reads the value at SD_2 and the address of the actuator (SD_1), then writes the value of the control variable to RD_1, the actuator physically connected to the process.

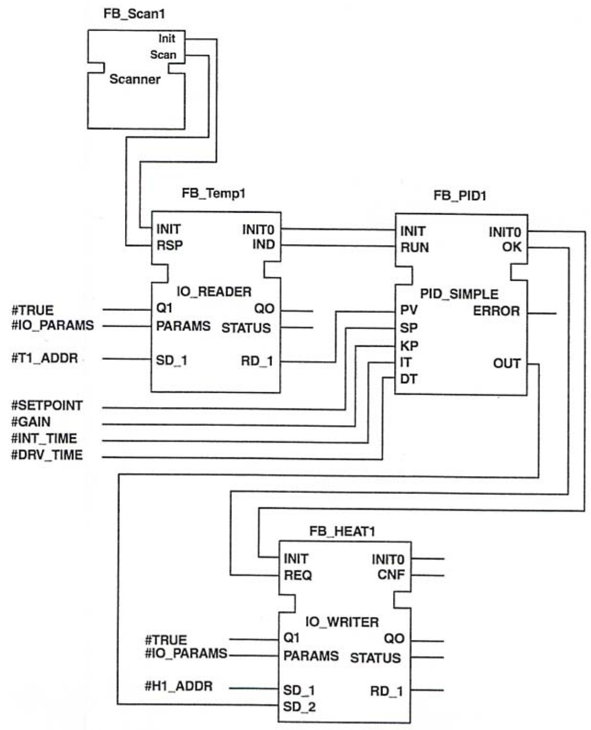


Figure 4 PID Controller Programmed with Function Blocks

The above example shows the elements of the function block and an application. The execution of the algorithm for a FB is described by a state machine, called an execution control chart (ECC). For example, figure 5 shows the ECC for the PID function block. The initial state is labeled "Start" and drawn with a double-sided rectangle. The condition on the arc (RUN) is the input event that sends the FB to the state RUN_PID. The PID algorithm is executed in that state. The PID algorithm uses the current values of the data inputs and sets the current value of the control variable output. The action box attached to the right of the algorithm action block is the name of the output event that is triggered upon completion of the algorithm. The label "1" on the output arc means "true always", directing a transition back to the start state when the algorithm execution is complete.

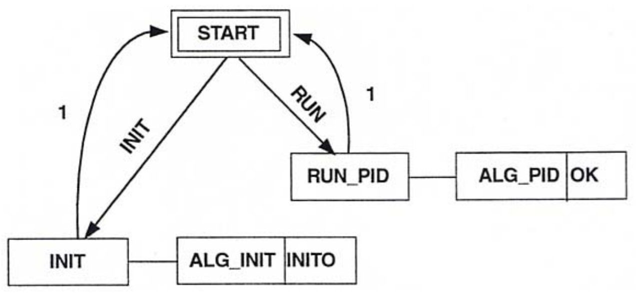


Figure 5 Execution control chart of PID_SIMPLE function block

This is a basic introduction to the IEC 61499 standard. For a more complete description, the reader is referred to *Modeling Control Systems Using IEC 61499* by Robert Lewis (The Institution of Electrical Engineers, London: 2001).

The purpose of this workbook is to provide hands-on tutorials on the use of IEC 61499 in designing function blocks and distributed applications. The creation of function blocks and the use of standard function blocks have been facilitated by Rockwell Automation through the Holobloc IEC 61499 prototyping software. The use of this software in designing function blocks will be described in the lessons of this workbook.

The Function Block Developmental Kit can be found online at Holobloc's website, http://www.holobloc.com. Under the link, Function Block Developmental Kit (FBDK), the software can be downloaded for free, non-commercial use. After clicking on the link, you will be at a web page titled "Getting Started." The instructions on downloading and setting up the FBDK are listed on this web page. Before downloading the FBDK, please read the Terms and Conditions of Use section of the page before downloading the software. Once you click on the statement, "I have read and agree to the following Terms and Conditions of Use," you will download a zip file. This zip file contains the FBDK executable file and other files for use with the FDBK (i.e. function blocks). Return to the "Getting Started" web page and follow the remaining steps for extracting the files from the zip file and setting those files up on your computer. If you do not have Java Developer Kit 5.0, or an older Java Developer Kit version; it is highly recommended that you download and install the latest version. Without it, it will not be possible to create your own function blocks. Once those steps are completed, you are ready to proceed through the following lessons to learn and use the FBDK.

TABLE OF CONTENTS

IEC TUTORIAL 1: INTRODUCTION TO IEC 61499

1.1 Introduction

The IEC 61499 is an evolving standard that uses function blocks in the distribution of control in industrial processes. This tutorial will go through the basics of using function blocks as well as explaining the use of the software for developing and testing control programs.

SITUATION: There is a programmable logic controller that has switches as inputs and lights as outputs. In this situation, there are two switches (switch 3 and switch 2) and two lights (light 1 and light 2). The system must meet the following requirements.

1) Both lights will be on if I:1/3 (switch 3) is turned on and I:1/2 (switch 2) is turned off.
2) If I:1/3 and I:1/2 are both turned on, then O:2/1 (light 1) will be on and O:2/2 (light 2) will be off.
3) If I:1/3 is turned off, then both lights will be off, regardless of whether I:1/2 is on or off.

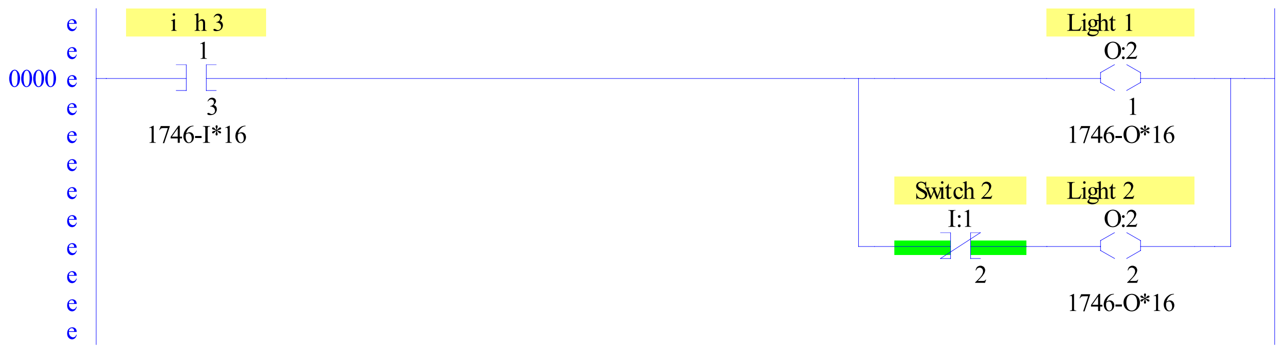


Figure 1.1 Ladder Logic for Tutorial 1

In Boolean logic, the PLC follows these equations.

$$\text{Light 1} = \text{Switch 3}$$

$$\text{Light 2} = \text{Switch 3} \bullet \overline{\text{Switch 2}}$$

1.2 Steps in Using the Holobloc FBDK Software

Now that the logic of the system is realized, programming it in IEC 61499 can now begin. First we must launch the function block editor. Using Windows Explorer, go to the directory that contains the function block editor: C:\fbdk. The right side of the screen should look like Fig. 1.2 in which the editor is highlighted. Double click on the editor.

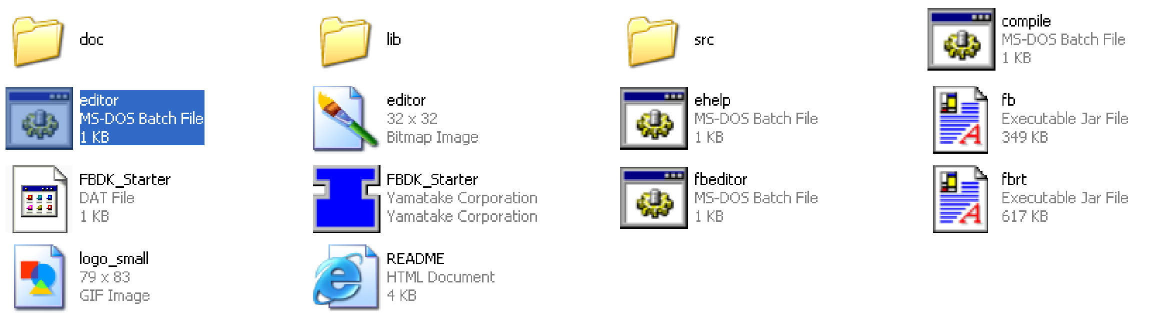
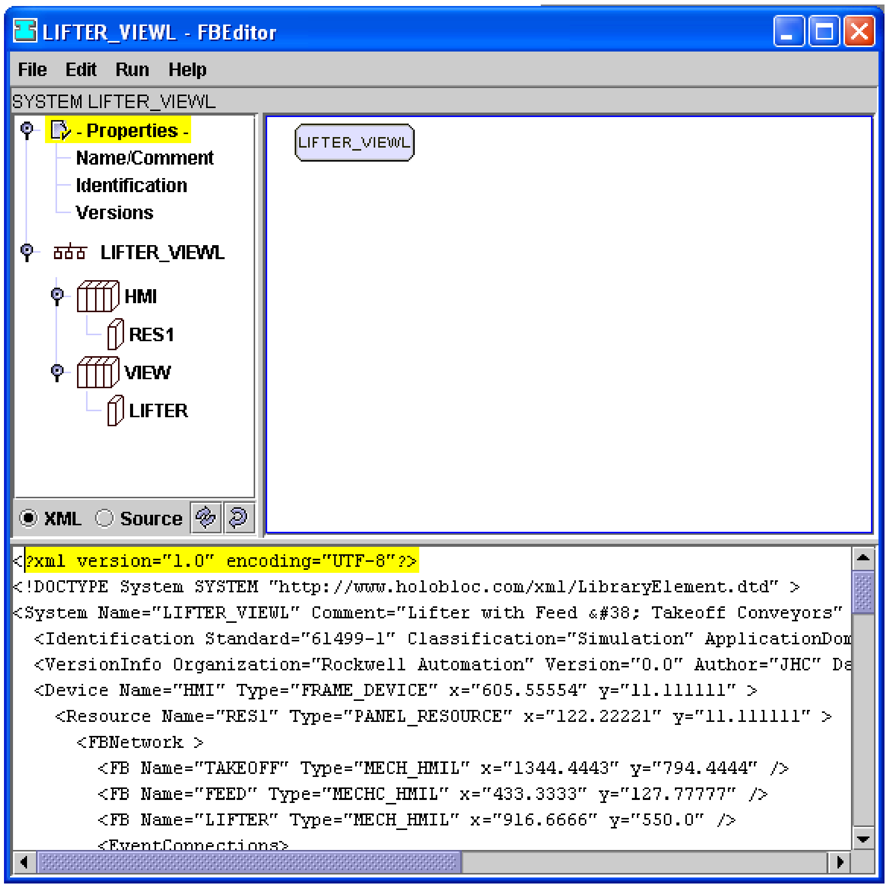

doc
lib
src
compile MS-DOS Batch File 1 KB
editor MS-DOS Batch File 1 KB
editor 32 x 32 Bitmap Image
ehelp MS-DOS Batch File 1 KB
fb Executable Jar File 349 KB
FBDK_Starter DAT File 1 KB
FBDK_Starter Yamatake Corporation Yamatake Corporation
fbeditor MS-DOS Batch File 1 KB
fbrt Executable Jar File 617 KB
logo_small 79 x 83 GIF Image
README HTML Document 4 KB

Figure 1.2 Function Block Editor

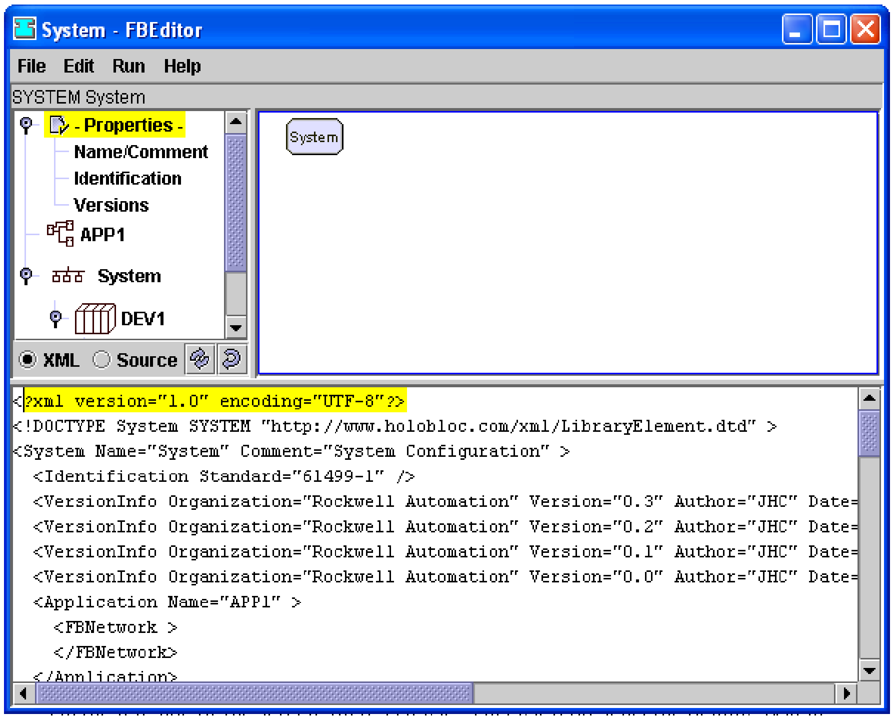
Step 1: Creating a New System

Once the editor is launched, you will see a screen that looks like Fig. 1.3a. The first step is to begin a new system configuration. Go the menu bar and click on File -> New -> System. The screen should be similar to Fig. 1.3b.



```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE System SYSTEM "http://www.holobloc.com/xml/LibraryElement.dtd" >
<System Name="LIFTER_VIEWL" Comment="Lifter with Feed &#38; Takeoff Conveyors"
  <Identification Standard="61499-1" Classification="Simulation" ApplicationDom
  <VersionInfo Organization="Rockwell Automation" Version="0.0" Author="JHC" Da
  <Device Name="HMI" Type="FRAME_DEVICE" x="605.55554" y="11.111111" >
    <Resource Name="RES1" Type="PANEL_RESOURCE" x="122.22221" y="11.111111" >
      <FBNetwork >
        <FB Name="TAKEOFF" Type="MECH_HMIL" x="1344.4443" y="794.4444" />
        <FB Name="FEED" Type="MECHC_HMIL" x="433.3333" y="127.77777" />
        <FB Name="LIFTER" Type="MECH_HMIL" x="916.6666" y="550.0" />
        <EventConnections>
```

Figure 1.3a Default Screen

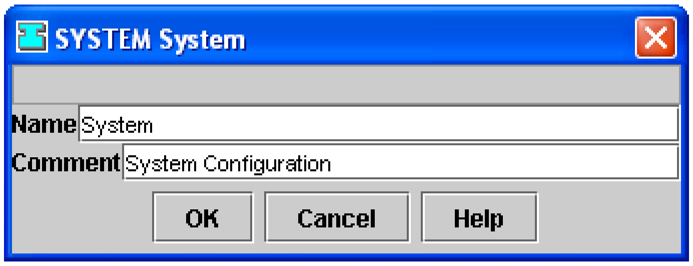


```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE System SYSTEM "http://www.holobloc.com/xml/LibraryElement.dtd" >
<System Name="System" Comment="System Configuration" >
  <Identification Standard="61499-1" />
  <VersionInfo Organization="Rockwell Automation" Version="0.3" Author="JHC" Date=
  <VersionInfo Organization="Rockwell Automation" Version="0.2" Author="JHC" Date=
  <VersionInfo Organization="Rockwell Automation" Version="0.1" Author="JHC" Date=
  <VersionInfo Organization="Rockwell Automation" Version="0.0" Author="JHC" Date=
  <Application Name="APP1" >
    <FBNetwork >
    </FBNetwork>
  </Application>
```

Figure 1.3b New System Screen

Step 2: Properties.

On the left side of the screen, there is a list. This list represents the default system configuration. The first entry is titled Properties. Under the Properties listing, there are three entries: Name/Comment, Identification, and Versions. The first entry, Name/Comment, gives the new system a name and adds any comments about it. The second entry, Identification, is used to add any extra information about the system (i.e. the standard used). The third entry, Versions, is used to add information on the system's author and the system version history. By double clicking on each of those entries, windows will pop up on screen to allow you to enter the proper information. First, double click on Name/Comment entry to bring up the window at the bottom screen as shown in Figure 1.4a.



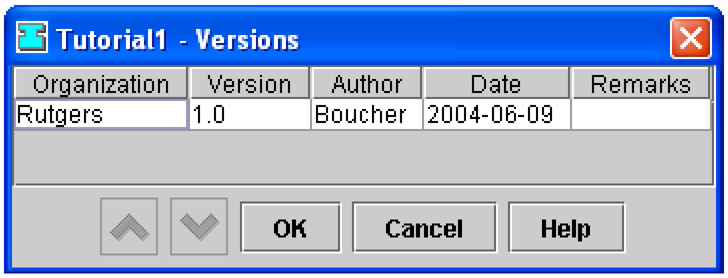
Name: System
Comment: System Configuration

Figure 1.4a Properties – Name/Comment Window

By inserting text into the Name field, the system can be given a name along with user comments. The default name is System. Give the new system a name. Be sure not to use any spaces or special characters. For the purposes of this tutorial, it will be named

Tutorial1. This name will not only be the name of the system but also the name of the file that will be saved. When the information is entered, click OK and the system will now have a new name.

Once the system has a new name, double click on the third entry, Versions. Here you will provide the name of the organization programming the system, the version number, the date, any comments, and the author's name can be added. If there are any other entries there, you can delete them by clicking on the row to highlight it, right click on the row, and selecting Cut. You can add an entry by right clicking within the window and selecting New and clicking on each field to insert the desired information. Click OK when finished so the information can be saved. When you are finished, the Versions tab will look something like Fig. 1.4b.



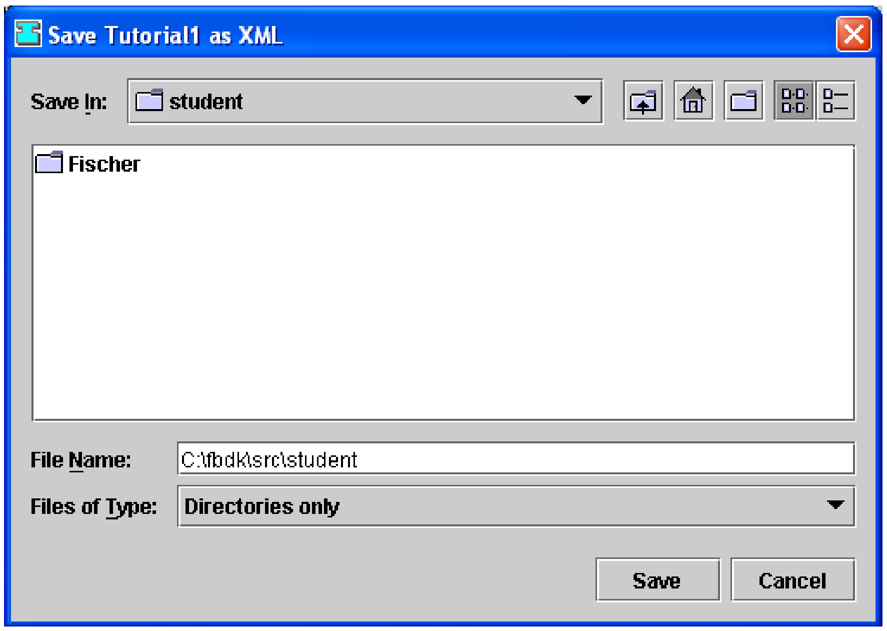
Organization	Version	Author	Date	Remarks
Rutgers	1.0	Boucher	2004-06-09	

Figure 1.4b Example Version Settings

Double clicking on the entry titled Identification will open up another window allowing you to enter information. However, for the purposes of this tutorial, nothing needs to be added there. The system now has a name as well as information on the system's version, the system's author, the date of the version, and the organization the system is for.

Step 3: Saving & Loading Systems.

It is important to constantly save the system after entering important information. Saving a system allows you to avoid losing any progress you have made on your system. To save a system, go to the menu bar, click on File -> Save As -> XML. All systems are saved as XML files. For the purposes of this tutorial, it is recommended to save these XML files under the file path C:\fbdk\src\student. If you want to separate your files from the other files already in that directory, you can create another subdirectory within student. Please save the Tutorial1 system. Figure 1.5a shows what the Save As screen will look like.

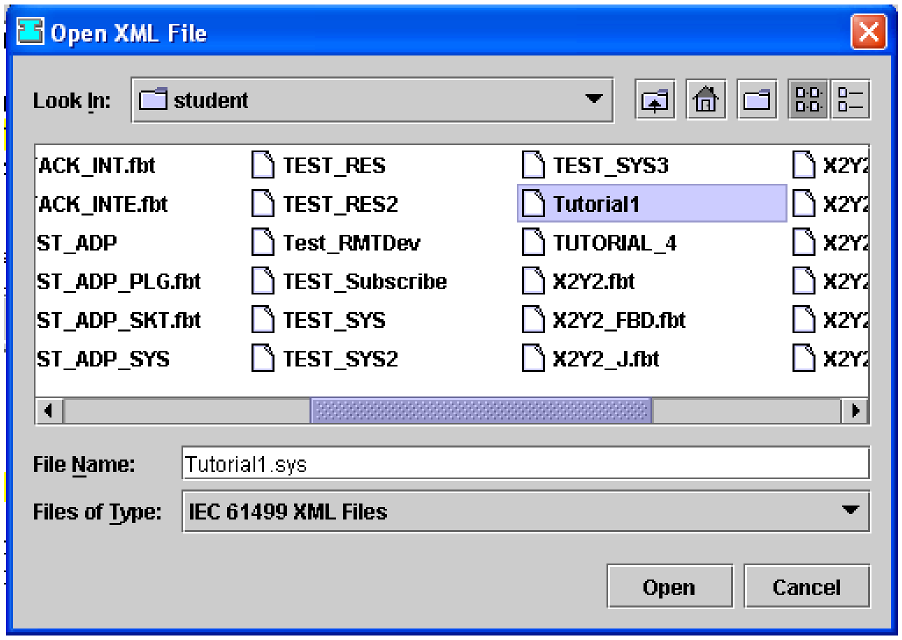


File Name: C:\fbdk\src\student

Files of Type: Directories only

Figure 1.5a Save System as XML Screen

To load a system, go to the menu bar click on File -> Open -> XML. A load XML file screen will appear. When set to IEC 61499 XML Files, the Files of Type drop down menu displays all of the XML files that can be loaded. Figure 1.5b shows the screen that appears to load up a XML file as well as the Tutorial1 system file that was just saved.



ACK_INT.fbt	TEST_RES	TEST_SYS3	X2Y
ACK_INTE.fbt	TEST_RES2	Tutorial1	X2Y
ST_ADP	Test_RMTDev	TUTORIAL_4	X2Y
ST_ADP_PLG.fbt	TEST_Subscribe	X2Y2.fbt	X2Y
ST_ADP_SKT.fbt	TEST_SYS	X2Y2_FBD.fbt	X2Y
ST_ADP_SYS	TEST_SYS2	X2Y2_J.fbt	X2Y

File Name: Tutorial1.sys

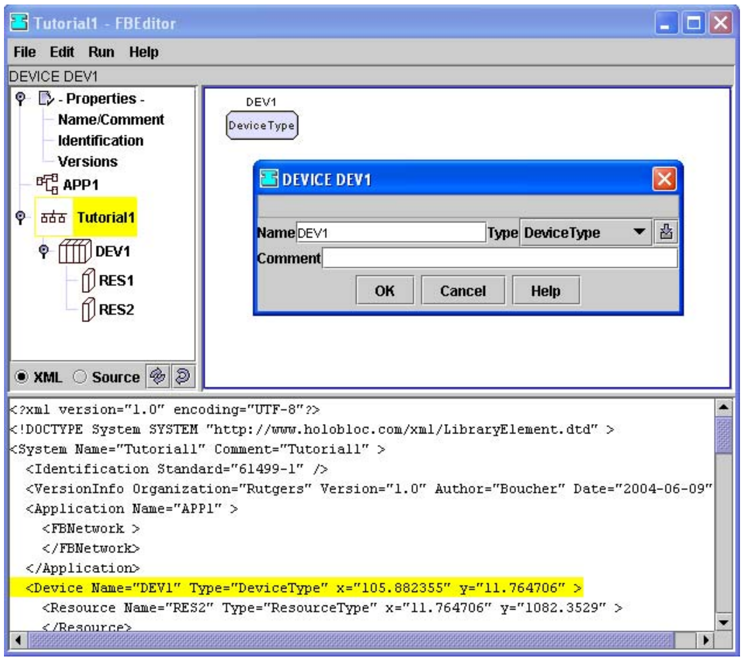
Files of Type: IEC 61499 XML Files

Figure 1.5b Open XML File Screen

Step 4: Defining Devices.

The next entry on the list to the left, below Properties, is titled APP1. More details on this entry will come later. The following entry is the system name, as defined in under the Name/Comments entry. Click on this name and the workspace will display the devices in the system. There should be a block named DEV1 with the name DeviceType. All systems are made up of devices. Devices are made up of one or more independent processors with their own memory, I/O, and thread of control. These independent processor units are called resources. A PLC with one processor is roughly equivalent to a device with one resource. In this case the device is shown with two resources.

By right clicking in the workspace, more devices can be added by selecting New -> Device. For this tutorial, only one device is needed; thus, the current device, DEV1, needs to be edited. Right click on the name at the top of the block, DEV1, and select Edit. As Figure 1.6 shows, a window pops up with a set of text fields and a drop down menu. The current type of device is DeviceType, which must be changed; the program will not run as a DeviceType. Click on the drop down menu titled Type and select FRAME_DEVICE to change the type to FRAME_DEVICE. This type will make the program run in a window. When finished, click OK and you will see the changes made to the device. Incidentally, to delete a device, you can right click on the name of the device and select Delete. Notice how the block changes, there are additional listings within it now. You can right click on the parameters within the block titled BOUNDS and GRID to configure the device. Select Connect to give them base values of [0,0,0,0] and [0,0].



Name DEV1 Type DeviceType
Comment

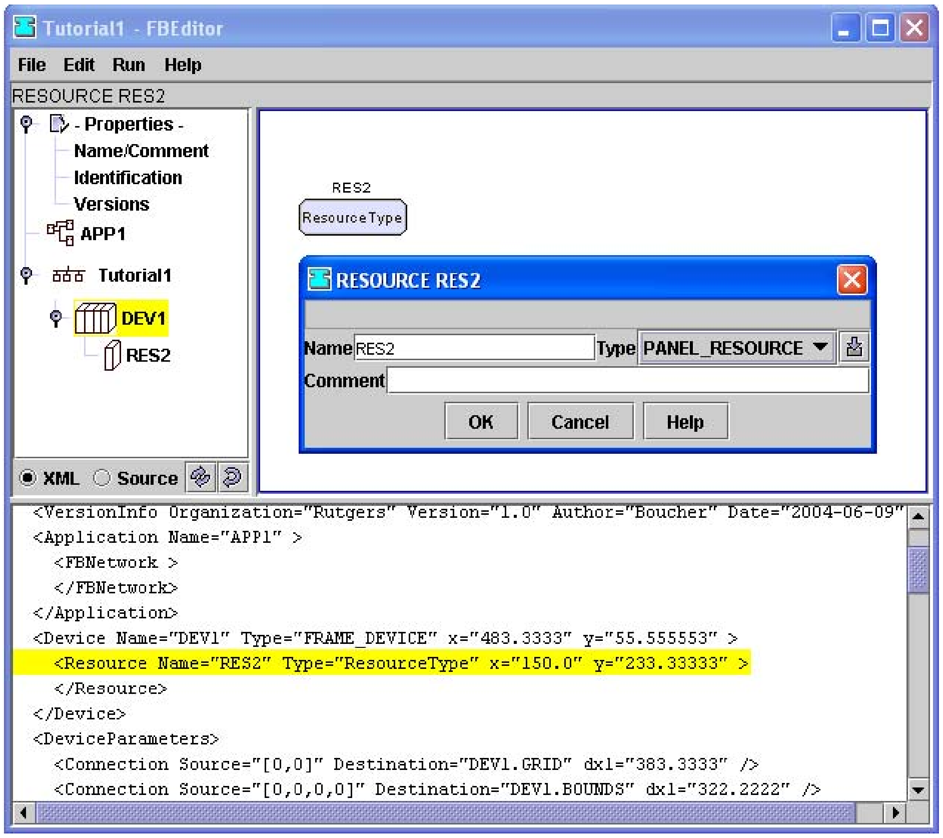
```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE System SYSTEM "http://www.holobloc.com/xml/LibraryElement.dtd" >
<System Name="Tutorial1" Comment="Tutorial1" >
  <Identification Standard="61499-1" />
  <VersionInfo Organization="Rutgers" Version="1.0" Author="Boucher" Date="2004-06-09"
  <Application Name="APP1" >
    <FBNetwork >
    </FBNetwork>
  </Application>
  <Device Name="DEV1" Type="DeviceType" x="105.882355" y="11.764706" >
    <Resource Name="RES2" Type="ResourceType" x="11.764706" y="1082.3529" >
    </Resource>
```

Figure 1.6 Programming Devices

These two device parameters adjust program window display can be defined. The first listing, BOUNDS, has four values. The first two, x and y, defines where the window will appear with respect to the top left corner of the screen when the program is run. The last two, width and height, defines the initial size of the window. To change the values after connecting the parameter with a base value, simply double click on it. A window will pop up allowing you to edit the parameter. For this tutorial, set the BOUNDS parameter to [100,100,150,200], but feel free to change them. Click OK to finish and update the parameter. The second listing, GRID, determines the grid size within this window. For this tutorial, and in general, [1,1] will suffice. Again, double click on the base value to have a window pop up to edit the parameters. When finished, click OK, and the device parameter will be updated. The system configuration now has a defined window and layout for when the program is run. Now the resources within device can be defined.

Step 5: Defining Resources

Before developing this system, click on DEV1 on the left side of the screen. Figure 1.7 shows that there is one resource currently within the previously defined DEV1, named RES2. RES2 is a resource. Devices are made up of resources, which are used to contain and execute the function blocks necessary for the program. Right click on the resource block's name, RES2, select Edit, and a window will pop up allowing you to edit the name and resource type of the resource. This is also shown in Figure 1.7.



Name RES2 Type PANEL_RESOURCE
Comment

```
<VersionInfo Organization="Rutgers" Version="1.0" Author="Boucher" Date="2004-06-09"
<Application Name="APP1" >
  <FBNetwork >
  </FBNetwork>
</Application>
<Device Name="DEV1" Type="FRAME_DEVICE" x="483.3333" y="55.555553" >
  <Resource Name="RES2" Type="ResourceType" x="150.0" y="233.33333" >
  </Resource>
</Device>
<DeviceParameters>
  <Connection Source="[0,0]" Destination="DEV1.GRID" dx1="383.3333" />
  <Connection Source="[0,0,0,0]" Destination="DEV1.BOUNDS" dx1="322.2222" />
```

Figure 1.7 Programming Resources

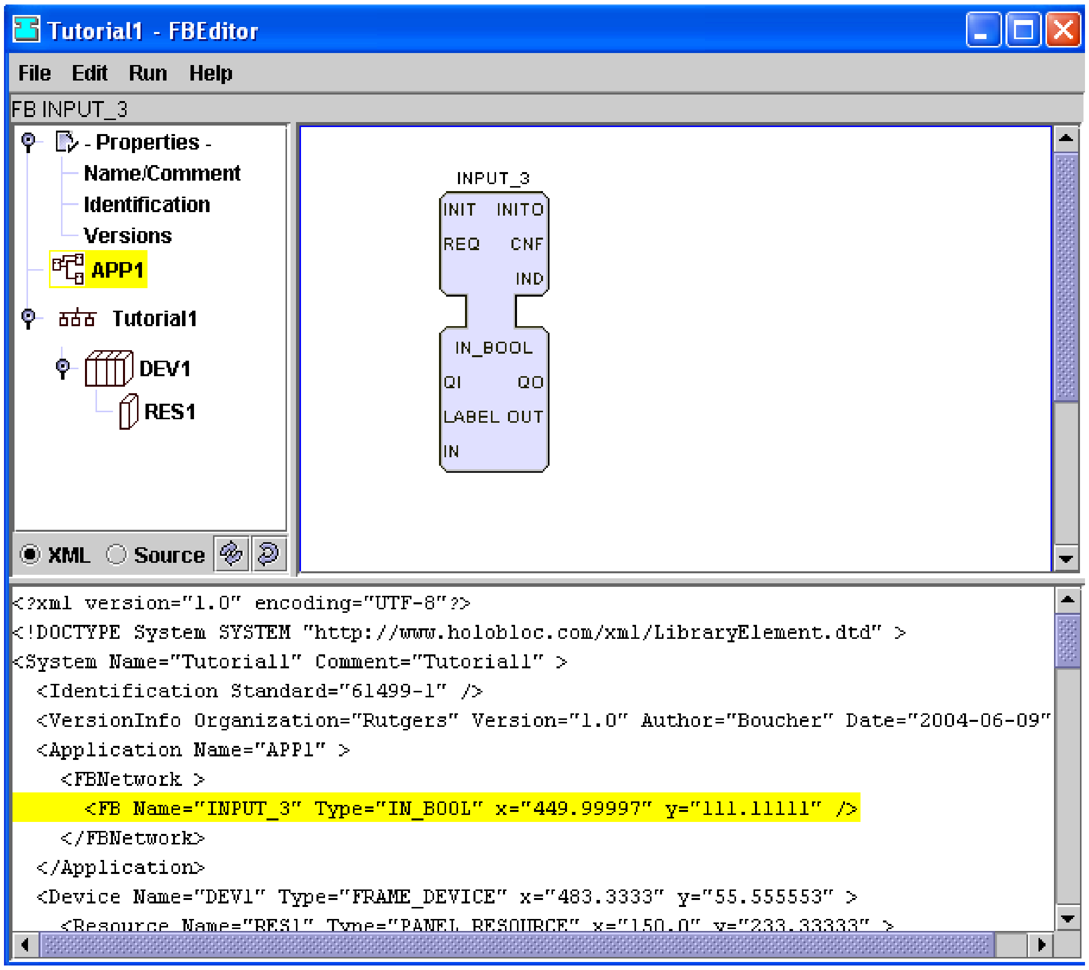
For this tutorial, only one resource is needed. You can choose to give it a new name, such as RES1; but it's not required. Click on the drop down menu named Type and select PANEL_RESOURCE. Similar to defining the device as a FRAME_DEVICE, the resource must have this type in order for the program to run. Click OK when finished and the resource has been edited. Deleting resources is similar to deleting devices, simply right click on the name of the resource and select Delete. To add resources, right click on the workspace, and select New -> Resource. Now that the device and resource for this tutorial is defined, the system can now be programmed. Save your system before doing so.

Step 6: Adding Function Blocks

The next step is to program the application using function blocks. There are three types of function blocks: 1) basic function blocks, 2) composite function blocks, and 3) service interface function blocks. In this exercise we will use only basic function blocks. Basic function blocks represent elemental functions, such as logical and mathematical operations. Many basic function blocks are defined as part of the IEC 61499 standard. Composite function blocks are constructed by combining basic function blocks and/or user-defined function blocks in a sequence to perform higher-level control operations. Service interface function blocks define communication among resources and devices. The programming of basic function blocks and composite function blocks will be described in a later tutorial.

The application, titled APP1, is where the programming will take place; it is where the source code will be written. Click on the entry titled APP1 and the workspace will be completely empty. When starting a new system, there will be one application entry and it will be empty. This tutorial will program the system described in Figure 1.1.

To start programming the system described in Figure 1.1, try adding a function block for one of the input switches of the PLC program. According to the situation, the switch is either on or off; so the function block must represent a Boolean input. Such a standard input block comes with the software. To add a new function block, right click on the empty workspace, select New -> FB. Click on the button next to the pull down menu named Type to add a new function block type. Any blocks previously called up while the program has been used will be listed in the pull down menu named Type. Clicking on the button next to the pull down menu allows searching through the directories for the desired block. For this example, the necessary function block type is a Boolean input. A screen will pop up to load a new function block type. This is similar to loading a system. The function block type needed can be found through this path: fbdk-> src -> hmi. The file for this block is IN_BOOL.fbt. Click on it and it will not only be the type for that block, but it will now be in that pull down menu. Now this new function block needs a name. For the name of a function block, be sure not to use any spaces (use underscores instead) or special characters. For this tutorial, one suggestion would be to call it INPUT_3. Give it that name or any other name now. Click OK and there will now be a Boolean input function block named INPUT_3 in the workspace, as shown in Figure 1.8.



```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE System SYSTEM "http://www.holobloc.com/xml/LibraryElement.dtd" >
<System Name="Tutorial1" Comment="Tutorial1" >
  <Identification Standard="61499-1" />
  <VersionInfo Organization="Rutgers" Version="1.0" Author="Boucher" Date="2004-06-09"
  <Application Name="APP1" >
    <FBNetwork >
      <FB Name="INPUT_3" Type="IN_BOOL" x="449.99997" y="111.11111" />
    </FBNetwork>
  </Application>
  <Device Name="DEV1" Type="FRAME_DEVICE" x="483.3333" y="55.555553" >
    <Resource Name="RES1" Type="PANEL_RESOURCE" x="150.0" y="233.33333" >
```

Figure 1.8 Workspace After Adding an IN_BOOL Function Block

To move a function block in the workspace, simply click on the name of the function block and drag it to the desired location. Editing and deleting a function block can be done by right clicking on the function block's name and selecting the proper choice in the resulting menu.

EXERCISE: In this situation, another Boolean input function block is needed as well as two Boolean output function blocks (lights are either on or off; thus, they are Boolean outputs). The file name for the Boolean output function block is OUT_BOOL.fbt and it is in the same directory as the IN_BOOL.fbt file. According to the logic of the problem, AND and NOT functions are involved, so function blocks for them are needed. Insert these five additional function blocks on your own. The function blocks for AND and NOT functions are in the directory fbdk -> src -> math, the files are FB_AND.fbt and FB_NOT.fbt respectively. You may need to move the blocks to see them all clearly. This can be done by clicking and dragging them in the workspace. Save the system when finished

Figure 1.9 shows all of the blocks needed for this system.

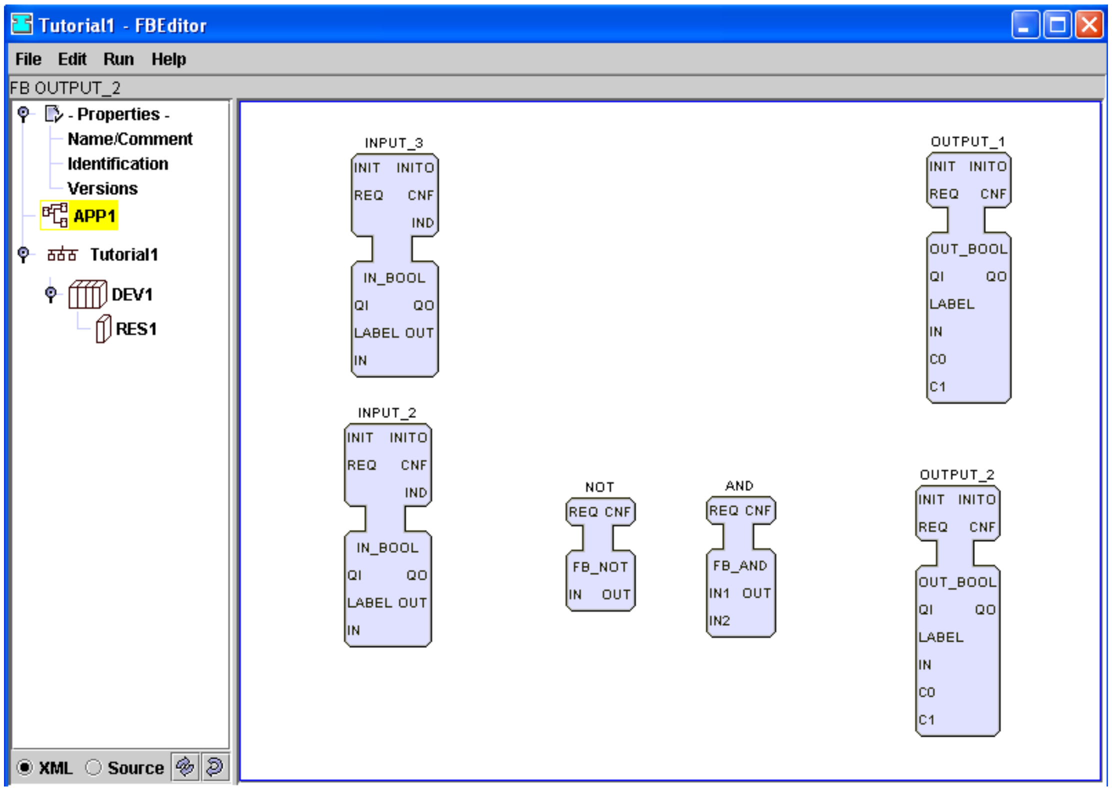


Figure 1.9 Function Blocks Required for System

Step 7: Events

Function blocks are controlled by events. Events are shown at the top, or head, of the block. Think of events as pulses. When a function block executes an event, it sends out a pulse to any function block connected to the output of that event. Whenever an event reaches a function block, data that corresponds with the arrival of that event gets read, the algorithm within the function block is executed based on the type of event, and an event is outputted from that function block to the next one. Click on APP1 and notice in the workspace that none of the function block events are connected with any other function block events.

Connecting function block events is simply done by holding the Alt key on the keyboard, clicking on the event of one function block, and dragging the cursor to a second event on another function block. A connection will appear between the events in the workspace. For example, an event is necessary between the IND event in INPUT_3 and the REQ event in NOT. IND stands for indication; whenever there is a change in INPUT_3 (switch is turned on or off), this event will occur and go to its destination. REQ stands for request; whenever an event comes to REQ, the algorithm associated with it is activated. What is needed is that whenever there is a change in INPUT_3, NOT should be activated and executes its algorithm. Hold the Alt key, click on the IND event on the INPUT_3 function block, and then drag the cursor to the REQ event on the NOT function block. A line will appear connecting the two events, as seen in Figure 1.10.

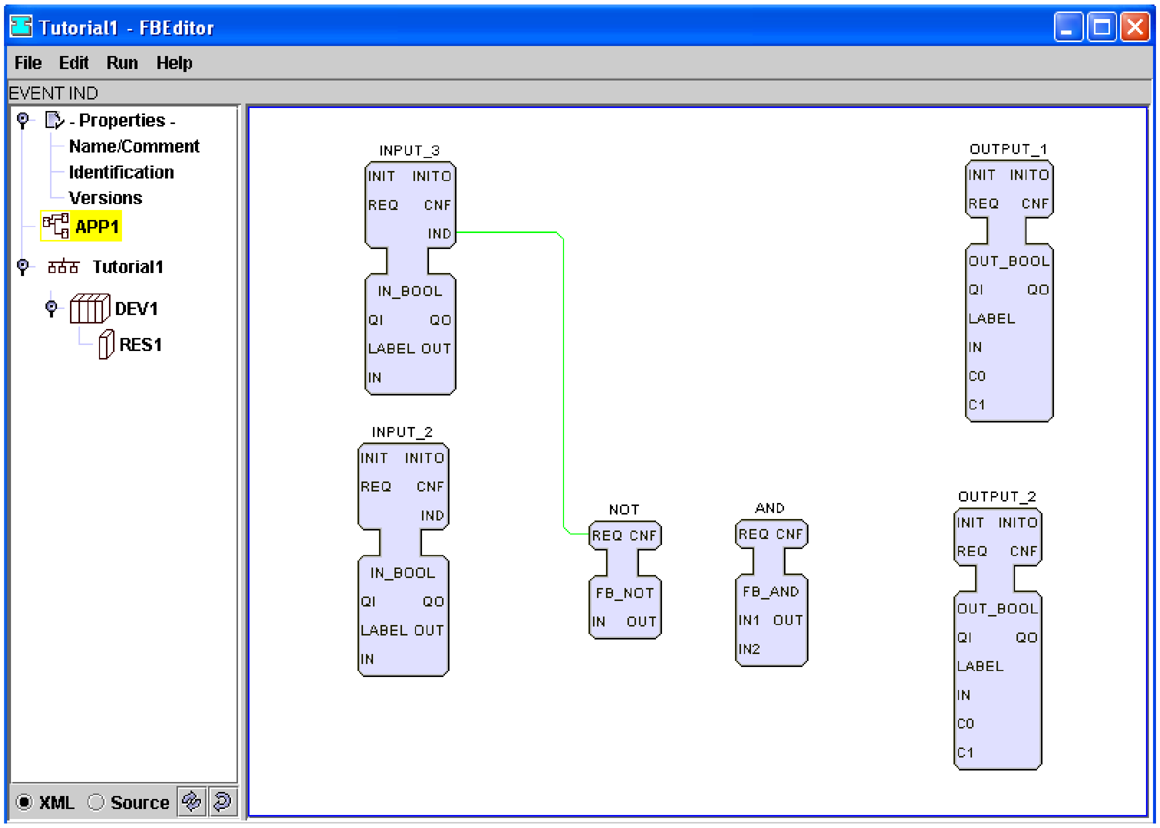


Figure 1.10 Workspace After Connecting Two Function Block Events

Event connections are always displayed as green lines. Deleting a connection between events is done by clicking on the connection line itself. The connection will turn red; it can then be deleted by holding the Alt key and pressing the Delete key or by right clicking on the connection and clicking on Delete.

It should be noted that both input and output function blocks have additional events named INIT and INITO. These events are called initialization events; they are used in the initialization of the program and ensure that the blocks will be active and in their initial states when the program is started. Most function blocks have these events, so it is recommended that those events are always connected.

EXERCISE: Connect the remaining necessary events. Use Figure 1.11 as your guide. Connect the INIT and INITO events by starting with the INITO event in INPUT_3 with the INIT event in INPUT_2. Connect the INITO event in INPUT_2 with the INIT event of OUTPUT_1; and then connect the INITO event in OUTPUT_1 with the INIT event of OUTPUT_2. Connect the IND event in INPUT_3 with the REQ event for OUTPUT_1, as well. Connect the IND event in INPUT_2 with the REQ event for NOT. Connect the CNF event in NOT with the REQ event in AND. Connect the CNF event in AND with the REQ event in OUTPUT_2. Ignore the INIT event for INPUT_3 and INITO event for OUTPUT_2. Also ignore the REQ and CNF events for the two input blocks as well as the CNF events for the two output blocks. The final result should be similar to Figure 1.11.

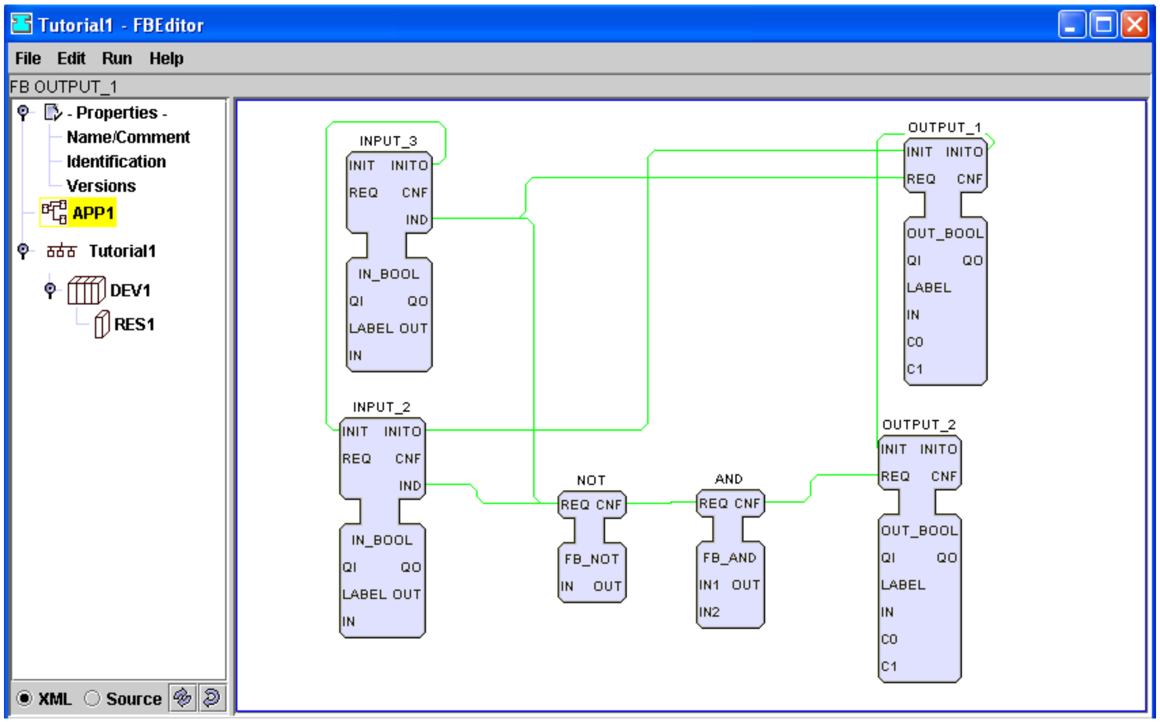


Figure 1.11 Workspace After All Event Connections Are Made

If the connections get too visibly confusing, right click on the main screen and select Redraw -> Connections. That should make the diagram clearer.

Step 8: Connecting Data

The data in a function block is represented in the bottom, or body, of the function block. Data gets read when an event arrives, is processed through an algorithm, and a result is outputted. Like events, data needs to be connected between function blocks. For this tutorial, the data outputted from the two input blocks need to be used with the logic to control the final output. Connecting data is done with the same method used for connecting events.

According to the logic, OUTPUT_1 is only affected by INPUT_3. Therefore, whatever is outputted from INPUT_3 should go into OUTPUT_1. Connect the OUT in INPUT_3 to the IN in OUTPUT_1 using the same method to connect events. Hold the Alt key, click on OUT in INPUT_3, and drag the cursor to IN in OUTPUT_1. That part of the logic is now successfully implemented in the program, as shown in Figure 1.12. Data connections are always displayed as blue lines, and can be deleted in the same way as event connections.

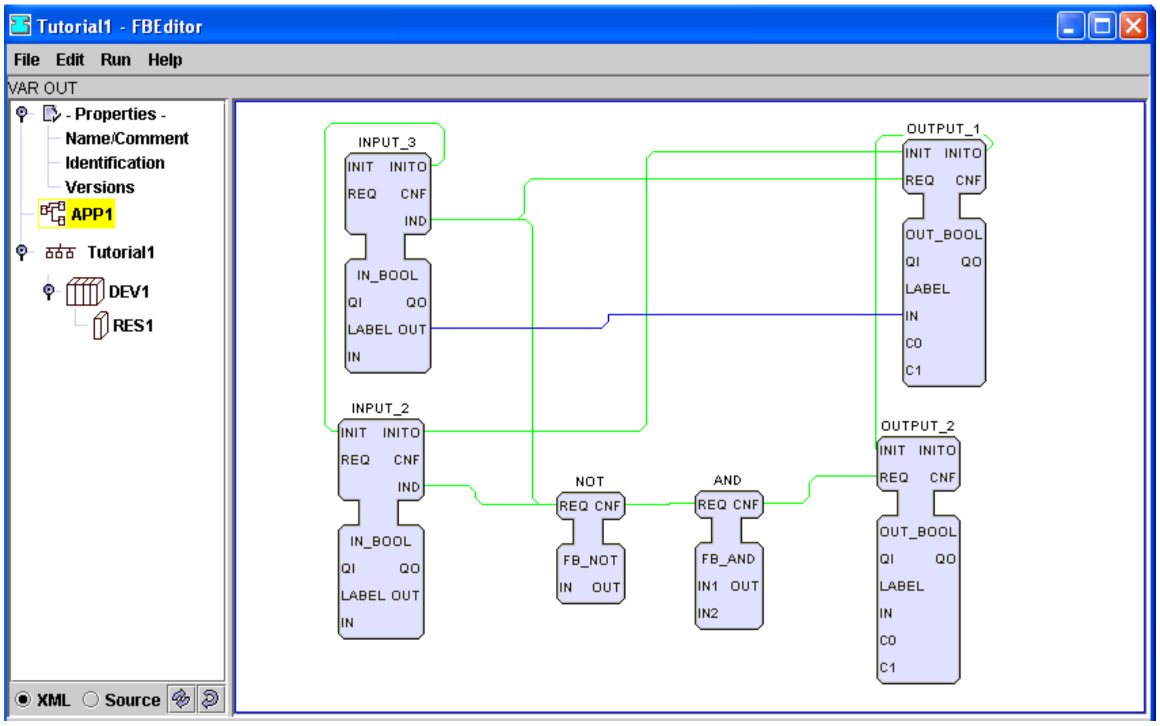


Figure 1.12 Workspace After First Data Connection

EXERCISE: Make all of the necessary data connections to complete the logic for the rest of the program. Remember that data, like events, can go to multiple destinations. When finished, all the necessary data connections will be made and the system should look similar to Figure 1.13.

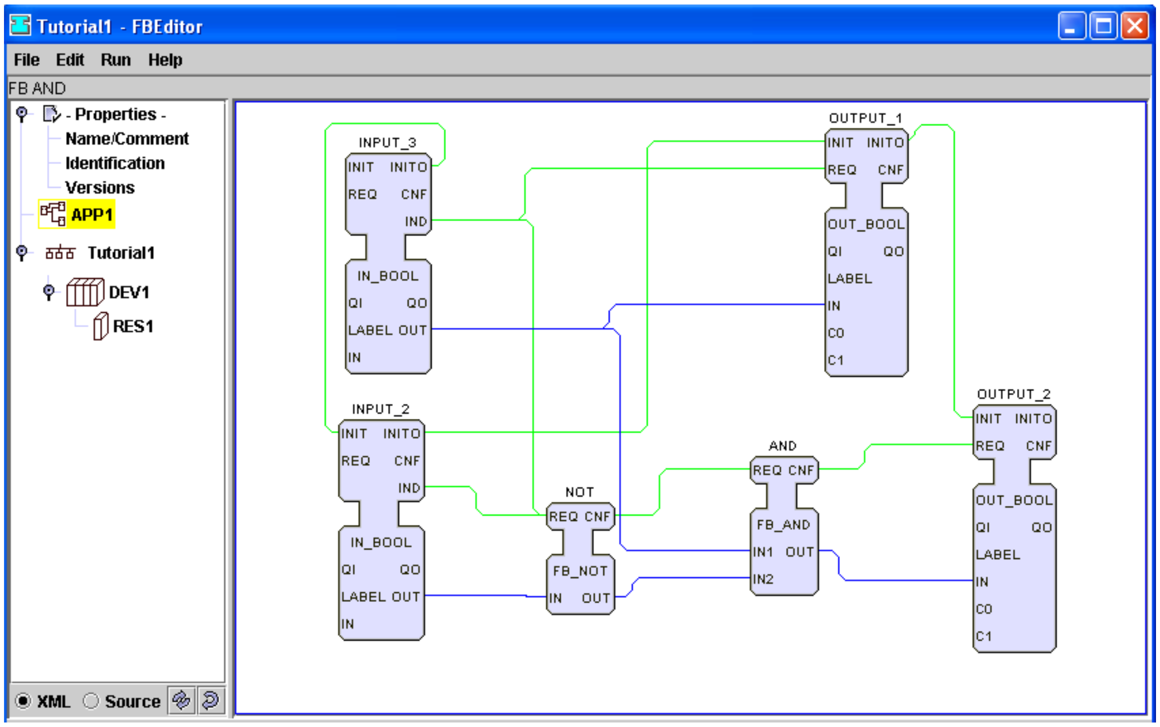


Figure 1.13 Workspace with Event and Data Connections

Step 9: Assigning Data

While all the data connections are made, the system is still not complete. There are multiple data inputs that have nothing assigned to it. While events need only to be connected to other events, some data inputs require their own values. For example, both the input and output function blocks have a data input called QI. QI activates the block when it is defined as 1 so that the function block will run when the program is executed. By setting QI=0, the user prevents the function block from executing when the program is running. This is similar to commenting out a line of code in a program. Therefore, all of the QIs should be defined as 1. To do this, right click on the QI data input for INPUT_3, and select Connect. There should be a display similar to the one shown in Figure 1.14.

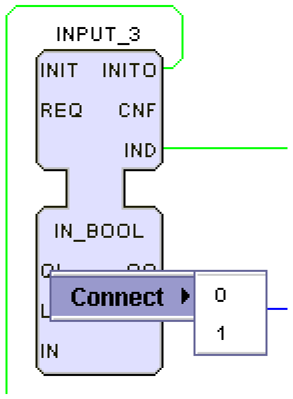


Figure 1.14 Assigning A Data Input

Select 1 for QI and the data input will display a 1 for that data input. To edit this value, double click on the value of the data input, and a window will pop up allowing the parameter to be edited. Repeat this process for the other data inputs titled QI in the system. Be careful not to assign more than one source to one destination, otherwise there will be an error.

Other data inputs can be defined by right clicking on the data input name on the function block, select Connect, and the base value that appears. Editing a data input value is done the same way as editing a device parameter. Double click on the data input value, a window will pop up, and the parameter can be edited. If the base value is "", that means the data input is a string; insert text in between the quotation marks. Some function blocks have specific data inputs, such as the OUT_BOOL function block. The data inputs C0 and C1 refer to color when the output is a 0 or a 1. Therefore, the value should correspond to a red-green-blue value (i.e. [0,0,0] is black, [255,0,0] is red, etc.).

EXERCISE: Connect and assign values to the data inputs for the remaining input and output function blocks. Give labels to each function block, so when the program is run there will be a name assigned to that input or output. For the input function blocks, set IN to 0, since the switches will be off when the program first starts. For the output function blocks, choose a different color each for C0 (when the light is off) and C1 (when the light is on).

Once finished with this exercise, the program should look similar to Figure 1.15. Do not worry about QO, there is no use for that data output in this situation. The code in program is finished, and the logic of the situation is realized.

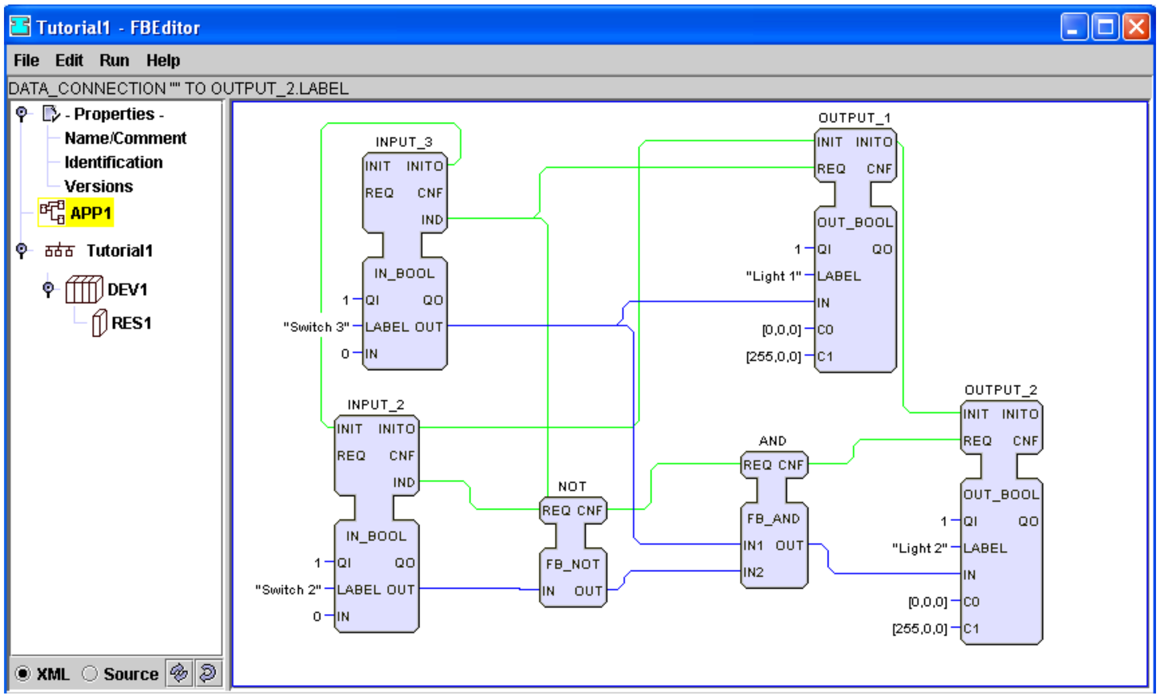


Figure 1.15 Completed Data and Event Configuration

Step 10: Mapping and Running the Program

Check for any errors made in the program. Any errors that do occur will appear in a text box at the top of the screen. If there are no errors, this program can now be mapped to the resource that was previously defined. Function blocks have to be downloaded (mapped) to the resource on which they will run. To map a function block, right click on the name of function block and select Map. The following menu will ask where to map the function block. While it is possible to map it to the device directly when there is only one resource on the device, which may cause errors in the program. Therefore, map it to DEV1.RES1 as shown in Figure 1.16.

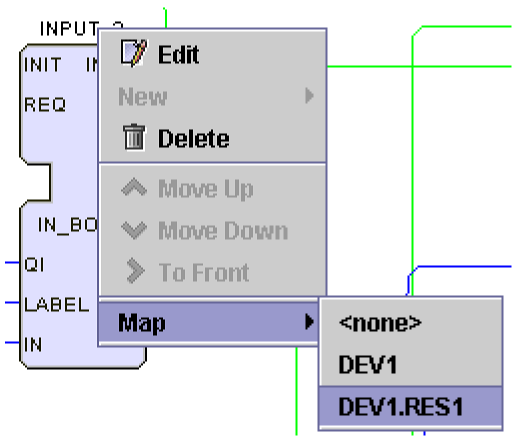


Figure 1.16 Mapping the INPUT_3 Function Blocks to RES1

Click on RES1 on the list left of the workspace now. The block should appear there just as it was in APP1. The only change that may be made is to move the block, but that is a minor change.

EXERCISE: Map the other blocks in APP1 to the resource RES1 in device DEV1.

Click on RES1 and notice that all of the blocks in APP1 are in RES1 with the same connections and data assignments. Also notice that there is an E_RESTART block present. This block is in all resources, it controls the starting and stopping of the program. The E_RESTART block only has three events: COLD, WARM, STOP. The COLD event initially starts the program, similar to turning on a computer after it has been off. The WARM event starts the program over after it has already been started, similar to restarting a computer. The STOP event ends the running of the program, similar to turning a computer off. For this program, only the COLD event is needed; connect that event with the INIT event of the INPUT_3 block. Once that is complete, the resource should look similar to Figure 1.17. Save the system now, just before running it.

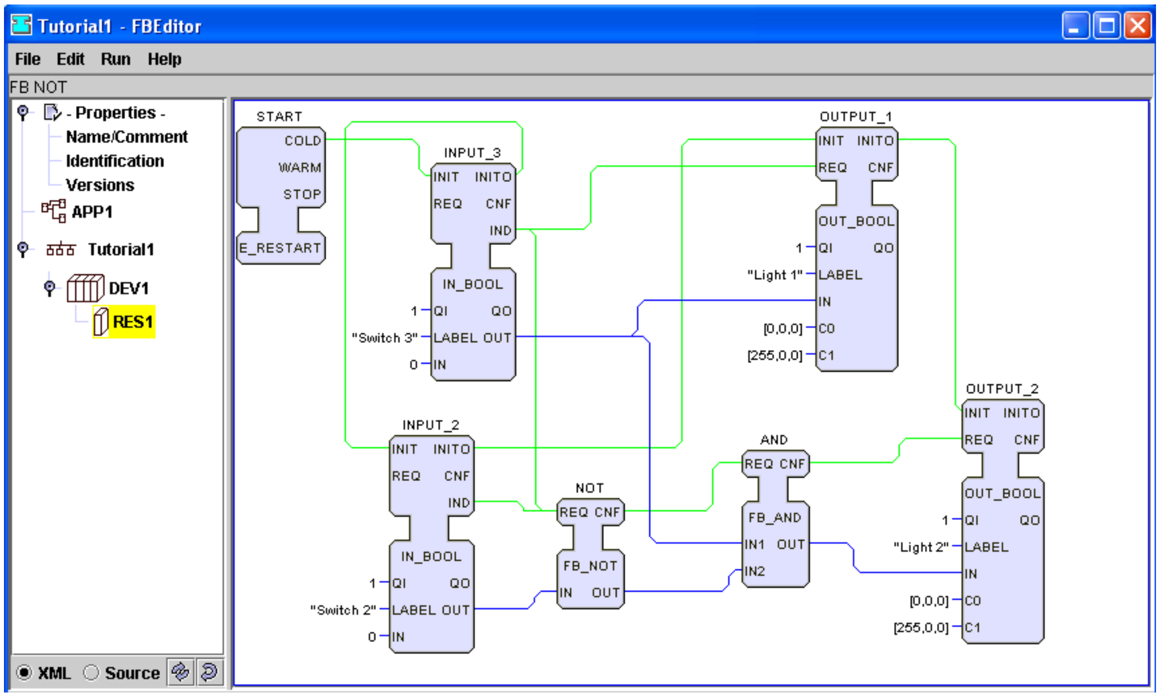


Figure 1.17 System Downloaded to a Resource

To run a system, go to the menu bar and click on Run -> Launch -> the system name, Tutorial1. For this system, the window shown in Figure 1.18 should appear at the defined coordinates with the defined dimensions from Step 3.

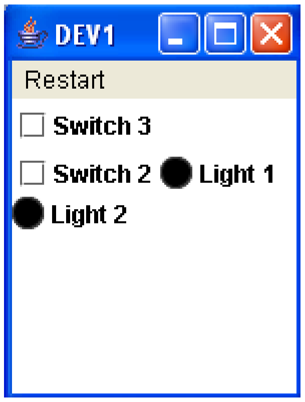


Figure 1.18 Device Running the Program

The two switches, being Boolean inputs, appear as checkboxes. The two lights, being Boolean outputs, appear as lights. Check the boxes in all possible combinations to see if the program follows the necessary logic and works correctly. Figure 1.19 shows what should happen if all 10 steps were followed properly.

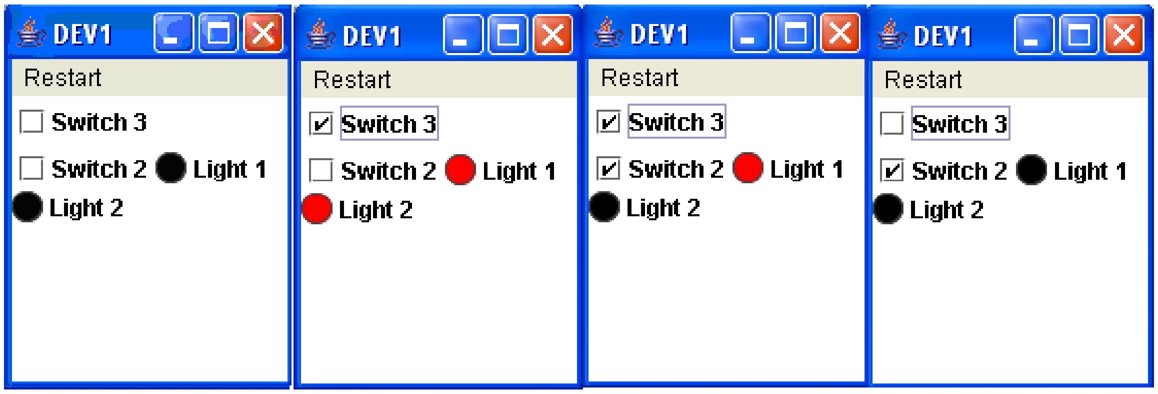


Figure 1.19 Various Combinations Testing the Program

If the program does not run correctly, there is something wrong with the program. Make any changes to correct these errors in APP1, but the program will not run with these changes until the original program is deleted from the resource and the updated program is mapped again to that resource. This ends the tutorial on using function blocks in the IEC 61499 standard.

1.3 Summary

In summary, this tutorial has shown the following sequential steps on basic use of the IEC software. To design a system on this software:

1. Configure the new system by adding properties.
2. Define devices by their type (usually FRAME_DEVICE) and their bounds and grid parameters.
3. Define resources in the device(s) and their type (usually PANEL_RESOURCE).
4. Add function blocks that will be used.
5. Connect the events of the function blocks
6. Connect the data inputs and outputs of the function blocks.
7. Define the necessary data inputs for certain function blocks.
8. Map the function blocks to the proper resource(s) in the device(s).
9. Connect the E_RESTART function block to the other function blocks in the resource(s).
10. Click on Run at the menu bar, select Launch and then select the system's name to launch the program.

Do not forget to save the system by clicking File -> Save As -> XML.

IEC TUTORIAL 2: COMPOSITE FUNCTION BLOCK TUTORIAL

2.1 Introduction

In the IEC 61499 standard, there are three main types of function blocks: 1) basic function blocks, 2) composite function blocks, and 3) service interface function blocks. Basic function blocks describe elemental logical and mathematical operations. They consist of events; data; algorithms to process data; and an execution control chart to control what algorithms are used based on what events are inputted into the block. Composite function blocks consist of multiple basic function blocks, but do not have an execution control chart. These blocks are useful for large programs, since many functions can be combined into a single block.

SITUATION: Create a composite function block called ON_OFF_OC that will take two Boolean inputs and yield one Boolean output. The notation used to name the composite function block type is user determined. Here we indicate that it is for on/off operation and consists of one open (O) and one closed (C) contact. The ladder logic for this function block is as follows in Figure 2.1a.

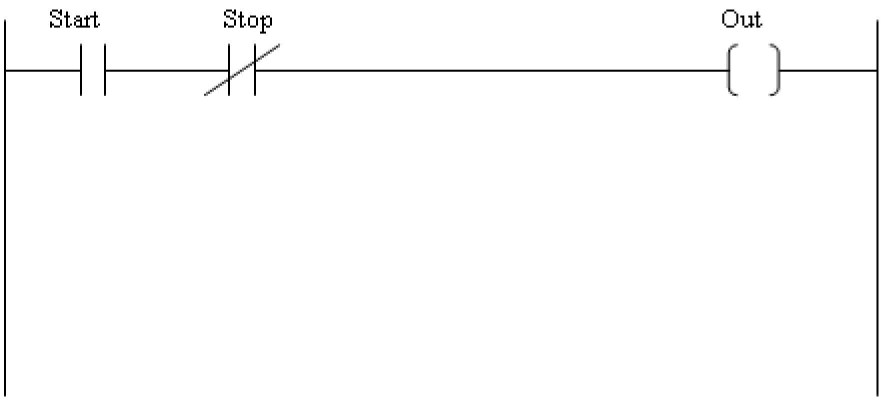


Figure 2.1a Ladder Logic for Composite Function Block

Before we begin to create a composite function block, let us look at how the logic of Figure 2.1a would be implemented using basic function blocks. This is shown in Figure 2.1b. The start and stop switches and the output lamp function blocks are used to emulate inputs and outputs. The control logic is given by the FB_NOT and FB_AND function blocks. The logic is as follows:

$$\text{Start} \bullet \overline{\text{Stop}} = \text{Out}$$

Therefore, a composite function block to execute this logic is composed of one NOT function block and one AND function block. This is the composite function block, ON_OFF_OC, which we will create in this tutorial.

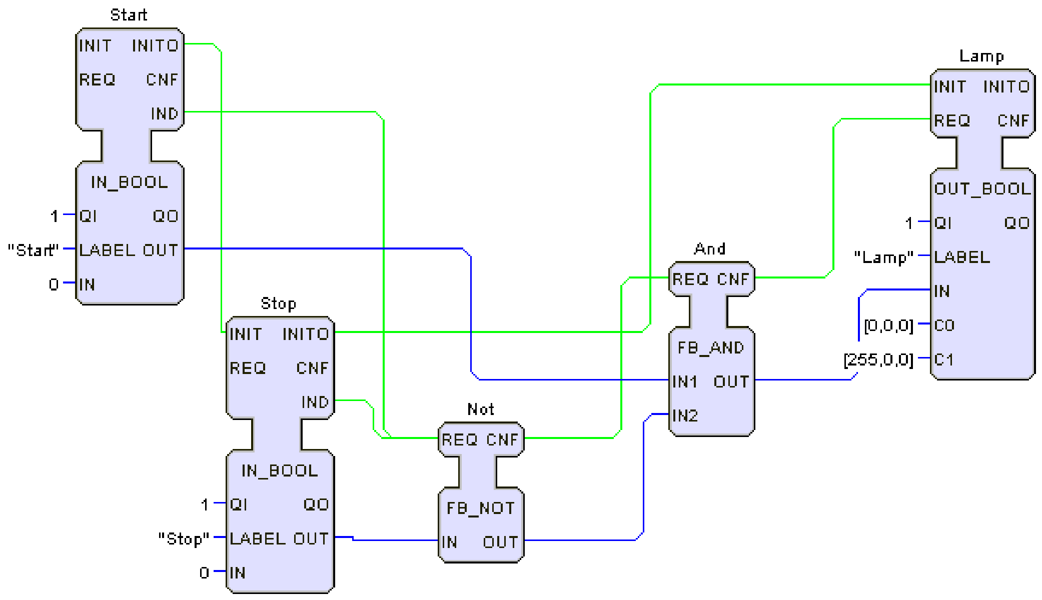
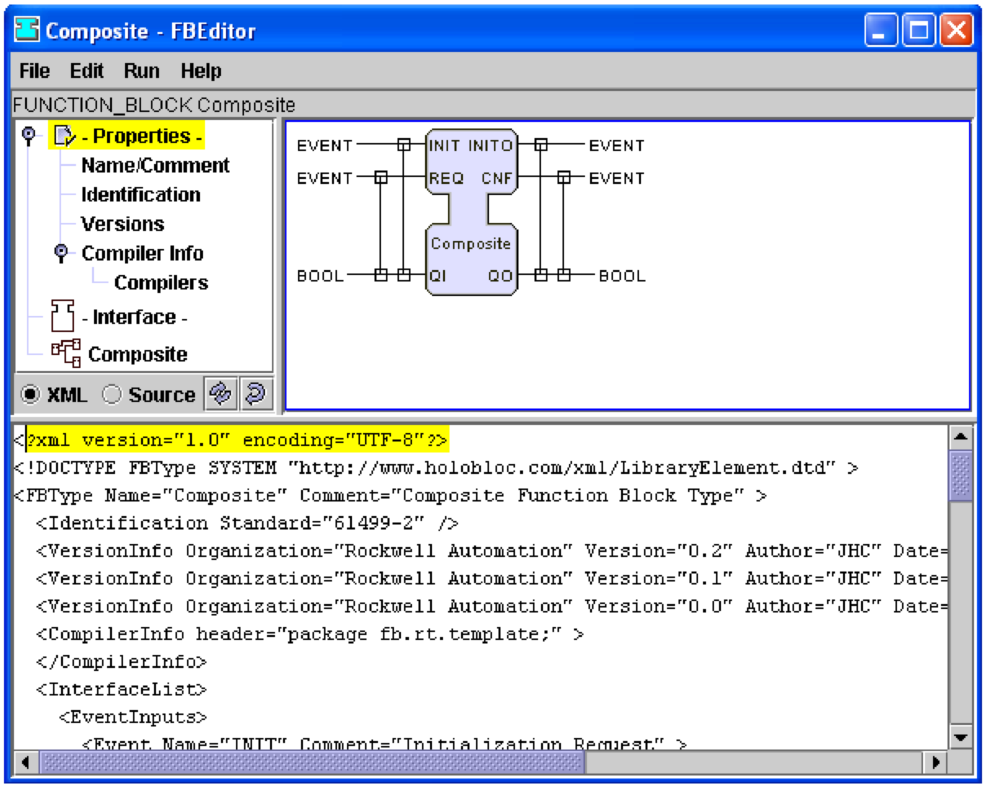


Figure 2.1b Function Block Equivalent of Figure 2.1a

2.2 Steps in Creating a Composite Function Block

Step 1: Creating a New Composite Function Block

Go to the menu bar, click on File -> New -> FB Type -> Composite to create a new Composite Function Block. The composite function block Composite will then appear, already with input data, output data, input events, and output events. The new window should appear like Figure 2.2.

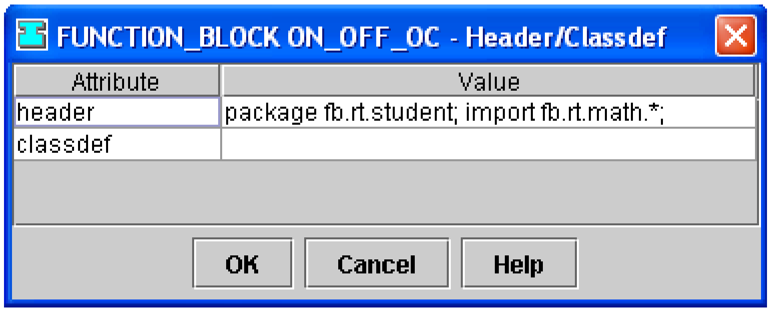


```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE FBType SYSTEM "http://www.holobloc.com/xml/LibraryElement.dtd" >
<FBType Name="Composite" Comment="Composite Function Block Type" >
  <Identification Standard="61499-2" />
  <VersionInfo Organization="Rockwell Automation" Version="0.2" Author="JHC" Date=
  <VersionInfo Organization="Rockwell Automation" Version="0.1" Author="JHC" Date=
  <VersionInfo Organization="Rockwell Automation" Version="0.0" Author="JHC" Date=
  <CompilerInfo header="package fb.rt.template;" >
  </CompilerInfo>
  <InterfaceList>
    <EventInputs>
      <Event Name="INIT" Comment="Initialization Request" >
```

Figure 2.2 Default New Composite Function Block

Step 2: Setting the Properties

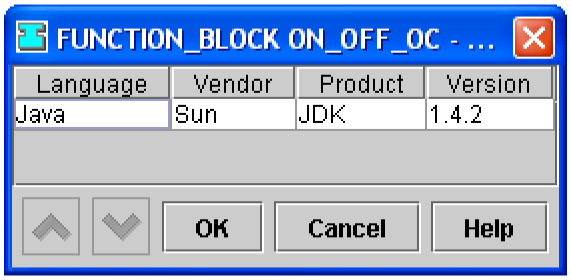
On the left side of the screen, there is a list of entries for this new composite function block. The first entry is called Properties, and it has five entries below it. Notice that the first three entries listed are exactly the same for the properties of a system: Name/Comments, Identification, and Versions. For now, give this composite function block the name ON_OFF_OC and input the version information in the same manner done in IEC Tutorial 1. To do that, simply double click on the desired entry and input the necessary information. Click OK on the window when finished with each.



Attribute	Value
header	package fb.rt.student; import fb.rt.math.*;
classdef	

Figure 2.3 Properties - Compiler Info Window

There is fourth entry called Compiler Info with an entry listed below called Compilers, which is extremely important. This composite function block can not be used unless it is ultimately compiled. Therefore, a compiler must be defined. For this tutorial, this composite function block will be made up of existing basic function blocks. Therefore, the file-path to these basic function blocks also must be defined for the compiler. This information must be typed into the header field. First, double click on the entry called Compiler Info and a window will pop up with text fields for the compiler header and class definition. For this tutorial, only the header needs a definition. The compiler is defined as (without the quotes) "package fb.rt.student;". The function blocks needed are in the math folder in the src directory. To define that, put in the header (without the quotes) "import fb.rt.math.*;". Any function blocks from that directory can now be used. The field should be like Figure 2.3. Now click on OK. The Compiler Info will be updated. If it is not updated, simply look at the XML code at the bottom of the screen for the entry that begins with "<Compiler Info". The information for the compiler can be made there as well; just be sure to parse the new XML code by clicking on the button with a circle of arrows (or Alt+S). Now click on the entry called Compilers. A window will pop up with no listings in it. Right click in the empty space and select New to add a compiler. It should look like Figure 2.4.



Language	Vendor	Product	Version
Java	Sun	JDK	1.4.2

Figure 2.4 Properties - Compiler Window

Step 3: Interface – Events & Data Points

The next entry on the list to the left of the screen after the entries under Properties is called Interface. Click on Interface and this workspace will display the composite function block itself. This is where events, data inputs, and data outputs can be created, deleted, and modified. The default new composite function block already has four events: INIT and INITO events for initialization, a REQ event for requesting of information, and a CNF event for confirmation. In general, there should always be INIT and INITO events for function blocks that retain internal information in order to make sure the function block does not use data from previous program runs. However, this function block only has data coming in from other function blocks; thus, the INIT and INITO events are not necessary. The reader can confirm this by looking at Figure 2.1b. The FB_NOT and FB_AND function blocks do not require an INIT event. Remove the INIT and INITO events by right clicking on each event name and select Delete. This composite function block will be requesting events for data, so there does not need to be any change to REQ. In addition, data from the composite function block will be sent along with requesting events, so there needs to be a confirmation event that the data is received. As a result, there is no need to change CNF.

However, there is only one data input and one data output currently for this default function block. QI is the input and QO is the output. Generally, they are important for some function blocks to actually function in a program; but in this case they are not necessary. Look at Figure 2.1b to confirm that they are not required for the FB_NOT and FB_AND function blocks. Therefore, these data inputs and outputs must be deleted. Right click on each of the names of the data input and data output and select Delete to delete them. The composite function should now look like Figure 2.5 below.

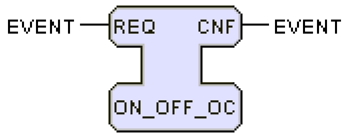


Figure 2.5 Composite Function Block with Input and Output Events

For this ON_OFF_OC function block, there needs to be two data inputs and one data output. Creating data inputs and data outputs are done by simply right clicking within the composite function block and click on New. Under New, there are two options: Events and Variable. Selecting Events allows new input and output events to be made; where as Variable is for creating new data inputs and data outputs. Click on Variable and click on Input to create a new data input. A window will pop up to define the name and data type for this input. For this tutorial, name this new data input IN1O-this will represent the open contact from Figure 2.1a. Click on Type and a pull down menu will appear to make the input a particular type. In this instance, both data inputs should be Boolean, so select BOOL to make IN1O a Boolean data input. Click OK when finished and the new data input should appear on the composite function block. Repeat this process for a data output named IN2C. This data input will represent the closed contact from Figure 2.1a to complete the necessary data inputs for the composite function block. When finished with the defining the data input, click on OK to see the updated results.

EXERCISE: Give the composite function block a single data output called OUT. Remember, the inputs for the composite function block and the basic function blocks that will be used are Boolean, so the output should also be Boolean.

After finishing the exercise, the ON_OFF_OC composite function block should look like Figure 2.6.

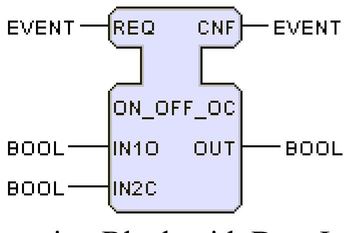


Figure 2.6 Composite Function Block with Data Inputs and an Data Output

You may have the same information but not organized the same way. To move events and data within a block, right click on the name of the event or data in the composite function block and select either Move Up or Move Down to move it up or down.

Step 4: Interface – Data Connections

While the composite function block ON_OFF_OC now has events and data inputs, the two are separate from each other. When an event is read into any function block, the data that it is associated with is read before it executes a particular algorithm. Therefore, data connections are necessary. When events or data are deleted, the data connections are automatically deleted. However, they must be added separately after creating new events and data. Since, there is only one event going into the function block, both data inputs must be connected to that event. To do this, hold the Alt key and click on the word EVENT associated with the event you want to make a connection for. The cursor will change and by dragging it to the desired data input or output name (or the data type of the data input or output name), a connection between an event and data will be made. If the same procedure is done on the same connection, the connection will be deleted.

There is only one input event for this composite function block; thus both data inputs should be connected to this single input event. Hold down the Alt key and click on EVENT next to the REQ event. Drag the cursor onto the word BOOL next to IN1O and that data input will then be connected to the REQ event. Whenever an event comes to REQ, IN1O will then be read and go through the composite function block. By using the same process, connect the REQ event to the IN2C data input and the CNF event to the OUT data output. When finished, the composite function block should look like Figure 2.7.

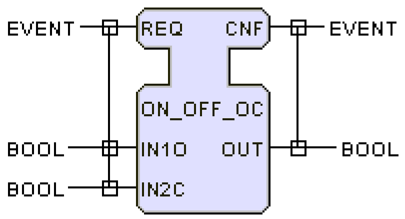


Figure 2.7 Composite Function Block with Data Connected to Events

Now that the data is connected to the events for the composite function block, all that needs to be done is to put basic function blocks in the composite function block for this function block to perform the desired operation.

Step 5: Filling the Composite Function Block

While there is input and output data, as well as input and output events, currently nothing is being done with the information coming to or leaving the composite function block. In order to have the composite function block to have any actual use, there must be function blocks (basic or other composite function blocks) within it. On the left side of the screen, the last entry in the list is the name of the block, ON_OFF_OC. Click on it and the workspace will show what is currently in the composite function block. Right now, only a function block named E_DELAY should be in the workspace. This workspace is exactly like the one that appears when one clicks on APP or a resource entry in a system; this is where function blocks, data connections, and event connections take place.

For this ON_OFF_OC function block, the E_DELAY function block is not necessary, so delete it. Add a FB_AND and a FB_NOT function block and then click on Update. If you click on Update after removing the E_DELAY function block but before adding any others, the listing on the side of the screen for this window disappears. This is done exactly as it was done under Step 6 in IEC Tutorial 1. Remember that both blocks can be found on the following directory path: fbdk->src->math. When finished, the screen should be like Figure 2.8.

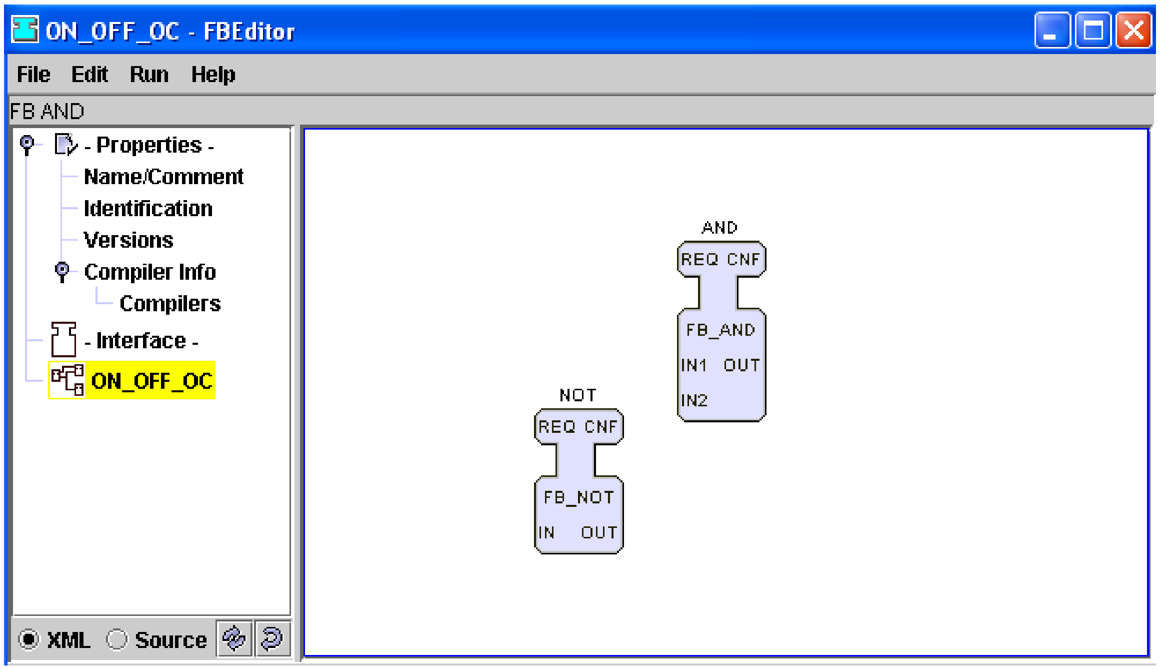


Figure 2.8 Basic Function Blocks for ON_OFF_OC Composite Function Block

EXERCISE: Connect these two blocks together with one data connection and one event connection. Be sure to follow the ladder logic shown in Figure 2.1a.

When finished with the exercise, the two blocks should be connected as shown in Figure 2.9.

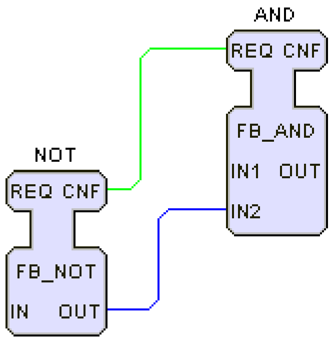


Figure 2.9 Basic Function Blocks with Connections for ON_OFF_OC

Step 6: Completing the Connections.

Now the necessary function blocks are in the ON_OFF_OC composite function block. The two are connected in the right order (NOT then AND function). However, there is still the issue with the data and events defined in the composite function block. Currently, the NOT and AND function blocks have no data connected to its inputs; the AND function block has no destination for its CNF event; and the NOT function block has no source for its REQ event. Further connections must be made so the data and events coming in and out of the composite function block will go through these function blocks inside of it.

Start with the events. The composite function block, ON_OFF_OC, has an event input called REQ to receive events and an event output CNF to output them. Since the FB_NOT function block should be accessed first, the REQ event from the composite function block should be connected with the REQ event of the function block named NOT. Right click on the REQ event on the NOT function block, click on Connect, and select REQ. By doing this, an event coming into REQ of the ON_OFF_OC composite function block will go directly to REQ of the FB_NOT function block within it. This connection is set. There needs to be a connection to the CNF event of the composite function block to output an event. Since the AND function block is the last function block to be used, the CNF event of that function block should be connected with to the CNF event of the composite function block. Right click on the CNF event on the AND function block, click on Connect, and select CNF. An event leaving through the CNF event of the AND function block will be the CNF event of ON_OFF_OC. The function block system within ON_OFF_OC should look like the system shown in Figure 2.10,

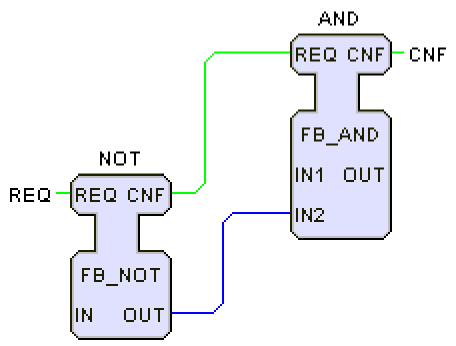


Figure 2.10 Events of Function Blocks Connected to the Events of ON_OFF_OC

The data inputted to and outputted from the ON_OFF_OC composite function block must also be connected to the basic function blocks. To start, connect IN2C of the ON_OFF_OC function block to the FB_NOT function block. IN2C represents a closed contact, so it needs to go through the IN data input of the NOT function block. Right click on the IN data input name, click on Connect, and select IN2C. The IN2C data input of ON_OFF_OC is now connected to the IN data input of NOT.

EXERCISE: Connect the IN1O data input of ON_OFF_OC to the IN1 data input of the AND function block and the OUT data output of the AND function block to the OUT data output of the ON_OFF_OC function block.

Once finished with that exercise, Figure 2.11 shows what should be within the composite function block ON_OFF_OC.

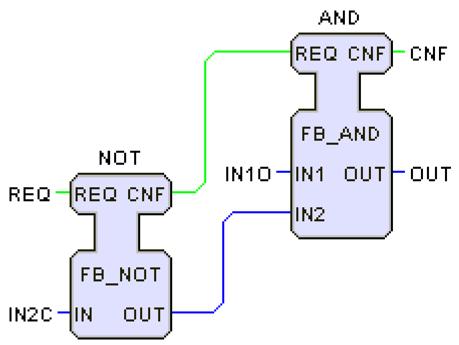
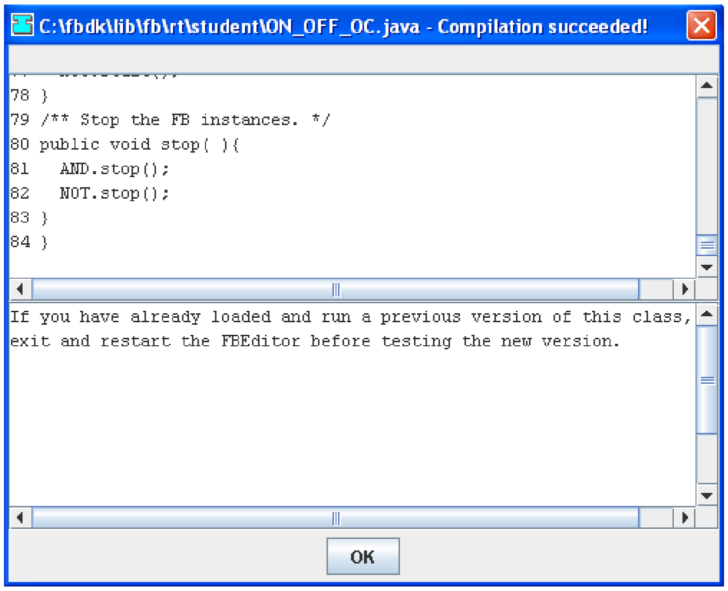


Figure 2.11 Linking Basic Function Block Data to Composite Function Block Data

Step 7: Compiling

The events, data, and logic of the ON_OFF_OC composite function block are now complete. However, ON_OFF_OC can not be used yet. The final step is to compile this function block so it can be used in other programs. Earlier, the compiler was defined as “package fb.rt.student;”. This is where the Java compiler within the program will check for this file and compile it. First, save this ON_OFF_OC composite function block as a Java file. This can be done by going to the menu bar and clicking on the following path: File -> Save As -> Java. Make absolutely certain the file is saved in the already defined directory. If, for some reason, the program does not automatically take you there, it can be found by this pathway: fbdk -> lib -> fb -> rt -> student. The name of the function block will be saved with the extension .java. The file will automatically be compiled. A window will appear, shown in Figure 2.12, if it is successful.


C:\fbdk\lib\fb\rt\student\ON_OFF_OC.java - Compilation succeeded!
OK

```
78 }
79 /** Stop the FB instances. */
80 public void stop( ){
81   AND.stop();
82   NOT.stop();
83 }
84 }
```

```
If you have already loaded and run a previous version of this class,
exit and restart the FBEditor before testing the new version.
```

Figure 2.12 Java Successfully Compiled

As it says in the window for Figure 2.12, before using this composite function block, the program used for making this composite function block must be closed. Save the ON_OFF_OC composite function block as an XML so it has the proper .fbt extension. It is suggested that you save this function block in a new directory under the src folder. Now it can be used like any other function block in any other system. For example, establish a directory with your name on it where you will save your function blocks. After following this tutorial, you have successfully created a composite function block.

EXERCISE: Implement this ON_OFF_OC composite function block in the following system in Figure 2.13. In this example we have given the name “Control” to this instance of the composite function block of type ON_OFF_OC.

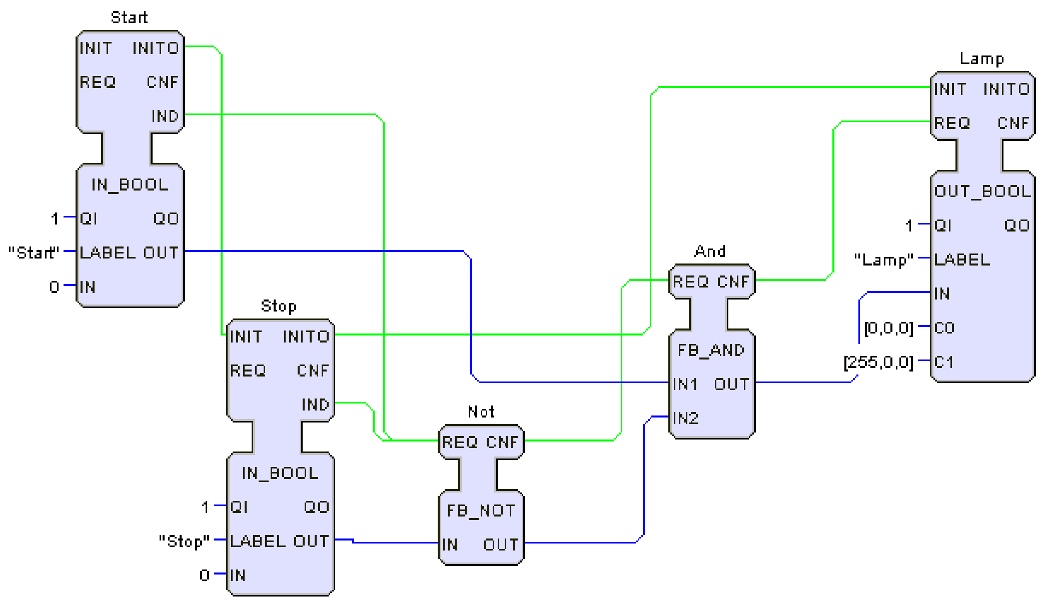


Figure 2.13 Example Using Basic Function Blocks

SOLUTION: The proper implementation of this function block system is in Figure 2.14.

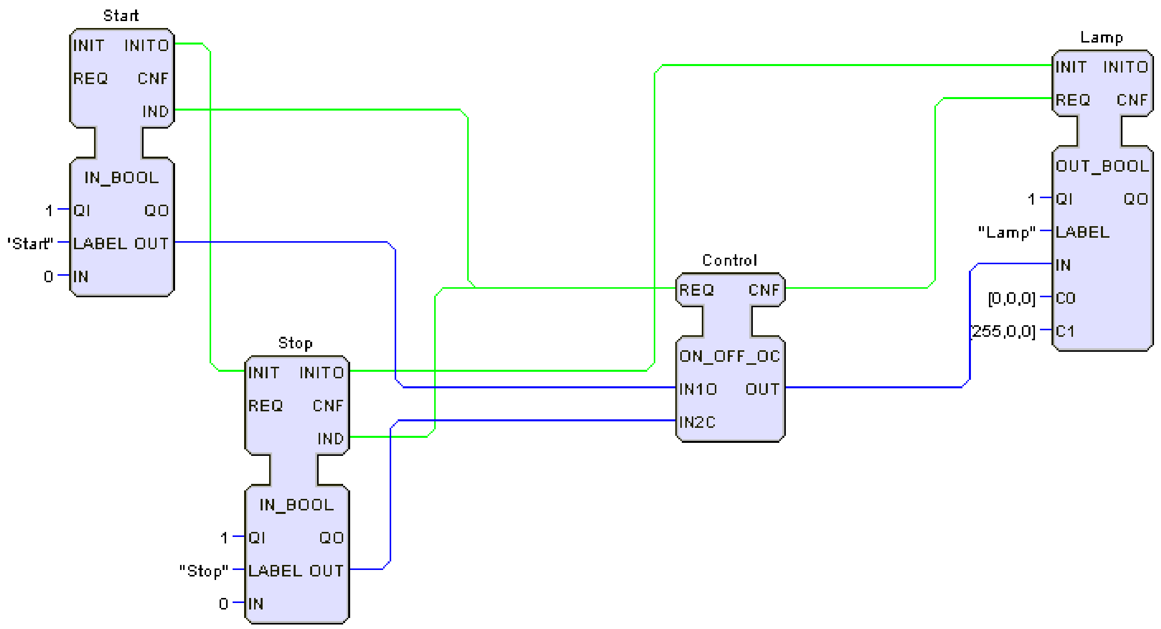


Figure 2.14 Example Using ON_OFF_OC Composite Function Block

2.3 Summary

This tutorial has shown the following sequential steps for creating and using a composite function block. To create a composite function block:

1. Define the properties of the composite function block.
2. Define the compiler information, where the compiler will look for the file, and the file path of other function blocks used in the composite function block.
3. Define the interface of the composite function blocks: events, data inputs, and data outputs.
4. Connect data inputs and outputs with corresponding events.
5. Insert all of the function blocks the composite function block will use and make any data and event connections within those function blocks.
6. Connect events defined in the interface to the proper function blocks inside the composite function block.
7. Connect data inputs and outputs defined in the interface to the proper function blocks inside the composite function block.
8. Save the function block as a Java file in the same directory defined for the compiler. The Java file will automatically be compiled.
9. If the file was successfully compiled, save the function block as an XML file.
10. Close the editing of the composite function block and add it to a system like any other function block.

IEC TUTORIAL 3: USER-DEFINED BASIC FUNCTION BLOCK

3.1 Introduction

In the IEC 61499 standard, there are two main types of function blocks: basic and composite function blocks. In addition to receiving and sending events and data, basic function blocks have algorithms to process the data received and execution control charts to control which algorithms are used depending on the event received. Composite function blocks are made up of combinations of basic function blocks.

There are several basic function block types that are part of the IEC 61499 standard, such as logical and mathematical function blocks. In addition, users can create their own basic function blocks for particular functions. To do so, it is necessary to define the execution control chart and the algorithms in addition to the events and data it will use. In this exercise we will create a user-defined basic function block.

SITUATION: Create a basic function block called ON_OFF_OC_B that implements the following ladder logic shown in Figure 3.1.

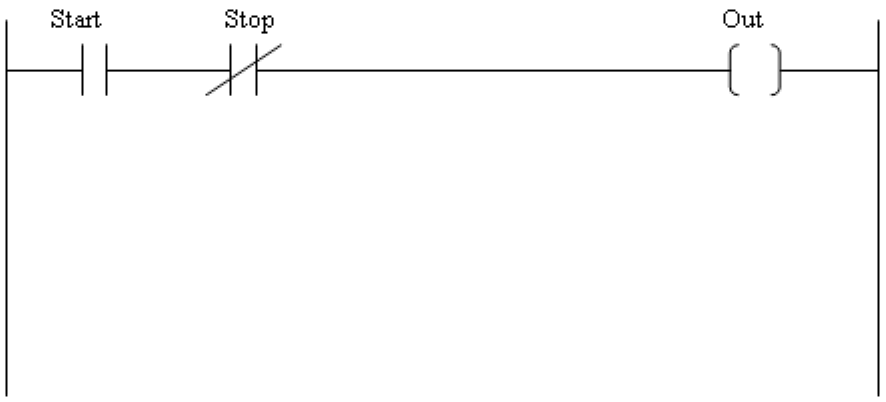
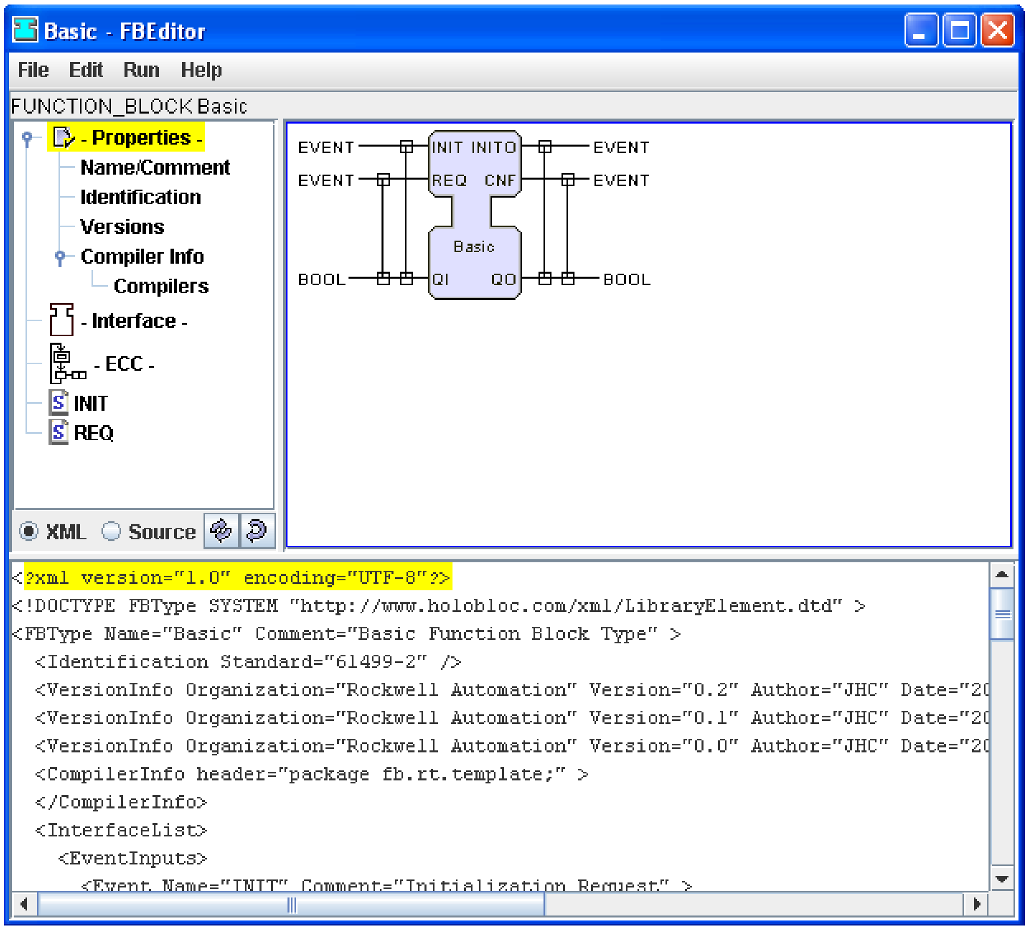


Figure 3.1 Ladder Logic for Basic Function Block

3.2 Steps in Creating a Basic Function Block

Step 1: Creating a New Basic Function Block

Go to the menu bar and click on File -> New -> FB Type -> Basic FB to create a new Basic Function Block. The basic function block NEWBASICFB will appear, already with input data, output data, input events, and output events. The new window should look like Figure 3.2.

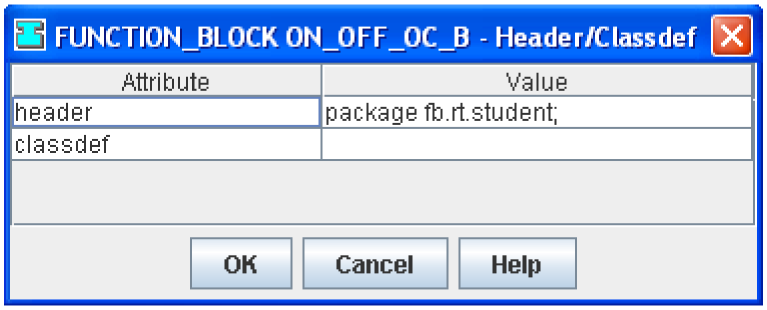


```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE FBType SYSTEM "http://www.holobloc.com/xml/LibraryElement.dtd" >
<FBType Name="Basic" Comment="Basic Function Block Type" >
  <Identification Standard="61499-2" />
  <VersionInfo Organization="Rockwell Automation" Version="0.2" Author="JHC" Date="2(
  <VersionInfo Organization="Rockwell Automation" Version="0.1" Author="JHC" Date="2(
  <VersionInfo Organization="Rockwell Automation" Version="0.0" Author="JHC" Date="2(
  <CompilerInfo header="package fb.rt.template;" >
  </CompilerInfo>
  <InterfaceList>
    <EventInputs>
      <Event Name="INIT" Comment="Initialization Request" >
```

Figure 3.2 Default Basic Function Block

Step 2: Setting the Properties

On the left side of the screen, there is a list of entries related to this new basic function block. The first entry is called Properties, with five entries below it. Notice that the first three entries under Properties are exactly the same for the properties of a system: Name/Comment, Identification, and Versions. For now, give this basic function block the name ON_OFF_OC_B (Note: This extra _B is needed if you followed the composite function block tutorial first and you do not want to overwrite that function block) and input the version information. Click OK when finished.



Attribute	Value
header	package fb.rt.student;
classdef	

Figure 3.3 Properties – Compiler Info Window

The remaining two entries are called Compiler Info and Compilers that are extremely important. This basic function block can not be used unless it is ultimately compiled. Therefore, like with a composite function block, a compiler must be defined. This information must be typed into the header field. The compiler is defined as (and without the quotes) "package fb.rt.student;". The field should be like Figure 3.3. No other directory paths need to be imported since a basic function block does not use other function blocks. Click OK to update the compiler information. Again, if the compiler information is not updated, modify the header in the XML file at the bottom of the screen and click on the circle of arrows to parse the code (or hit Alt+S). Double click on the Compilers entry and right click on the empty space and select New. This will add a compiler for the function block; click OK to finish adding the compiler.

Step 3: Interface – Events and Data

The next entry below Properties on the left side of the screen is called Interface. Click on that and the workspace will show the basic function block. This window is where event and data inputs and outputs can be created, deleted, and modified. The default new basic function block already has two input events: an INIT event for initialization and a REQ event for requesting of information. In general, there should always be an INIT event to make sure the function block does not use data that has been stored internally from previous running of the program. However, this function block only has data coming in from other blocks; an INIT event is not necessary. This basic function block will be receiving information, so there does not need to be any change to REQ. The basic function block has two output events: an INITO event for an output of an initialization and a CNF event for confirmation. Since there is no INIT event, there should be no INITO event. Data is being requested, so there needs to be a confirmation event, CNF, that the data is received.

However, there are only one data input and one data output currently for this function block. QI is the input and QO is the output. Generally, they are important for some function blocks to actually function in a program; but in this case they are not necessary. Right click on the QI data input and select Delete to delete the data input, and do the same for the QO data output. This will delete both the data as well as the data connection between the event and the data. It is not possible to delete any events that are in the execution control chart. This will be explained later. The basic function block should now look like Figure 3.4.

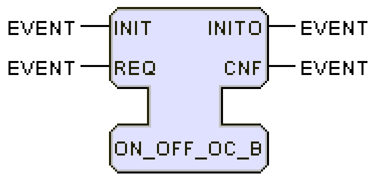


Figure 3.4 Basic Function Block with Input and Output Events

For the ON_OFF_OC_B function block, there needs to be two data inputs and one data output. Creating data inputs and data outputs is done by right clicking on the function block, clicking on New, and clicking on Variable. Select input to create a new data input. A window will pop up to name and define the data type for the new data input. For this tutorial, name it IN1; which will represent the open contact shown in Figure 3.1. Click on drop down menu next to the Name text field to select a data type for the data input. In this tutorial, both data inputs should be Boolean, so select BOOL to make IN1 a Boolean data input. Click on OK and the new data input IN1 will appear on the function block. Repeat this process for IN2, which will represent the closed contact shown in Figure 3.1. Now, the necessary data inputs for the basic function block are completed.

EXERCISE: Give the basic function block a single data output. Remember, the inputs for the basic function block and are Boolean, so the output should also be Boolean.

After finishing the exercise, the ON_OFF_OC_B function block should look like Figure 3.5.

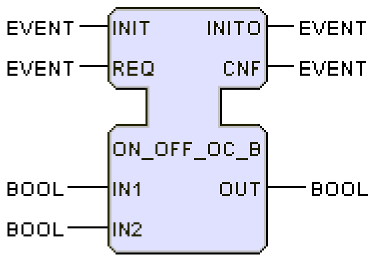


Figure 3.5 Basic Function Block with Input and Output Data

You may have the same information but not organized in the same way. To move events and data within a function block, right click on the name of the event or data and select either Move Up or Move Down to move it up or down.

Step 4: Interface – Data Connections

While the basic function block ON_OFF_OC_B now has events and data inputs, the two are separate from each other. When an event is read into any function block, the data that it is associated with is read before it follows a particular algorithm. Therefore, data connections are necessary. When events or data are deleted, the data connections are automatically deleted. However, they must be added separately after creating new events and data. Since, there is only one event going into the function block, both data inputs must be connected to that event. To do this, hold the Alt key and click on the word EVENT associated with the event you want to make a connection for. The cursor will change and by dragging it to the desired data input or output name (or the data type of the data input or output name), a connection between an event and data will be made. If the same procedure is done on the same connection, the connection will be deleted.

Hold the Alt key and click on EVENT next to the REQ event. The cursor will change and drag it over to IN1 or the BOOL next to IN1. The data input IN1 is now connected to the REQ event. Whenever an event comes to REQ, IN1 will then be read and go through the basic function block. By using the same method, connect the REQ event to the IN2 data input and the CNF event to the OUT data output. Do not make any data connections with the INIT or INITO events. When finished, the basic function block should look like Figure 3.7.

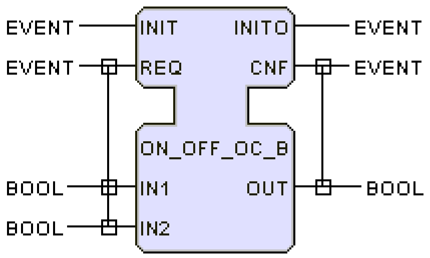


Figure 3.7 Function Block Data Connected to Events

Now that the data is connected to the events for the basic function block, it is necessary to set up an execution control chart for the events.

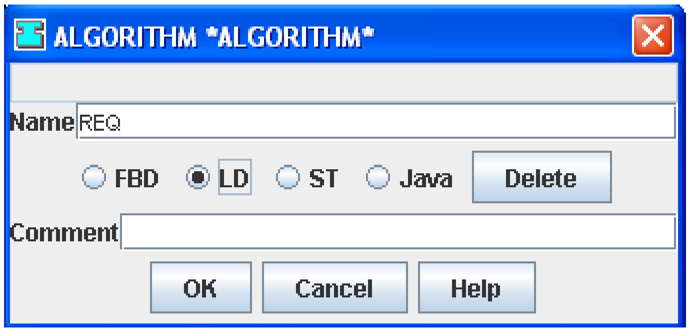
Step 5: Execution Control Charts

When events go into a basic function block, there is an execution control chart to determine what happens next, what algorithms are run, and what events should be outputted. For example, if event A is received, the chart will show that event A changes the state of the function block from its initial state (usually called START) to a new state that causes a certain algorithm to be activated. The execution control chart displays all this and can be edited to allow for different states, events, and pathways.

In Step 3, it was determined that the INIT and INITO events were not necessary. When Update was clicked on that point, an error appeared. That is because the default execution control chart still had a state, two transitions, and an algorithm assigned to INIT. Click on the entry named ECC on the left side of the screen and the workspace will display the default execution control chart. In order to delete the INIT and INITO events, the state, transitions, and algorithm assigned to INIT must be deleted. Right click on the box named INIT, which represents the state INIT. Click on Delete and click OK to delete the state, transitions with the state, and any algorithms associated with it. You can now go back to Interface and delete the INIT and INITO events. Right click on the box named REQ, select Delete, and click OK in order to make an execution control chart from scratch. There should now only be a single box displayed that is named START in the workspace.

The events for this basic function block ON_OFF_OC_B are REQ and CNF. Therefore, there should be a state REQ. Right click in the empty workspace, click New, and select State. A window will pop up to define the new state. Name the state REQ and give a comment to know what REQ stands for, such as Request State; click OK when you are finished. To make a state the initial state for the function block, click on the check box marked Initial State. However, the START state already in the execution control chart is the initial state for the basic function block, so there is no need to assign a new initial state. Now this new REQ state, there needs to be a transition to that state. A transition provides the means that has the function block change from one state to another. Creating a transition requires holding the Alt key, clicking on the source state of the transition, and dragging the new cursor to the destination state. Do so now with START as the source state and REQ as the destination state. Transitions not only require a source and a destination, but also a condition. This condition is the arrival of the event itself, whenever this event arrives, this transition among states will be made. The arrival of the REQ event will cause a transition between these states; thus, right click on the transition itself, click on Edit, and a window will pop up allowing the transition condition to be changed. By default, the condition event is 1; click on the drop down menu under Event and select REQ as the event condition. Click OK when finished assigning the new event condition. This means whenever the event arrives at REQ, the function block will go from the START state (assuming it is in that state to begin with) to the REQ state. Eventually, the basic function block should return to the initial state to avoid remaining the REQ state when there is no REQ event. Create a transition to return the function block to its initial state. Hold the Alt key, click on the source state REQ, and drag the cursor to the destination state START. After the REQ state, the basic function block will always return to the initial state. Therefore, the condition for this transition should be 1. The default condition for a transition is 1, so there is no need to edit it. That means after the algorithm is run, the function block will always return to the START state from the REQ state.

Now an algorithm must be defined before assigning algorithms to event. There are already two algorithms pre-defined for the default basic function block, INIT and REQ. Double click on each and select Delete to remove these algorithms. Click OK on the Algorithm window and then click on the circle of arrows below the list on the left side of the main screen to update the basic function block; the two algorithms in the list will then be deleted. Now, to create a new algorithm named REQ, which will be used for the REQ state. Right click on the empty workspace, click on New, and select Algorithm. A window will pop up to define a new algorithm. Name it REQ in the text field named Name. Underneath Name, there is a list of four languages for this algorithm. The four languages that can be used for algorithms in basic function blocks are structure text (ST), function block diagrams (FBD), ladder logic (LD), and Java. Since Figure 3.1 has is a ladder logic diagram, we will use ladder logic (LD) to define the REQ algorithm. When finished, the screen should look like Figure 3.8.


ALGORITHM *ALGORITHM*

Name: REQ

FBD [✓ LD] ST Java

Comment:

Figure 3.8 New Algorithm Definition Window

Now that the algorithm name and language for the algorithm have been defined, the REQ state should have its algorithm assigned to run whenever the function block reaches the REQ state. The output event of the function block should also be fired at this state. To do this, an action needs to be assigned to the REQ state. An action assigns an algorithm to be run and/or an event to be outputted whenever the function block reaches a particular state. Right click on the REQ state, click on New, and select Action. A window will pop up to define an algorithm and event for the action. Select CNF as the event and REQ as the algorithm, click OK when finished. This means once the basic function block is in the REQ state, the REQ algorithm will execute, and the CNF event will be launched in the end. We do not need an action for the START state, since it is acting as an initial state and therefore does not need an algorithm or an event to output. If it is hard to see the transitions and states, simply click and drag on the states and transitions to move them. The execution control chart for ON_OFF_OC_B should look like Figure 3.9 below.

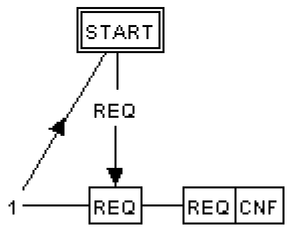
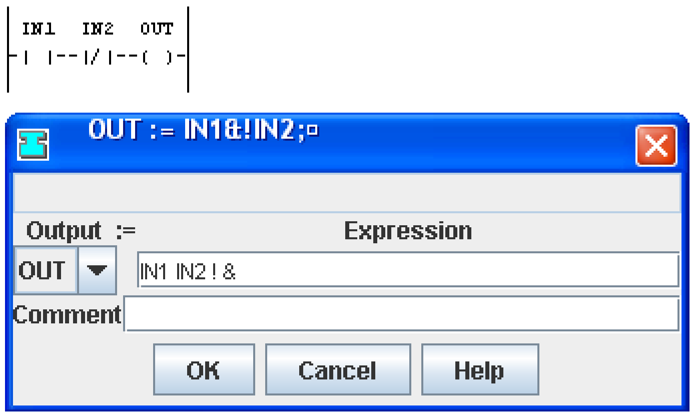


Figure 3.9 Execution Control Chart of ON_OFF_OC_B Function Block

Step 6: The REQ Algorithm

Click on the REQ at the bottom of the list on the left side of the screen and the workspace will become blank. Since the REQ algorithm was originally defined to use ladder logic, the workspace will be used to insert, modify, and delete the rungs for the ladder logic. This way, the ladder logic can be implemented into the basic function block.

For this tutorial, the ladder logic shown in Figure 3.1 will be implemented in this algorithm. Right click on the empty workspace, click on New, and select Rung. A window will pop up to define the output of the rung and the expression of the rung. The output of the function block is OUT, so the output of this algorithm should also be OUT. For the expression, the ladder logic has an open switch called Start and a closed switch called Stop. The two switches are in succession, so both must be true in order to yield an output. This function block has two Boolean inputs, IN1 and IN2. IN1 should represent the open switch called Start and IN2 should represent the closed switch called Stop. Closed switches (and other NOT functions) can be coded by putting an exclamation mark (!) after the variable. In this case, the ! is needed for IN2 since it is a closed contact. To include successive switches in ladder logic, an ampersand (&) is used after IN2. A space must be between any characters and the variable, otherwise the algorithm will not work properly. The expression, with the output, is shown in Figure 3.10 below.



OUT := IN1&!IN2;

OUT

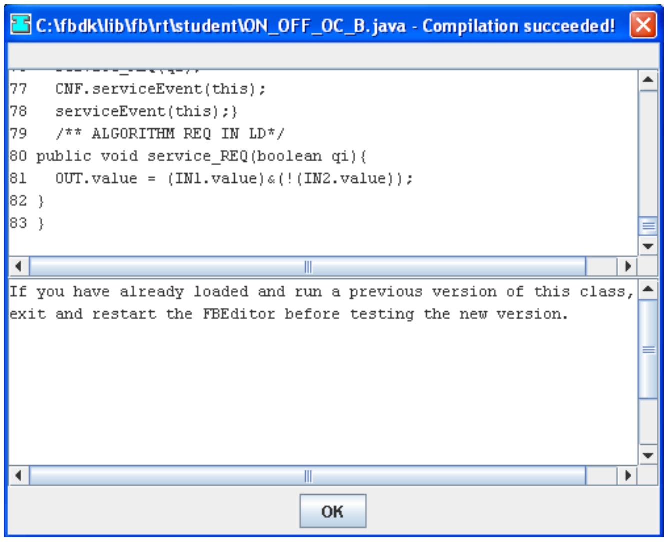
IN1 IN2 ! &

Figure 3.10 Ladder Logic Expression, Output, and Rung for Basic Function Block

The basic function block ON_OFF_OC_B now has the algorithm that follows the ladder logic required, an execution control chart that will have the algorithm be used, and the necessary events and data.

Step 7: Compiling

While everything needed is now in the basic function block, ON_OFF_OC_B can not be used yet. The final step is to compile this function block so it can be used in other programs. Earlier, the compiler was defined as “package fb.rt.student;”. This is where the Java compiler with the program will check for this file and compile it. First, save this ON_OFF_OC_B basic function block as Java. This can be done by going to the menu bar and selecting File -> Save As -> Java. Make absolutely certain the file is saved in the already defined directory. If, for some reason, the program does not automatically take you there, it can be found by this pathway: fbdk -> lib -> fb -> rt -> student. The name of the function block will be saved with the extension .java and automatically compiled. A window will appear, shown in Figure 3.11, if the file was successfully compiled.



C:\fbdk\lib\fb\rt\student\ON_OFF_OC_B.java - Compilation succeeded!

```
  CNF.serviceEvent(this);
  serviceEvent(this);}
  /** ALGORITHM REQ IN LD*/
public void service_REQ(boolean qi){
  OUT.value = (IN1.value)&(!(IN2.value));
}
}
```

```
If you have already loaded and run a previous version of this class,
exit and restart the FBEditor before testing the new version.
```

Figure 3.11 Java Successfully Compiled

As it says in the window for Figure 3.11, before using this basic function block, the program used for making this basic function block must be closed. Save the ON_OFF_OC_B basic function block as an XML file so it has the proper .fbt extension. Make sure to save the XML file in the same directory as your other function blocks (suggestion: C:\fbdk\src\student). Now it can be used like any other function block in any other system. After following this tutorial, you have successfully created a basic function block.

EXERCISE: Implement the basic function block ON_OFF_OC_B within the system shown in Figure 3.12.

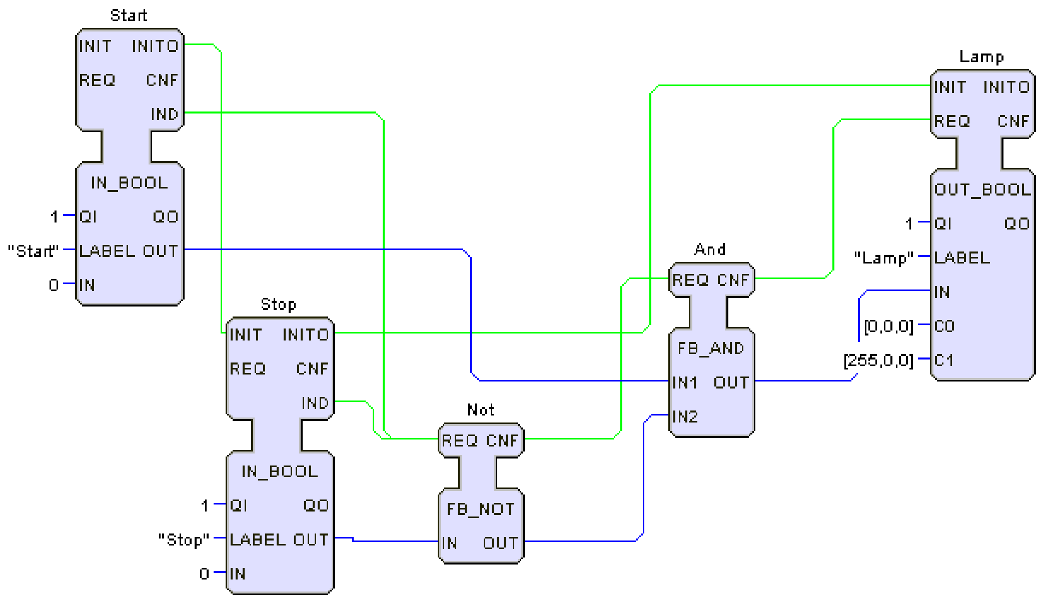


Figure 3.12 Example using Generic Basic Function Blocks

SOLUTION: The proper implementation of this function block is shown in Figure 3.13.

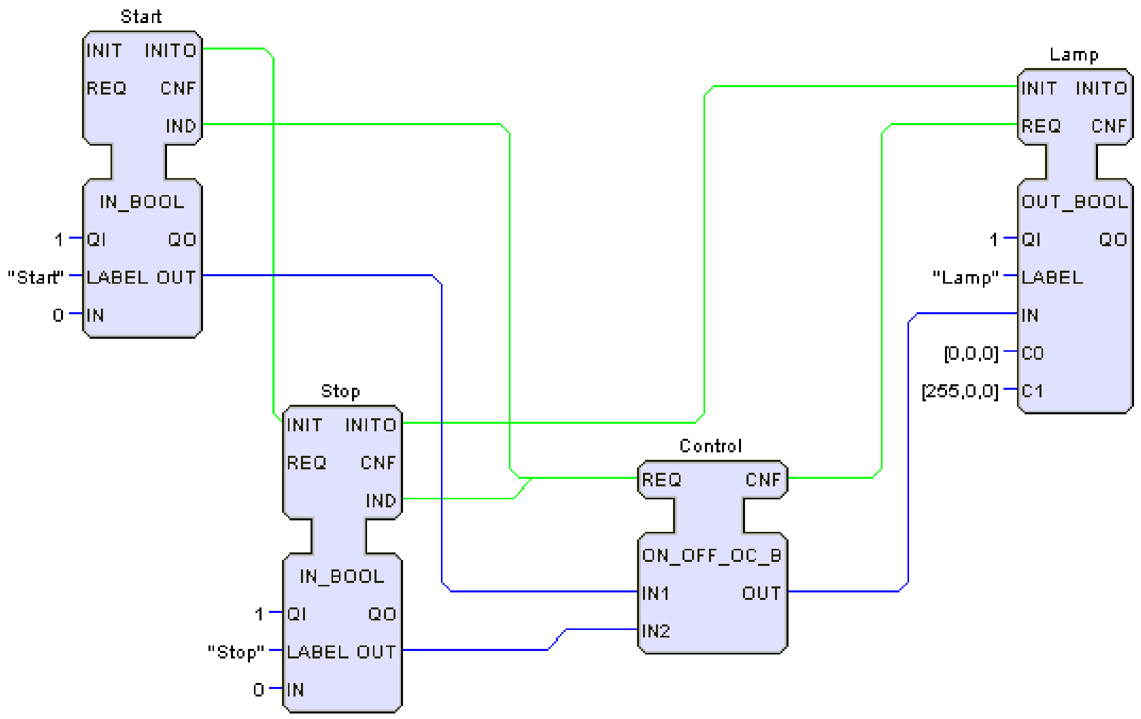


Figure 3.13 Example Using Created Basic Function Block, ON_OFF_OC_B

3.3 Summary

In summary, this tutorial has shown the following sequential steps for creating and using a basic function block. To create a basic function block:

1. Set the properties of the basic function block.
2. Define the compiler information for the basic function block.
3. Define the interface of the basic function blocks: events, data inputs, and data outputs.
4. Connect data inputs and outputs with corresponding events.
5. Modify the execution control chart to include the states of all events of the basic function block.
6. Define all the transitions between the states in the execution control chart.
7. Define the algorithms and their languages.
8. Assign the event outputs and algorithms to each state under Actions in the execution control chart.
9. Define the algorithm and its output.
10. Save the function block as a Java file in the same directory as the compiler. The Java file will be compiled automatically.
11. Save the basic function block as a XML file.
12. Close the basic function block editor and it to a system like any other function block.

IEC TUTORIAL 4: PUBLISH AND SUBSCRIBE FUNCTION BLOCKS

4.1 Introduction

The IEC 61499 standard is for distributed control. In tutorials 1 through 3 we have programmed function block applications but have not distributed those applications over more than one device. In this tutorial we show the use of multiple devices in a network. The IEC 61499 standard includes function blocks for enabling communication among devices in a network. The function block pair publish and subscribe are used when devices and their resources communicate with each other over a network, such as Ethernet, field bus, Allen Bradley data highway, or similar. The publish function block roughly corresponds to the message send (MSG) block in ladder logic. The message send transmits a word to an address in another node (device, resource) on the network. The subscribe function block enables a read at the recipient node of the contents of the address to which the message has been transmitted.

SITUATION: There is a semiautomatic drill press that has two controllers. The first controller (Device 1) runs a master panel with two buttons that control whether the system is on or off. One switch is System Start. The other switch is System Stop. The second controller (Device 2) runs the operating cycle of the drill press.

The two controllers must communicate with each other. Therefore, the data coming from the master panel needs to indicate whether the system has been started or not. Device 1 can do this by latching an ON indicator bit when the System Start switch is open and the System Stop switch is closed, and unlatching that bit when the System Start switch is closed and the System Stop switch is open. The status of that bit is then sent to Device 2. A diagram of the drill press and control panels is shown in Figure 4.1a. The ladder logic for the master panel is shown below in Figure 4.1b.

Semiautomatic Drill Press Operation:

1. System Start button (I1) turns the pilot light on.
2. A part to be drilled is placed under the spindle by the operator.
3. With a part in place, the operator must press both the left and the right start buttons simultaneously. The shield down solenoid (O2) is enabled. Start buttons must be held down until shield is down.
4. When the shield is down, the drill spindle starts to rotate and the drill head comes down. The downward force is from an air cylinder (O4).
5. When the drill head is completely down, LSDD sensor is activated (I9).
6. The activation of LSDD turns off the drill down solenoid, the shield down solenoid, and the drill motor. The spindle automatically returns to the up position and the shield is raised.
7. If at any time you hit a left or right stop button, the operating cycle should shut down.
8. If at any time you hit the master panel stop button, the system should shut down.

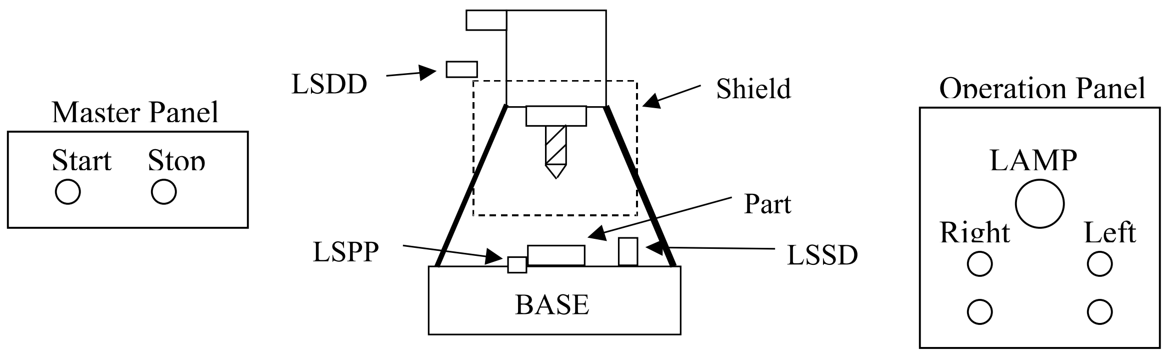
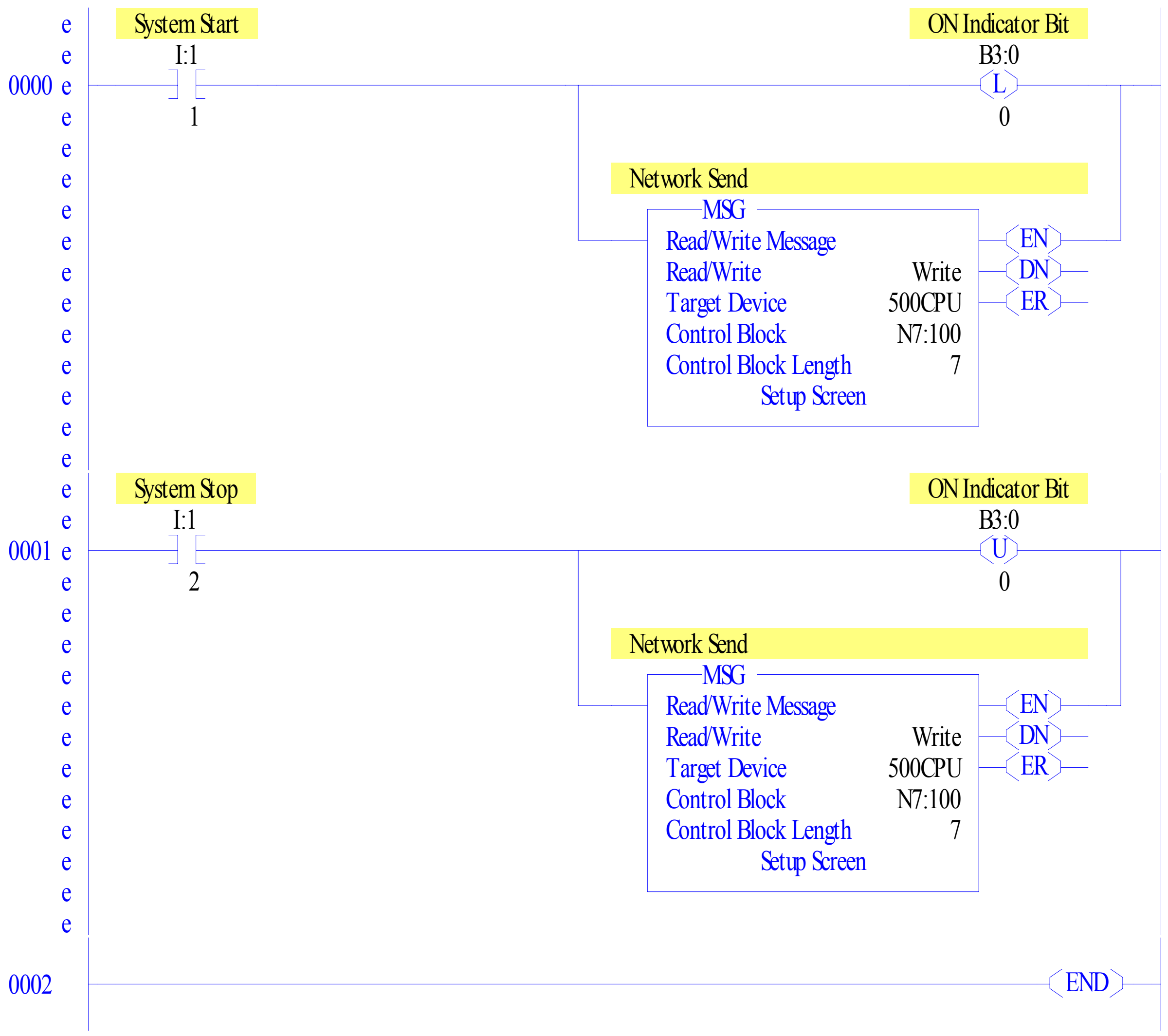
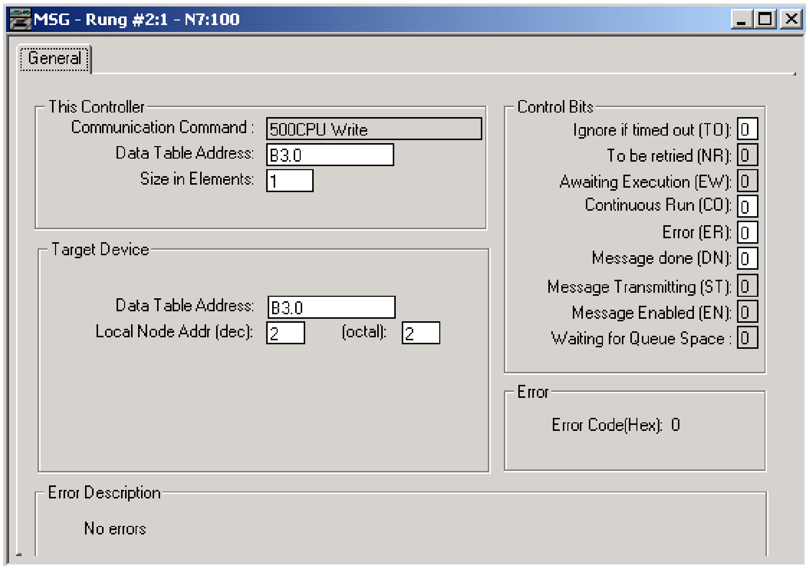


Figure 4.1a Drill Press Problem




MSG - Rung #2:1 - N7:100

General

This Controller
Communication Command : 500CPU Write
Data Table Address: B3:0
Size in Elements: 1

Target Device
Data Table Address: B3:0
Local Node Addr (dec): 2 (octal): 2

Control Bits
Ignore if timed out (TO): 0
To be retried (NR): 0
Awaiting Execution (EW): 0
Continuous Run (CO): 0
Error (ER): 0
Message done (DN): 0
Message Transmitting (ST): 0
Message Enabled (EN): 0
Waiting for Queue Space : 0

Error
Error Code(Hex): 0

Error Description
No errors

Figure 4.1b Master Control Panel Ladder Logic (Device 1)

The controller of the operating cycle (Device 2) must receive bit B3/0 and then use it in conjunction with the logic of two closed switches, Left Stop and Right Stop. Ultimately, it will control the pilot lamp. The ladder logic for the first rung of Device 2 is shown below in Figure 4.2.

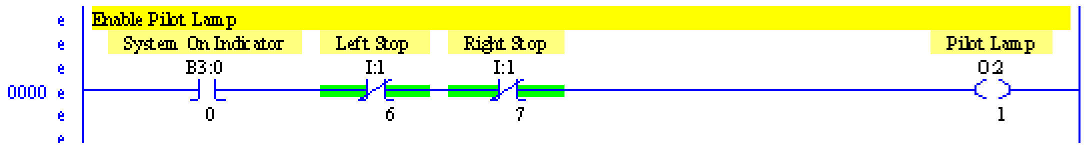


Figure 4.2 Ladder Logic of First Rung of Device 2

Create a system of function blocks that realizes the above logic. Use both publish and subscribe function blocks to control the communication of both controllers.

4.2 Steps in Implementing Publish and Subscribe Function Blocks

Step 1: Defining Multiple Devices.

Begin by creating a new system configuration and setting the properties. See Tutorial 1 for further details in doing so. Now, click on the entry below properties (the system name) to edit the overall system. As done previously in Tutorial 1, edit the first device to make it a FRAME_DEVICE and name it DEVICE_1. In prior tutorials, all of the necessary function blocks and logic were placed on one device having one resource. However, for this tutorial, two separate controllers that communicate with each other are needed. Therefore, there needs to be two devices with one resource each. One device will be for the master panel, the other for the operations panel. Right click on the empty workspace, click on New, and then click on Device. A window will pop up allowing the new device to be given a name and a device type. Give the new device the name

DEVICE_2, and set it as a FRAME_DEVICE. Two devices will appear, similar to Figure 4.3.

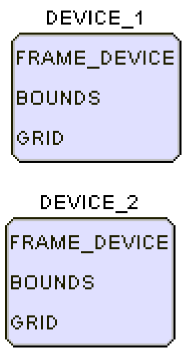


Figure 4.3 Establishing Two Devices in a System

As in Tutorial 1, these devices must have their boundaries defined. Just for the sake of this tutorial, it would be good to have both devices appear next to each other. For the first device, use [100,100,150,150] for the Bounds and [1,1] for the Grid. For the second device, use [250,100,150,150] for the Bounds and [1,1] for the Grid. Now when the program is launched later, two windows of the same size will appear right next to each other. Before moving on, make sure that each device has one resource, and make sure the resource type is defined as PANEL_RESOURCE.

Step 2: Setting Up For & Using Publish Function Blocks

The easiest way to create a system involving both PUBLISH and SUSCRIBE function blocks is to start by setting up what needs to be published. Go into APP and insert two IN_EVENT function blocks, call one System_Start and the other System_Stop. These function blocks can be found in the directory fbdk-> src -> hmi; the same directory as input and output Boolean function blocks. These two function blocks are the buttons on the master panel. Now, as shown in Figure 4.1b, the data needs to be latched. Insert an E_RS function block, which will act as a latch. Latches are event driven function blocks, all event driven function blocks can be found in the directory fbdk -> src -> events. Insert an OUT_BOOL function block to represent the ON indicator bit. Call it System_On, the reason for that name will be explained later. Now, to represent the logic of this part of the program, the System_Start function block sets the latch, whereas the System_Stop function block unsets the latch. To do that, connect the IND event of the System_Start function block to event S of the E_RS function block and connect the IND event of the System_Stop function block to event R of the E_RS function block. Connect the data output Q to the IN data input of the OUT_BOOL function block to have the OUT_BOOL function block work properly. Connect the event EO of the latch to the REQ event of the OUT_BOOL function block. The completed logic is shown in Figure 4.4 below.

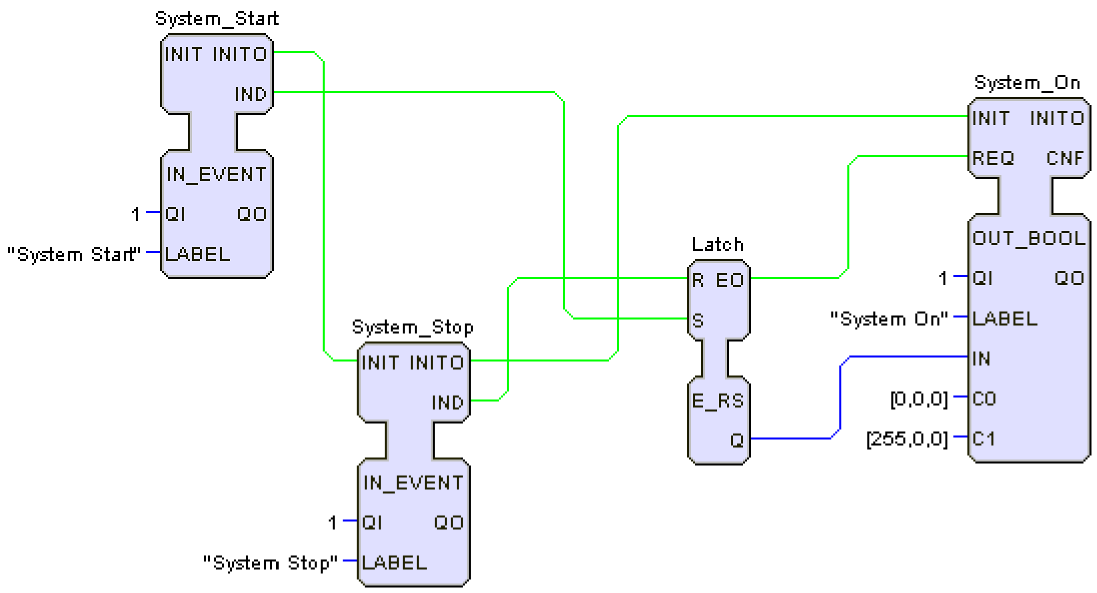


Figure 4.4 Completed Application program for Master Control Panel

Now the function block diagram for the master panel is nearly complete. This tutorial will use DEVICE_1 as the master panel, so map all of the blocks except for the System_On function block to the resource in DEVICE_1. Connect E_RESTART to the first function block mapped to DEVICE_1 (this will probably be System_Start) to ensure that blocks with INIT events are initialized. Now, the publish function block can be implemented. While the data outputted from the latch normally would go to an output function block, the second device-the operating panel-needs that data. As a result, this device will publish that data output. That is, this device will always take the data coming from the latch available to any subscribers-devices that read data from the publisher. Insert a new function block, the publishing (and subscribing) function blocks are found through the directory path: fbdk -> src -> net. Select PUBL_1, since only there is only one data input necessary (SD_1) and the network connection will be local. Name this function block Pub1.

Publish function blocks need an IP address and a port so data can be written to a specific address. It is important to make sure the IP address and port used is not used by any other devices, otherwise data and messages will get caught up and come in conflict with other data and messages being moved on that address and port. For this tutorial, the IP address 225.0.0.1 and the port 0001 is used to keep the data being written to and read from a local address. Make the necessary data and event connections; the completed master panel resource is shown in Figure 4.5.

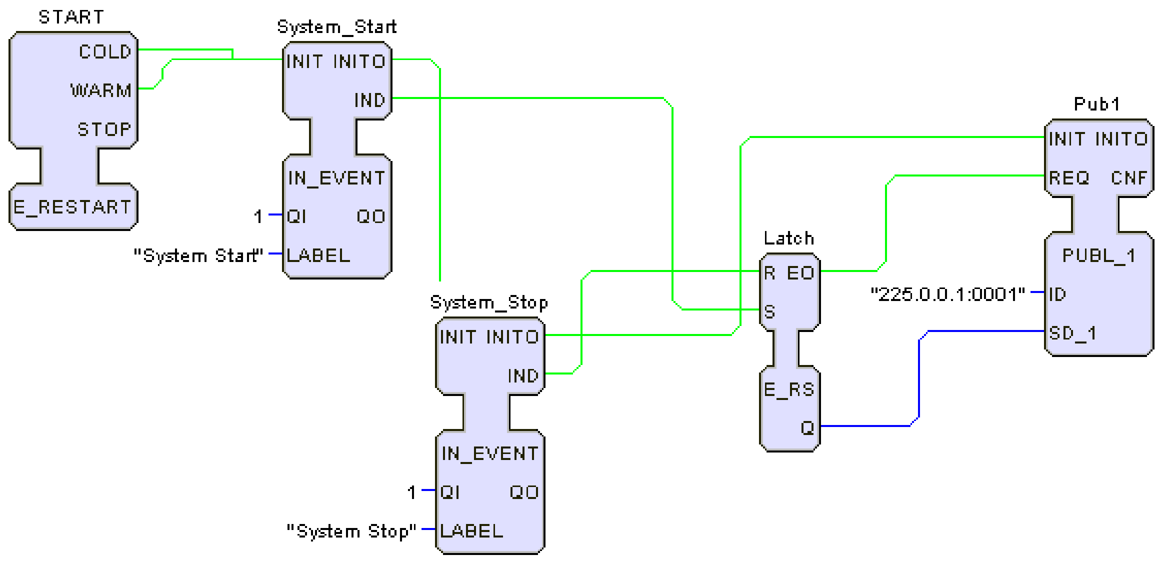


Figure 4.5 Master Panel Application Program with Network Communication

Step 3: Using A Subscribe Function Block

Go back to APP and map the System_On function block to the resource in DEVICE_2. The reason why this function block is going to the other device is to check whether the communication between the two resources is successful. The output function block is called System_On, since whenever the System_Start button is pressed, that output received from the master panel will be 1, representing that the system is on. If System_Stop is pressed, the output received from the master panel will be 0, representing that the system is off. In DEVICE_1, the resource is outputting this data from the latch to a PUBL_1 function block.

To receive data from a publish function block, there needs to be a subscribe function block. In the same directory where the publish function blocks are located, insert a SUBL_1 function block and name it Sub1. The publish function block has one data input being written; therefore, only one data output needs to be subscribed from the publisher. Exactly like the publish function block, the subscribe function block needs an IP address and a port to read data from. Make absolutely certain that the IP address and port defined for the Subscribe function block is exactly the same as the IP address and port defined for the Publish function block. Otherwise, the SUBL_1 function block will not receive the data from the PUBL_1 function block. Connect the data and events and the resource for the second device should look like Figure 4.6 below.

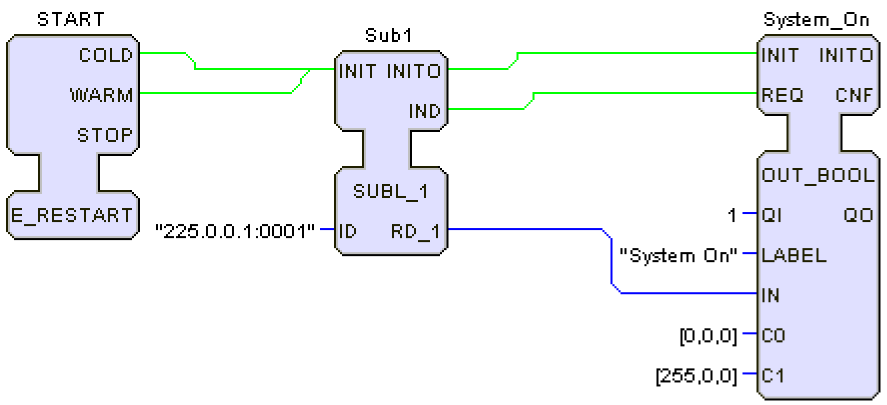
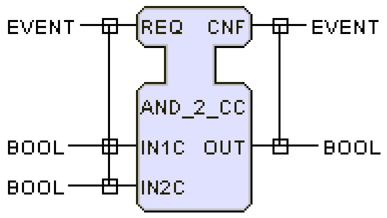
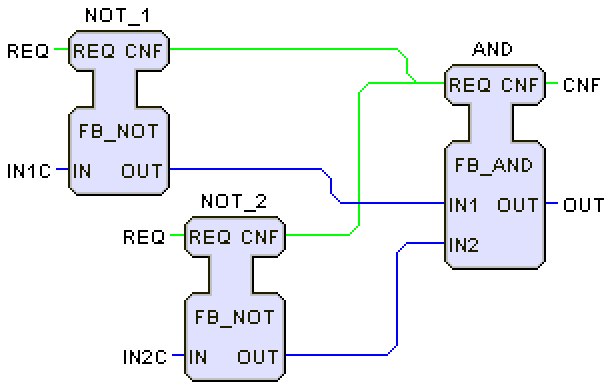


Figure 4.6 Operations Panel Resource with Network Communication

Step 4: Completing the System

Now, to complete the system, the data that comes from the SUBSCRIBE_1 function block acts as an open switch that comes in succession with two closed switches, Left Stop and Right Stop. Insert two IN_BOOL function blocks, called Left_Stop and Right_Stop, into the resource in DEVICE_2. If you have completed the Semiautomatic Drill Press example, you should have a composite function block that has the logic of two successive closed switches called AND_2_CC. If not, simply create a new composite function block with that name. The details on how to make a composite function block are in Tutorial 2. Make sure the composite function block has the following interface and system within it, shown in Figures 4.7a and 4.7b.





Left: Figure 4.7a, AND_2_CC Interface; Right: Figure 4.7b, AND_2_CC System

Save and compile the AND_2_CC function block first as a Java file and then as a XML file. If it is successful, insert the AND_2_CC function block and connect it with the two IN_BOOL function blocks. In this tutorial, CloseClose is the name for the AND_2_CC function block. The data that comes from the subscriber is in succession with the two closed switches, so the data output of AND_2_CC and the data from SUBSCRIBE_1 needs to be connected with a FB_AND function block. In this tutorial, this FB_AND function block is named And1. The output of that function block should be connected with the final output, the Pilot Lamp. Insert an OUT_BOOL function block for the Pilot

Lamp (named Pilot_Lamp) and make the final connections to complete the operations panel. The resource in DEVICE_2 should now look like Figure 4.8.

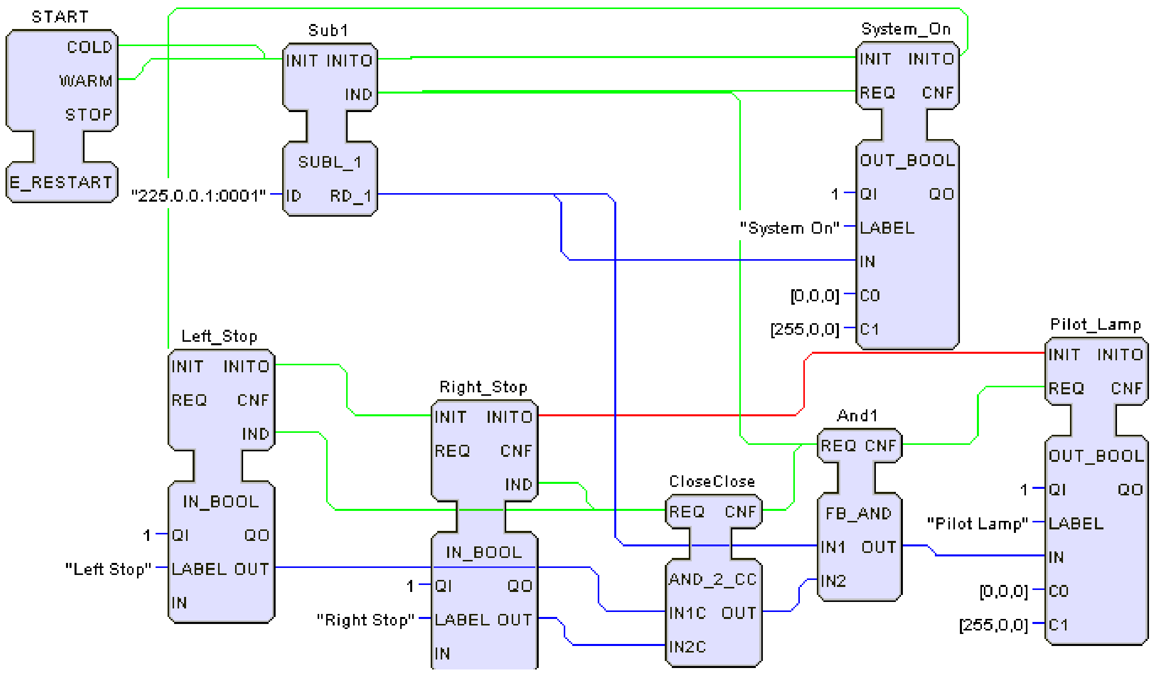


Figure 4.8 Completed Application Program for the Operations Panel Resource

Now that the resources in both devices are complete, launch the program. By pressing System Start, both lights will turn on. If you then select Left Stop and/or Right Stop, the Pilot Lamp will turn off but the System On should be on. If you press System Stop, both lights will turn off.

4.3 Summary

This tutorial has shown the following sequential steps for basic use of publish and subscribe function blocks. These steps are:

1. Create a new system.
2. Define as many devices and resources as necessary.
3. First program the function blocks that will yield the data that will be published.
4. Insert the necessary publish function blocks in resources to publish the data.
5. Define a proper IP address and port for each publish function block.
6. Insert the subscribing function blocks in other resources.
7. Define the exact same IP address and port as the publish function blocks for the subscriber function blocks to receive the data being published.
8. Using this data, program additional function blocks that may use this received data.

IEC TUTORIAL 5: CLIENT AND SERVER FUNCTION BLOCKS

5.1 Introduction

In addition to publish and subscribe function blocks, the IEC 61499 standard includes client and client function blocks. Similar to publish and subscribe function blocks, client and server function blocks are used in order for resources within devices to communicate with other resources in other devices on a network. What makes them different is how the communication works. The client function block communicates with the server function block, but not with other client function blocks. When a client function block requests information, the server function block responds by receiving and returning data to the address of the client function block.

SITUATION: There is a semiautomatic drill press that has two controllers. The first controller acts as a master panel, two switches that control whether the system is on or off. One switch is called System Start, when activated, the system will start. The other switch is called System Stop. Whenever System Stop is activated, the entire system should stop in spite of the state of other switches. There is a lamp on the master panel to indicate whether the system is on or not. The second controller acts as an operation panel, complete with any inputs and outputs that affect the operation cycle, but not controlling the entire system.

The two controllers must communicate with each other. Therefore, the data coming from the master panel needs to indicate whether the system is on or not. It can do this by switching on the open switch System Start, or by turning on the system's lamp (the lamp acts as an open switch). The ladder logic for the master panel is shown below in Figure 5.1.

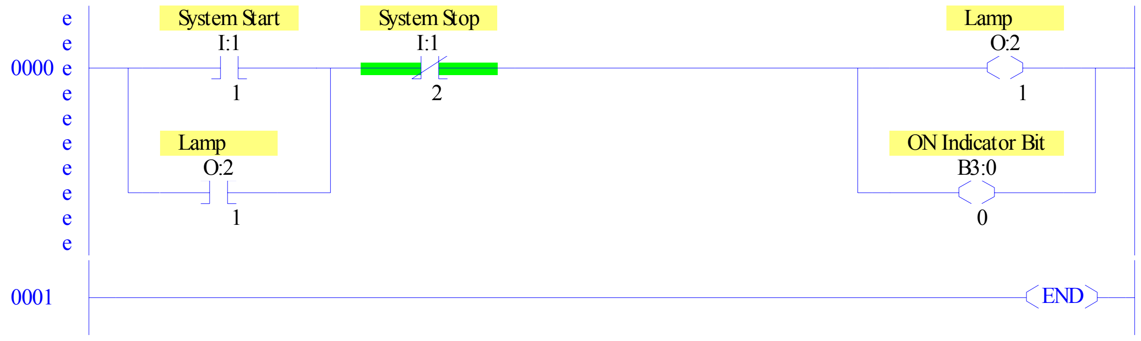


Figure 5.1 Master Panel Ladder Logic Diagram

The controller panel must receive the ON Indicator Bit and will control the Pilot Lamp in conjunction with two closed switches, Left Stop and Right Stop. Whether the Pilot Lamp is on or not will control whether the Safety Down Solenoid is on in conjunction with the open switches Left Start, Right Start, LSPP Part in Place, LSSD Drill Down, and LSSD Shield Down. The Safety Down Solenoid in conjunction with the LSSD Shield Down open switch will control the Spindle Motor, which will ultimately control the Drill Down Solenoid. The ladder logic for the controller is shown below in Figure 5.2. (Note:

This is the exact same ladder logic as seen in the Semiautomatic Drill Press example. It is suggested that you complete the example first in Tutorial 4, as this tutorial will only cover the usage of client and server function blocks.)

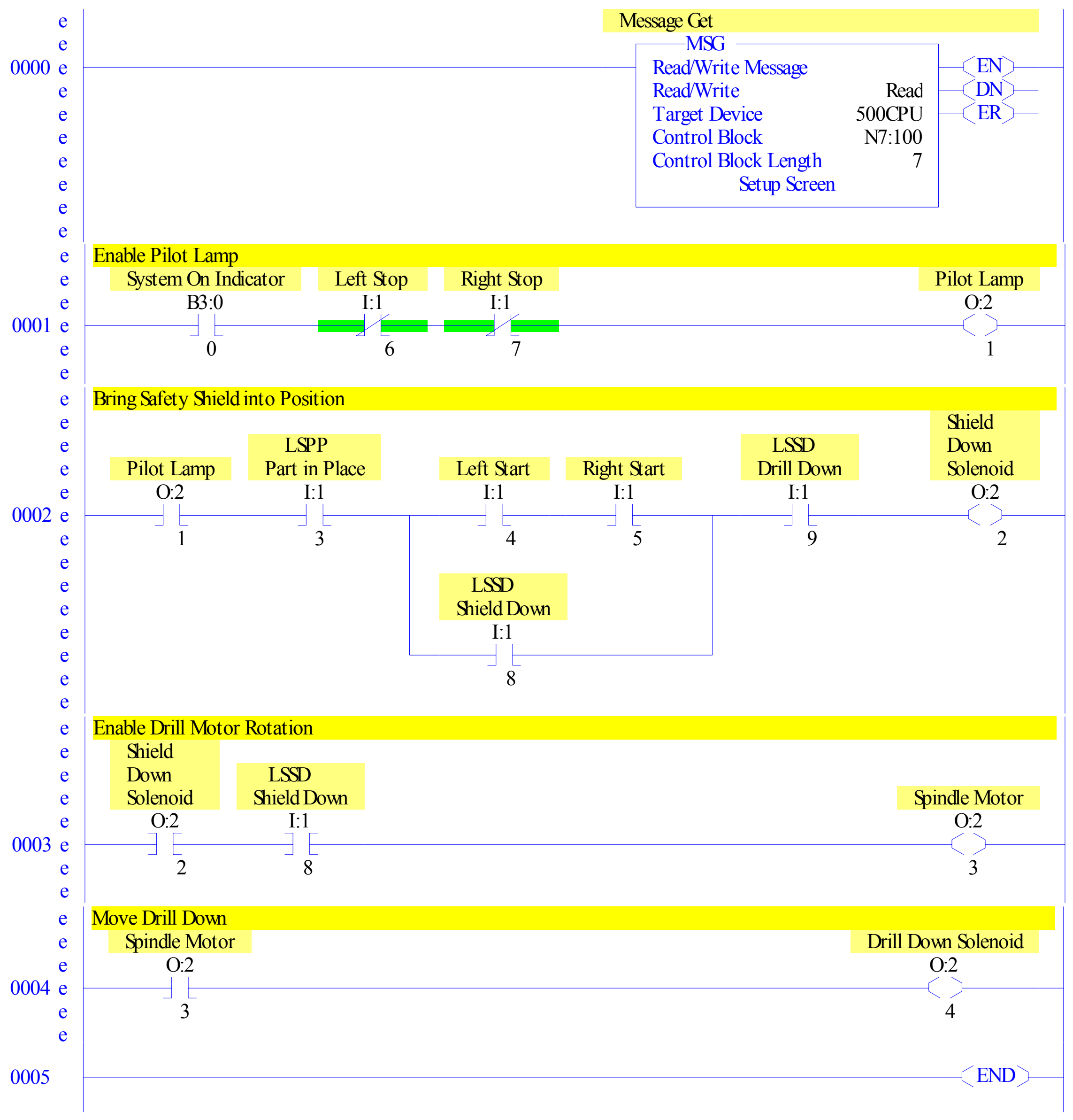


Figure 5.2 Ladder Logic of a Semiautomatic Drill Press

STEP 1: Define Devices and Resources

Create a new system and define their properties, as done in Tutorial 1. Create two devices and define them as FRAME_DEVICES. It may be useful to denote one device as the master panel and the other as the controller panel. For configuring the devices, DEV1 (the master panel) will have the bounds [100,100,150,150] and a grid of [1,1]. DEV2 (the controller panel) will have the bounds [250,100,150,295] and a grid of [1,1].

Once finished, make sure each device has one resource, both defined as PANEL_RESOURCE.

STEP 2: Setting Up & Using Client Function Blocks

When working with multiple resources, it is sometimes easier to just set up the application within the resources themselves as opposed to using APP1 and then mapping each function block to the proper resource. It is easier to check whether the system works properly, since launching the system only carries out what is in the resources and devices. More apt to this tutorial, it is easier to focus on using a client function block. As mentioned earlier, client function blocks only communicate with the server function block. Not just receiving information from the server, but also assigning information to the server. For this tutorial, the ON Indicator Bit is going to be communicated between the two devices, so the client and server function blocks will be passing that information between each other. On its own, a server function block does not have any information, which is why a client function block (or other function blocks in that resource) must assign data to it. Therefore, data from the master panel acts as the client and the controller panel acts as the server. It will not work if the roles are reversed.

Go into RES1 in DEV1 and insert function blocks to accomplish the ladder logic shown in Figure 5.1. Ideally, two IN_BOOL function blocks will be used for the switches; and a FB_AND function block, a FB_OR function block, and a FB_NOT function block will accomplish the logic of the system. Note that unlike Tutorial 4, the lamp is not being latched. The master panel, without the means for communication, is shown in Figure 5.3 below.

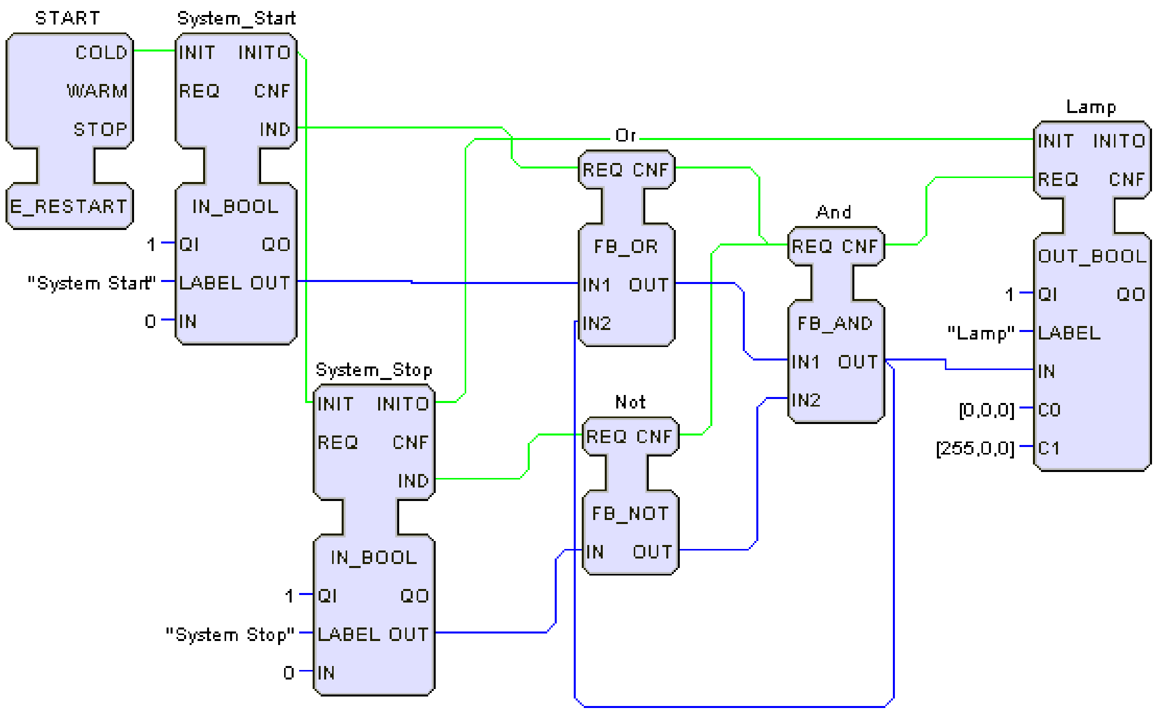


Figure 5.3 Application of Master Panel Resource

Launch the system and check to see if the system works; the light named lamp should turn on when System Start is checked on (you may need to check each input, sometimes it does not work right away). Now the client function block can be added to this resource. As with the publish and subscribe function blocks, the client function blocks can be found by following directory path: fbdk -> src -> net. For this tutorial, load up the CLIENT_2_1 function block, call it Client, and insert it into the resource for the master panel.

The CLIENT_2_1 function block, unlike a publish function block, contains both data inputs and data outputs. This particular function block allows for two data inputs, which will be read and sent the server at the specified address when there is a REQ event. Data on the corresponding server (either read into or stored on it) can be outputted through the single data output. There are multiple client function blocks with varying amounts of data inputs and outputs. However, for this tutorial, the master panel only needs to output the OUT Indicator Bit to the controller panel. Therefore, only a single data input of the Client function block will be used. Connect the OUT data output of the And function block to the SD_1 data input of the Client function block. Connect the CNF event of the FB_AND function block to the REQ event of the Client function block, and connect the INIT event of the Client function block from the INITO event of the Lamp function block. Once the event and data connections are finished, assign 1 to the QI of the Client function block. Right click on the Client function block to configure it and give it the ID "localhost:1499." An IP address is unnecessary since this will run on a solitary computer. 1499 is the port that will be used for transferring data. The master panel resource should look like Figure 5.4 below.

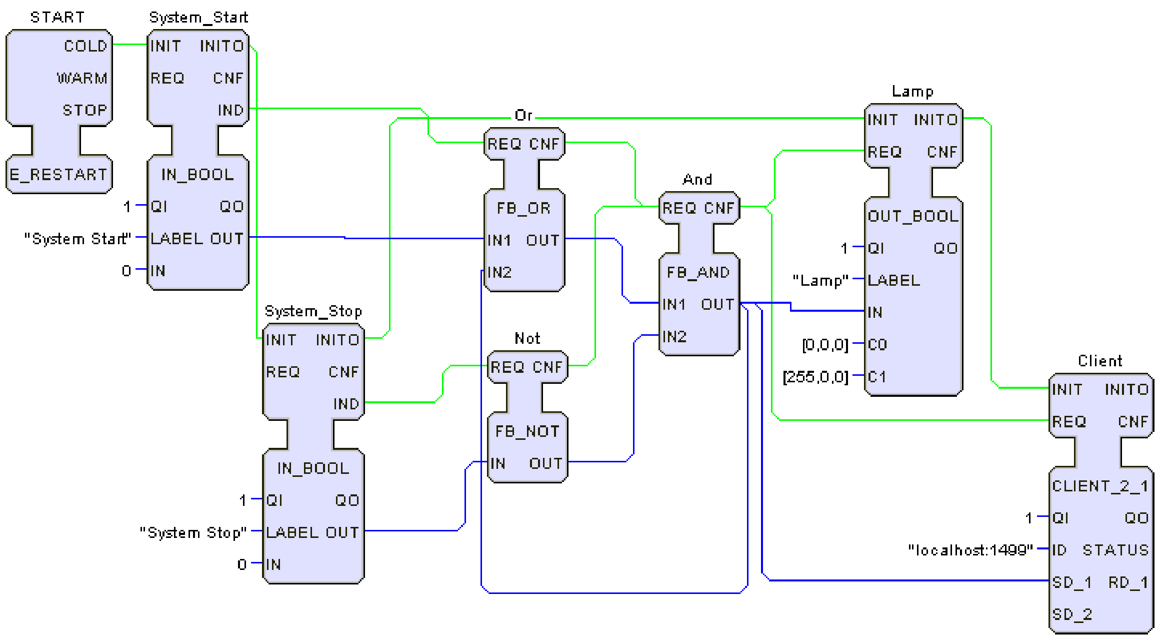


Figure 5.4 Master Panel Resource with Network Connection

STEP 3: Setting Up & Using Server Function Blocks

Now the master panel device is complete and the controller panel device must be set up. The master panel device used a CLIENT_2_1 function block, and since client function blocks only communicate with corresponding server function blocks, the controller panel must have a server function block. In the same directory as the client function blocks, load and insert a SERVER_1_2 function block and call it Server. Just like the CLIENT_2_1 function block, the SERVER_1_2 function block has both data inputs and data outputs. If there is data coming into data input of the Server function block, the Server function block will send that data to Client function blocks that request data. Data that comes to the server can be outputted through its data outputs. For this tutorial, there is no need to send any information to the Client function block as the CLIENT_2_1 function block is sending the ON Indicator Bit data. Therefore, only one data output needs to be used. Configure the SERVER_1_2 function block and give it the same ID as the CLIENT_2_1 function block. The ID for both the Client and the Server must be the same; otherwise there will be no communication between the two devices. Also make sure to assign 1 to the QI data input of the SERVER_1_2 function block and connect it with the E_RESTART function block in the resource. When finished, the controller panel resource should look like Figure 5.5.

To check whether the two devices will communicate properly, insert an OUT_BOOL function block into the resource in DEV2. If the communication works properly, both the Lamp and Check (the name for this OUT_BOOL function block) will turn on when System Start is pressed; both will turn off when System Stop is pressed. Configure the function block, assign QI the parameter of 1, and connect it with the SERVER_1_2 function block. Save the system and then go to the menu bar, click on Run, click on Launch, and click on the system name.

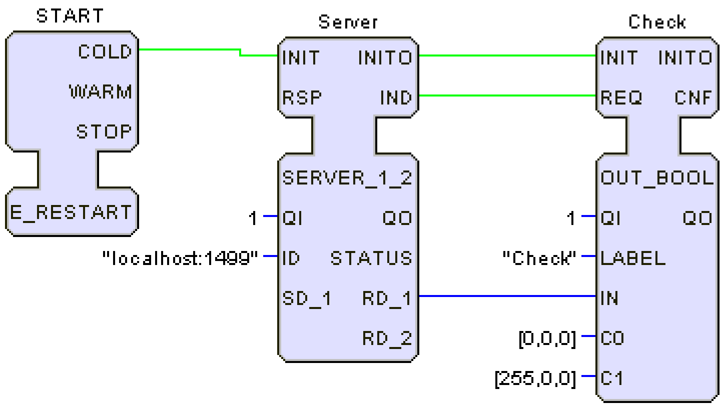


Figure 5.5 Controller Panel Resource with Network Communication

When the System Start check box is checked on, both the Lamp and the Check should both turn on (turn red, in this case). When System Stop is checked on, they should turn off (turn black, in this case). If this occurs, the communication between the two devices is working.

At this point, you can delete the connections and the OUT_BOOL function block used for testing the communication between the two devices. Now the rest of the controller panel's logic can be fulfilled. As mentioned earlier, this logic is the same as the Semiautomatic Drill Press example. If you have not done that example, it is advised that you go complete the example. If you have completed that example, simply copy what was done in that example into the second resource except for the System Start, System Stop, and Control1 function blocks as the SERVER_1_2 function block replaces those three function blocks. Connect the SERVER_1_2 function block to that system of function blocks to complete the controller panel. Figure 5.6 shows how that is accomplished. Only a part of the controller panel logic in the resource in DEV2 is shown in Figure 5.6 with the SERVER_1_2 function block.

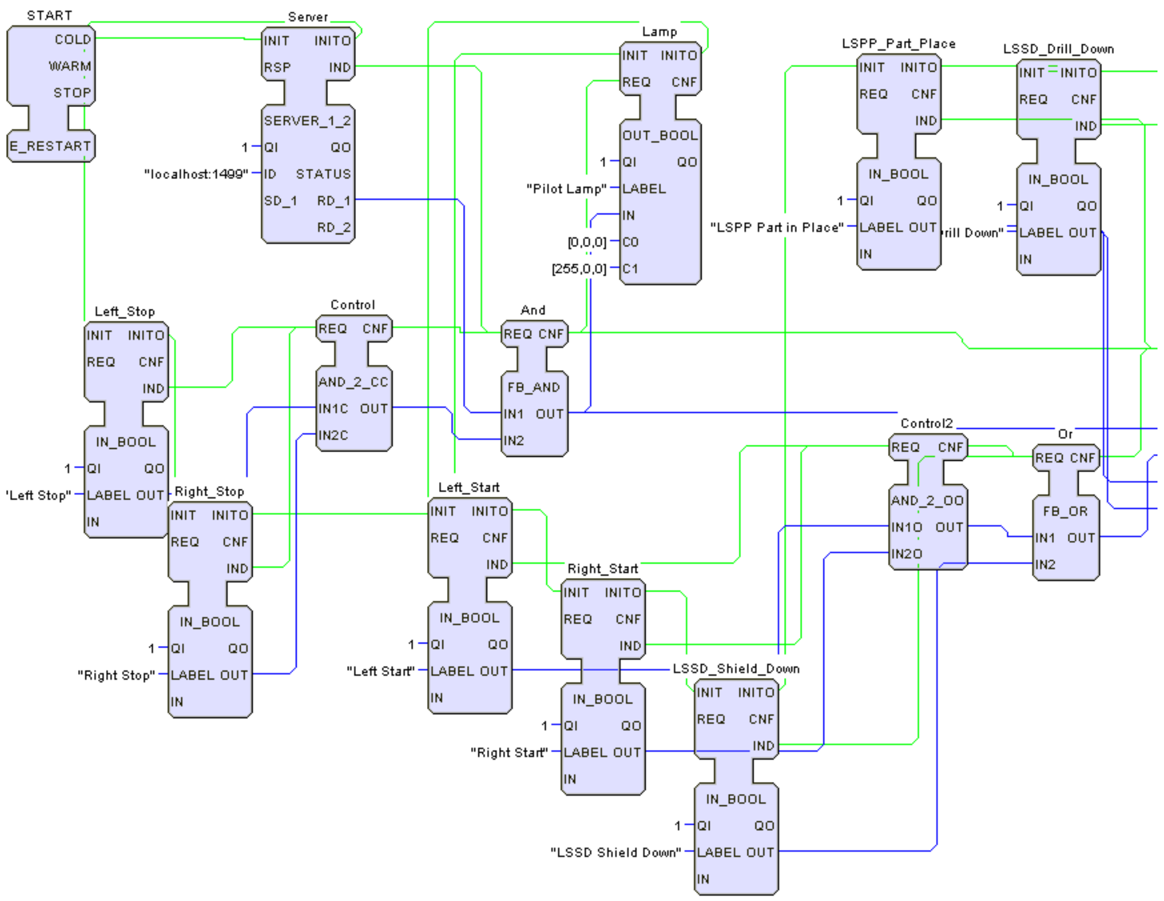


Figure 5.6 Application of Controller Panel Logic with Network Connection

Once the connection is made, save the system, and launch it to see if it works properly. If the system done for the Semiautomatic Drill Press example is correct and used here, then it should work properly since the communication between the two devices already works.

5.3 Summary

This tutorial has shown the following sequential steps for basic use of client and server function blocks. To design a system that uses client and server function blocks.

1. Create a new system.
2. Define as many devices and resources as necessary.
3. First program the function blocks that will yield the data that the client will send.
4. Insert the CLIENT function blocks in resources to send and receive the desired data.
5. Define a proper IP address and port for each CLIENT function block.
6. Insert the SERVER function block in other resources or devices.
7. Define the exact same IP address and port as the CLIENT function blocks for the SERVER function block to receive the data from the clients and send data right back.
8. Using this data, program additional function blocks that may use this received data.

IEC TUTORIAL 6: ADDING JAVA CODE TO A FUNCTION BLOCK

6.1 Introduction

When creating basic function blocks, it is important to input a proper algorithm so it can perform the desired action. In some cases it is necessary to program the algorithm in a high level programming language. The Holobloc software supports programming in Java. This lesson will provide an example on how to incorporate a Java algorithm into a function block.

SITUATION: An Automated Guided Vehicle (AGV) is an autonomous materials handling device that travels on a guided path, called a network. Figure 6.1 shows a network of 9 nodes that are connected by links. Links are identified by node labels. For example, link 12 is the link between nodes 1 and 2.

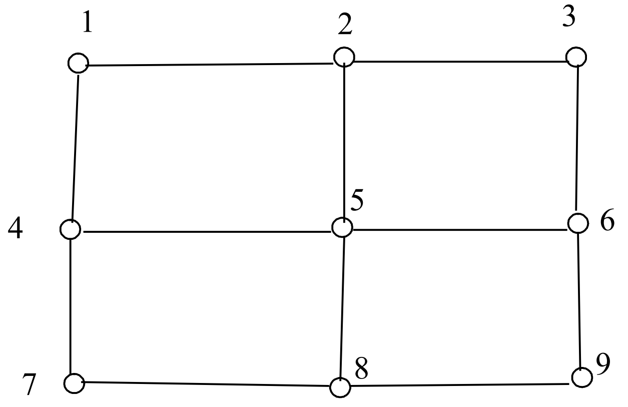


Figure 6.1 Automated Guided Vehicle Network

Information on the status of a link is kept in a database table. The database is named TC_DB and the table, shown in Figure 6.2, is called LINK_STATUS. The table has two fields. LINK_ID is the link identification and L_STATUS indicates whether the link is currently occupied by an AGV (1) or is not occupied (0). The objective of the exercise is to create a function block that will take an inquiry about the status of a particular LINK_ID and return the current L_STATUS of the specified link. (Note: This tutorial assumes the values in LINK_ID are in the database as a number data type, specifically as an integer. Otherwise, more modification to the final Java code will be necessary to get the final program to work properly.)

LINK_ID	L_STATUS
12	0
14	0
23	0
25	0
36	0
45	0

LINK_ID (Number, Long Int.)
L_STATUS (Number, Long Int.)

LINK_ID	L_STATUS
47	0
56	0
58	0
69	0
78	0
89	0

Figure 6.2 LINK_STATUS Table

This exercise will illustrate an important function in a distributed system, i.e., performing a database query. Such a function is usually coded in a high level language. Therefore, this example is an appropriate vehicle to illustrate the use of a Java algorithm within a function block.

Create a basic function block that accomplishes those tasks and implement it into a system. The user should be able to input a LINK_ID and there should be an output L_STATUS. For this we will use text boxes. In addition the system should be controlled by a momentary contact button such that, when it is toggled on, the function block (which we shall call TC_Agent) will read from the database.

6.2 Steps to Add Java Code to a Function Block

Step 1: Setting Up the Framework – Controlling the System

The first thing we are going to do is to establish the "framework" for inputting requirements to the TC_AGENT function block and displaying the output from the function block. Later we will design the TC_AGENT function block and embed it within the framework.

Start a new system configuration and give it a name such as TC_AGENT_1. Complete the Versions information and configure the device as a single resource device. Configure DEV1 as a FRAME_DEVICE and RES1 as a PANEL_RESOURCE as done in the previous tutorials. Do not forget to define the bounds and grid parameters for the device (use any values you wish, they can be easily edited in the future).

For this system, a button (an IN_EVENT function block) will control when the TC_AGENT will perform its function. Therefore, in the application editing window, insert in an IN_EVENT function block. The IN_EVENT function block can be found using the following directory path: src -> hmi Label it BUTTON.

For the text input for the LINK_ID, load and insert an IN_ANY function block. Label it LINK_IN. There exists a function blocks for text inputs and text outputs, but the advantage to using IN_ANY (and OUT_ANY) function blocks is that the type of data that comes in or comes out can defined as any data type without. For this lesson, configure the IN_ANY function block as WSTRING data type, but it can just as easily be

REAL or STRING or any other type. Given that all of the LINK_ID values have two digits (see Figure 6.2), define W as 2. W represents the width of the text box for the input. By making it 2 there will be a limit as to how many characters (or digits in this case) you can type in. Since the input is an IN_ANY function block, insert an OUT_ANY function block and define the type as WSTRING. Label it LINK_OUT. There is no need, however, to define a width for the text box for the output. The IVAL configuration item is for an initial value. By leaving it blank, the initial value defaults to NULL.

Eventually a basic function block will be created to represent the TC_AGENT that will take a LINK_ID from the IN_ANY function block and output L_STATUS to the OUT_ANY function blocks. The IN_EVENT function block will control when the inputted text from the IN_ANY function block will go into the TCAGENT basic function block and then onto the OUT_ANY function block.

Connect the function blocks as shown in Figure 6.3a. This will allow us to test the basic framework. Make sure that the QI input is set equal to 1 for each function block to make sure they will all fire. Map the application to DEV1.RES1, assuming that the framework was not directly programmed in the resource. Make sure to connect the COLD event output of E_RESTART to the INIT event input of BUTTON, and run the application. By entering a link in the LINK_ID text box and clicking on the ON/OFF momentary contact button, an identical string should appear in the LINK_OUT text box. This is shown in Figure 6.3b.

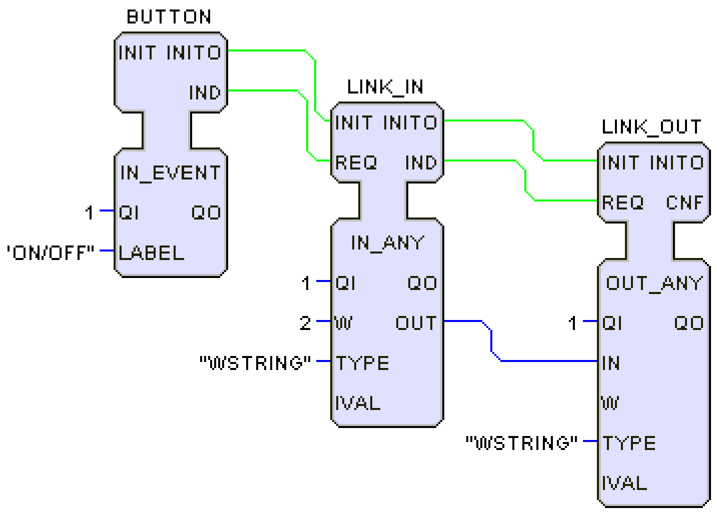


Figure 6.3a Framework Connected for Testing

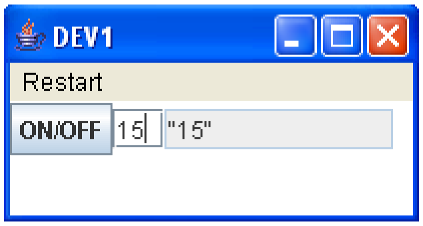


Figure 6.3b Testing Panel for Function Block Framework

Step 2: Creating the TCAGENT Basic Function Block

Save the system configuration (framework) for TC_AGENT_1. The next step is to create a new basic function block for TC_AGENT. The procedure for doing so follows closely that of Tutorial 3, except here we will use Java as the language for the algorithm. This tutorial will continue with the assumption that Tutorial 3 has been completed. If not, it is highly recommended that Tutorial 3 should be completed before moving on, as it will demonstrate step by step how to create a new function block.

At the menu bar, click on File -> New -> FB Type -> Basic FB to create a new Basic Function Block. Name this function block TC_AGENT and input the proper version information. For the Compiler Info, define the header as "package fb.rt.rutgers.b1;". If such a directory does not exist, either change the name of the directory (e.g. if you wish to use the student directory already within the FBDK directory, change the header to package fb.rt.student) or create the sub directories rutgers and b1 within the rt directory. Before continuing on, make sure to input the proper information also for Compilers.

The TC_AGENT function block will not require INIT and INITO events; it will also not need the data input QI and the data output QO. First, delete the two data inputs. Then, click on the execution control chart for the function block and delete the INIT state and click OK to delete any connections with it. Afterwards, delete the INIT algorithm by double clicking on the INIT algorithm on the left side of the screen and select Delete. After completing all of this, delete both the INIT and INITO events. Go back to the Interface entry and right click on the function block to create a new data input and a new data output. Make sure both variables have the data type WSTRING. While the function block editor has a data type called STRING, it is not the same as the String data type in Java. Since it is compatible with the String data type in Java, WSTRING is the appropriate data type. Name the data input, INPUT, and the data output, OUTPUT. Connect the REQ event with the INPUT data input and connect the CNF event with the OUTPUT data output. The basic function should now look like Figure 6.4.

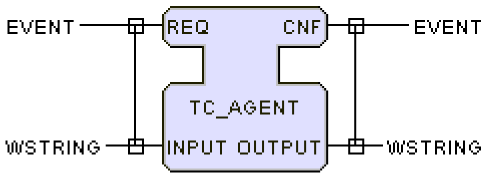


Figure 6.4 Partially Configured TC_AGENT Function Block

For the execution control chart, the only change that needs to be made is the algorithm. Double click on the REQ algorithm on the list to the left and select Java as the language. Do not put anything into the REQ algorithm, leave it blank for now. Everything should be completed and the execution control chart should look like Figure 6.5 below.

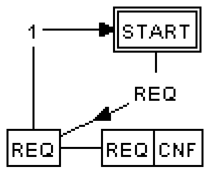


Figure 6.5 Execution Control Chart

Everything that had to be done to configure TC_AGENT function block is now complete. Save the TC_AGENT function block as a Java file in the directory specified for the Compiler Info header. While no algorithm has been added yet, the file should be compiled successfully. Save the TC_AGENT function block as a XML file, and then exit the function block editor. Open up the function block editor and open the system configuration TC_AGENT_1 and insert the TC_AGENT function block into the workspace (use any name for the function block, here it will be called CONTROL). Connect the TC_AGENT function block with the LINK_IN as its input and LINK_OUT as its output. The framework is now fully completed; it should look like Figure 6.6. Save the framework and close the software.

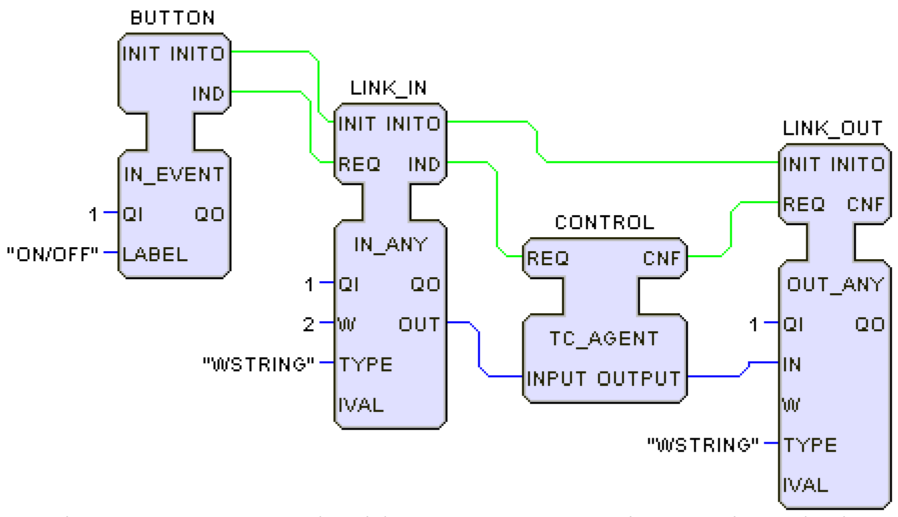


Figure 6.6 Framework with TC_AGENT Basic Function Block

Step 3: Editing the TC_AGENT Algorithm – Part I

When the Java file was saved, it was saved into the same file path as defined in the package. To find the file, assuming the “package fb.rt.rutgers.b1;” was defined as the header, follow the file path C:\> fbdk -> lib -> fb -> rt -> rutgers-> b1. Open up the b1 folder for all file types and you can see the Java file and class for TC_AGENT. Open up the TC_AGENT.java file by right clicking on the file name, click on Open With, and select Notepad. With Notepad, the Java code written for the basic function block TC_AGENT can be edited. Notice the code already in the file. It represents the events, data, and connections programmed for this function block type. Scroll all the way down to the end of the file and you will find the following lines, shown in Figure 6.7 below.

```
  /** ALGORITHM REQ IN Java*/
public void service_REQ(boolean qi){

}
```

Figure 6.7 Java Code for REQ Algorithm within TC_AGENT

In between the brackets is where the code for the REQ algorithm would go if code was entered into the large text space at the bottom of the REQ algorithm window. When creating the basic function block, nothing was put in for the algorithm. At this point, we can develop our own code and have the function block call other Java files. For reading a database, there must be a module that actually performs the reading function as well as a module that controls connections to the Agent database. To accomplish this, consider the following list of Java modules called.

TCAGENT.java is the main program which will call:

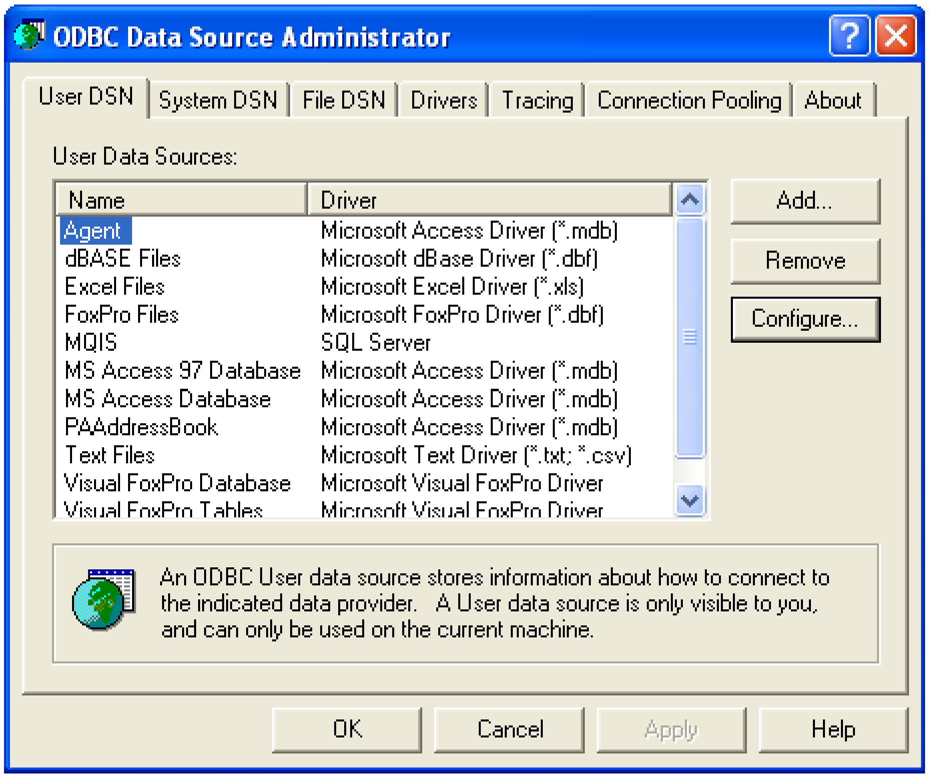
- ➔ A module which will read from a database. To do this, it will call:
    - o A module which will connect and disconnect the database from the Read module. That will require:
        - ▪ The Agent database, that needs to be defined.

The best way to accomplish this is to work from the bottom of the list and up. That way no important steps such as defining the database are missed. Furthermore, the modules that will be called in other programs will already be written and have specific names to be called by so they can be used again.

Step 4: Defining a Database

First, make sure you have a database set up in Access called Agent with a table called LINK_STATUS. In that table there should be the data given in Figure 6.2. Save it to the same directory as the TC_AGENT.java file. For Java files to call other Java files or other programs, they need to be in the same directory or have their exact directory path defined otherwise they will not be found.

In order to use a database in Java programs, the database needs to be defined with a data source name. Otherwise, it will not be recognized by any program that tries to open or close a connection to the database. Under Control Panel on the computer, there should be an icon for Administrative Tools. Click on it and then click on Data Sources (ODBC). A window will appear with a series of tabs. For this case, only be concerned with User DSN. Click on Add to add another data source. Select Microsoft Access Driver (*.mdb) and click on Finish. Name the database source Agent or the name of the database itself. Click on Select, locate the target database, and select it to define the location of the database. Click on OK when finished, which will return you to the ODBC Microsoft Access Setup. To finish the process, click on OK. Once finished, the window under the User DSN tab should look like Figure 6.8 below. Click on OK to close the ODBC Data Source Administrator.


ODBC Data Source Administrator

User DSN | System DSN | File DSN | Drivers | Tracing | Connection Pooling | About

User Data Sources:

Name	Driver
Agent	Microsoft Access Driver (*.mdb)
dBASE Files	Microsoft dBase Driver (*.dbf)
Excel Files	Microsoft Excel Driver (*.xls)
FoxPro Files	Microsoft FoxPro Driver (*.dbf)
MQIS	SQL Server
MS Access 97 Database	Microsoft Access Driver (*.mdb)
MS Access Database	Microsoft Access Driver (*.mdb)
PAAddressBook	Microsoft Access Driver (*.mdb)
Text Files	Microsoft Text Driver (*.txt; *.csv)
Visual FoxPro Database	Microsoft Visual FoxPro Driver
Visual FoxPro Tables	Microsoft Visual FoxPro Driver

An ODBC User data source stores information about how to connect to the indicated data provider. A User data source is only visible to you, and can only be used on the current machine.

Figure 6.8 Data Source Name Window

Step 5: Adding Modules

Once the database has been defined, now a Java program can be written that will use it. Start with the program that will open and close a connection the Agent database. This file will be vital, as opening a connection to the Agent database will allow its contents to be read by other programs. Open up a new file in Notepad and immediately select Save As, change the file type to "All Files," and save the file as ConnectToDataBase.java. The file type must be listed as "All Files" to ensure that the file is saved as a Java file and not a text file. By including the .java extension, it will automatically be defined as a Java file. Since this file will be called by another file which will be called by the TC_AGENT.java file, save it in the same directory as the TC_AGENT.java file. Just like with the database, the Java file needs to be in the same directory so the file called can be found and be used when the program runs. However, unlike the database, you absolutely need to include the compiler information for the function block that will call it as the first line of the Java file. Therefore, insert "package fb.rt.rutgers.b1;" without the quotes as the first line of code for ConnectToDataBase.java. Now you can program the rest of the code to open and close a connection to a database file. The detailed code for ConnectToDataBase.java is in section 6.4.A. When finished, do not forget to save ConnectToDataBase.java. It is a good idea to check for any syntax or other errors in the program before moving on. Compile the ConnectToDataBase.java file with the Java compiler installed on your computer.

Once the ConnectToDataBase.java file is complete, create a new Java file in Notepad for the module that will read from the Agent database. Save it as Read.java in the same directory as the TC_AGENT.java file, ConnectToDataBase.java file, and the TC_DB database. Now write the code for Read.java. Do not forget that this module must call the ConnectToDataBase module and it must have its first line be "package fb.rt.rutgers.b1;" without the quotes. The complete code for Read.java is in section 6.4.B. (Note: In section 6.4.B, notice that the Read module's input is an integer and the output is a string. Since the TC_AGENT function block's input and output are strings, you may want to make some alterations to the code given in section 6.4.B.) If an error occurs compiling the Read module, then it is necessary to write the TC_AGENT.java file-where it should compile correctly afterwards.

Step 6: Editing the TC_AGENT Algorithm – Part II

Now that the Read.java and ConnectToDataBase.java files are created and in the same directory as the TC_AGENT.java file, now the code can be written for the REQ algorithm for the TC_AGENT basic function block. The fbdk software does not have its own compiler; thus, any Java code that is in the algorithm must be inserted into the text space provided in the algorithm window, which will be compiled when the Java file is saved. Code can not be inputted into the resulting Java file in Notepad; it will not be there when the Java file is saved again in the fbdk software. Therefore, open the fbdk editor, load the TC_AGENT function, and double click on the REQ algorithm at the bottom of the list on the left. Now the code can be written in the large text space at the bottom of the window that will pop up from double clicking on the REQ algorithm. This code will take the data coming into the function block, call the Read module, which will call the ConnectToDataBase module, check for the corresponding L_STATUS for the inputted LINK_ID, and output that L_STATUS from the function block. All that is necessary to do this is to equate a variable to the INPUT.value (the value of the data coming into the INPUT data input), call the Read module (which will take care of everything but the output), and equate the OUTPUT.value (the value of the data that will come from the OUTPUT data output) from a string variable. Make sure that the line that calls the Read module is put in at the beginning of the program, just after the first open bracket under the class definition. The complete, detailed code for the REQ algorithm of the TC_AGENT is shown in section 6.4.C.

Save the TC_AGENT function block again as a Java file and it will be compiled automatically. Any errors in either of the TC_AGENT.java, Read.java, or ConnectToDataBase.java files will be found and the program will tell you where the problem is. The reason why errors in these other modules are also found when compiling TC_AGENT is because when TC_AGENT is run, those other modules are called. If they have errors, then TC_AGENT will not work at all. If there are no errors, save the TC_AGENT function block as an XML file. Close the program to ensure that the TC_AGENT function block works with the most recent Java class file when used. Start the program and load the TC_AGENT_1 system configuration, check to make sure the framework is in order, and run the program. Test the program by typing in 12 in the input textbox and then clicking on the On/Off checkbox. If "0" comes up in the other textbox, then the program works.

EXERCISE: If you insert a number that is not in the database for LINK_ID or a letter in the input textbox, nothing will change or nothing will appear in the output textbox. By checking the corresponding cmd.exe window, there will be the line "java.lang.NumberFormatException: null." While it is not wrong, it does not directly tell the user that there is an error. Edit the Read.java and TCAGENT.java files in a way such that whenever a number not in the database or a letter is typed in, the output textbox yields the word "Error."

6.3 Summary

This tutorial has shown the following sequential steps to implement Java code and modules with function blocks. To implement Java code with function blocks:

1. Create a new basic function block and set up the necessary properties, interface, and execution control chart.
2. Define the algorithm(s) as Java language.
3. When finished setting up the basic function block, save it first as a Java file and compile it. Save it as an XML file afterwards.
4. Define any databases or other files that the Java program will get and receive any information from. Make sure they are saved in the same directory as the Java file.
5. Open up the directory where the Java file was saved and open up the Java file in Notepad (or any other applicable Java-editing program).
6. Write any Java modules and programs that the function block will use. You may choose to compile each file separately to check for any errors.
7. Save any modules and programs with the extension .java under the file type "All Files."
8. Write the Java code within the function block for the algorithm(s) by double clicking on the algorithm name on the left side of the screen and inputting the Java code into the text box at the bottom of the algorithm's window.
9. Save and overwrite the Java file for the function block, the Java file will automatically be compiled.
10. Implement the function block in a system and run it to test if it works. If not, you may have to modify the Java code, which will require restarting the function block editor program and re-compiling the file.

6.4.A Code for the Module ConnectToDataBase.java

```
package fb.rt.rutgers.b1;
import java.sql.*;
import java.util.*;

public class ConnectToDataBase{

        protected Connection connection;

        public void openConnection(){
                try{
                        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
                        String url ="jdbc:odbc:Agent";
                        connection = DriverManager.getConnection(url);
                }
                catch(ClassNotFoundException cnfe){
                        System.err.println(cnfe);
                }
                catch(SQLException sqle){
                        System.err.println(sqle);
                }
        }
        public void closeConnection(){
                if(connection != null)
                try{
                        connection.close();
                        connection = null;
                }
                catch(SQLException sqle){
                        System.err.println(sqle);
                }
        }
}
```

6.4.B Code for the Module Read.java

```
package fb.rt.rutgers.b1;
import java.sql.*;
import java.util.*;

class Read extends ConnectToDataBase{
	public String read_database(int poll){
		try{
			openConnection();
			Statement statement = connection.createStatement();
			String query = "select L_STATUS from LINK_STATUS where
LINK_ID = " + poll;
			ResultSet rs = statement.executeQuery(query);

		if(rs.next()){
			String result = rs.getString("L_STATUS");
			closeConnection();
			return result;
		}
		else{
			return null;
		}
		}
		catch(SQLException sqle){
			return null;
		}
	}
	public static void main(String[] args){
		Read r = new Read();

		System.out.println(r.read_database(12));
	}
}
```

6.4.C Code for the Function Block Java File TCAGENT.java

REQ Algorithm

```
int in = 0;
try{
        in = Integer.parseInt(""+INPUT.value);
}
catch(NumberFormatException e){
        System.err.println(e);
}

Read readDB = new Read();
try{
        int out1 = Integer.parseInt(readDB.read_database(in));
        String out2 = Integer.toString(out1);
        OUTPUT.value = out2;
        CNF.serviceEvent(this);
}
catch(NumberFormatException e1){
        System.err.println(e1);
}
```

IEC TUTORIAL 7: DESIGNING DATABASE QUERY FUNCTION BLOCKS

7.1 Introduction

Certain situations require the use of a database within a function block's algorithm. However, the function block editor program does not have a function block available that works with databases. As a result, function blocks must be made that connects to the database and runs commands using it. This was done in Tutorial 6. However, the database, the table, and the conditions used in that tutorial were coded directly into the function block. The function block does not account for situations where there are other or multiple databases that are used-the one created in Tutorial 6 only works with the database TC_DB. The function block does not query the user for a database that may be something other than Agent. This lesson will provide an example on how to make a function block that queries a database that the user defines within the function block.

SITUATION: An Automated Guided Vehicle (AGV) is an autonomous materials handling device that travels on a guided path, called a network. Figure 7.1 shows a network of 9 nodes that are connected by links. Links are identified by node labels. For example, link 12 is the link between nodes1 and 2.

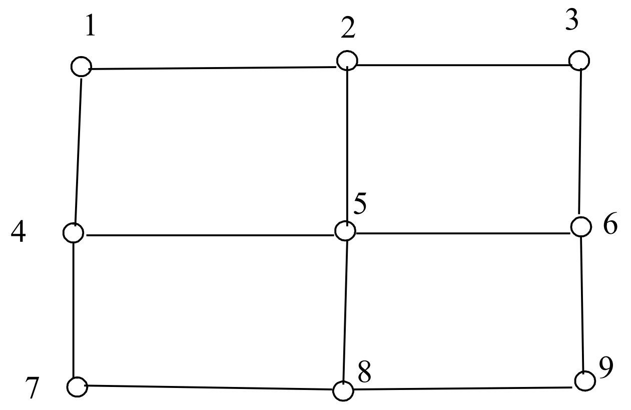


Figure 7.1 Automated Guided Vehicle Network

Information on the status of a link is kept in a database table. The database is named Agent and the table, shown in Figure 7.2, is called LINK_STATUS. The table has two fields. LINK_ID is the link identification and L_STATUS indicates whether the link is currently occupied by an AGV (1) or is not occupied (0). The objective of the exercise is to create a function block that will take an inquiry about the status of a particular LINK_ID and return the current L_STATUS of the specified link.

LINK_ID	L_STATUS
12	0
14	0
23	0
25	0
36	0

LINK_ID (Number, Long Int.)
L_STATUS (Number, Long Int.)

LINK_ID	L_STATUS
45	0
47	0
56	0
58	0
69	0
78	0
89	0

Figure 7.2 LINK_STATUS Table

This exercise will perform a database query with variables for the database, table, and conditions as opposed to static figures for each. A high level programming language is necessary to accomplish this. A Java algorithm will be used within the function block as a result.

Create a basic function block called DB_SELECT_1 that accomplishes the task recognizing and reading from a database that the user defines. The user should be able to input a LINK_ID and there should be an output L_STATUS. For this we will use text boxes. To define the database, the user will provide the name of the database that will be used, the table within that database, and the SELECT and WHERE conditions used to retrieve the L_STATUS. In addition, the system should be controlled by a momentary contact button such that, when it is toggled on, the function block will read from the database.

7.2 Steps in Creating a Database Query Function Block

Step 1: Setting Up the Framework – Controlling the System

The first thing we are going to do is to establish the "framework" for inputting requirements to the DB_SELECT_1 function block and displaying the output from the function block. Later, the DB_SELECT_1 function block will be created, designed, and eventually embedded within the framework.

Start a new system configuration and give it a name such as DB_Select. Complete the Versions information, define the default device DEV1 as a FRAME_DEVICE, and define the resource within DEV1 as a PANEL_RESOURCE called RES1. Do not forget to define the bounds and grid parameters for the device (there is no specific size necessary).

For this system, a button (an IN_EVENT function block) will control when the DB_SELECT_1 function block will perform its function. Therefore, in the application editing window, insert in an IN_EVENT function block. Label it On_Off. For the text input for the LINK_ID, insert an IN_ANY function block and name it Input. Configure the IN_ANY function block as WSTRING data type, since the inputted information will be a string. While the function block editor has a data type called STRING, it is not the same as the String data type in Java. Since it is compatible with the String data type in Java, WSTRING is the appropriate data type. Given that we know that all of the LINK_ID values have two digits (see Figure 7.2), set W as 2. Since the input is an IN_ANY function block, insert an OUT_ANY function block into the workspace. Name this function block Output and define the type as WSTRING. There is no need, however, to define a width for the text box for the output. There is also no need to define an initial value for either of the IN_ANY and OUT_ANY function blocks. Do not forget to assign 1 to all three function block's QI data input to make sure they will all function properly when the program is run.

Eventually, a basic function block will be developed to represent the DB_SELECT_1 that will take a LINK_ID from the IN_ANY function block and output L_STATUS to the OUT_ANY function blocks. Unlike in Tutorial 6, the user will have to assign to the function block the database (Agent), the table (LINK_STATUS), and the conditions to get the information (L_STATUS, LINK_ID). The IN_EVENT function block will control when the inputted text from the IN_ANY function block will go into the DB_SELECT_1 basic function block and then onto the OUT_ANY function block.

Connect the function blocks as shown in Figure 7.3a. This will allow us to test the basic framework. Map the application to DEV1.RES1. Go to RES1, connect the COLD event output of START to the INIT event input of BUTTON, and run the application. By entering a link in the Input text box and clicking on the On/Off momentary contact button, an identical string should appear in the Output text box. This is shown in Figure 7.3b.

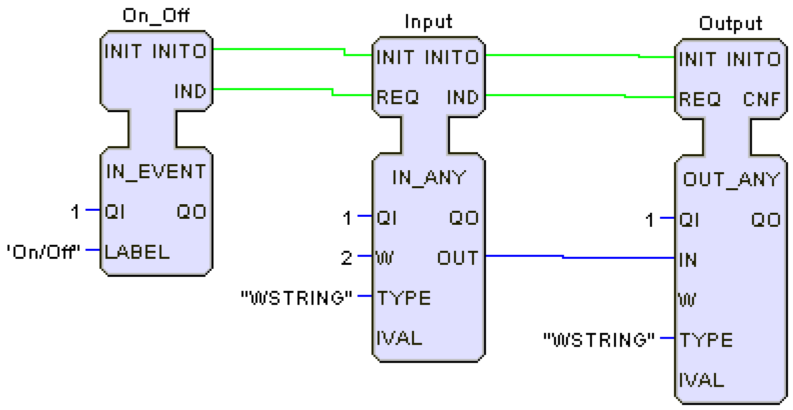


Figure 7.3a Framework Connected for Testing

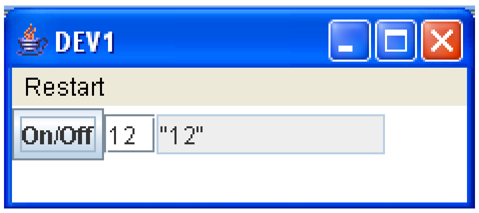


Figure 7.3b Testing Panel for Function Block Framework

Step 2: Creating the DB_SELECT_1 Basic Function Block

Save the system configuration (framework) for DB_Select. In this section we are going to define a new basic function block called DB_SELECT_1. The procedure for doing so follows closely that of Tutorial 3, except that the algorithm will be written in Java. This tutorial will continue with the assumption that Tutorial 3 has been completed. If not, it is highly recommended that Tutorial 3 should be completed before moving on, as it will demonstrate step by step how to create a new function block.

At the menu bar, click on File -> New -> FB Type -> Basic FB to create a new Basic Function Block. Name this function block DB_SELECT_1 and input the proper version information. For the Compiler Info, define the header as "package fb.rt.rutgers.b1;". If such a directory does not exist, either change the name of the directory (e.g. if you wish to use the student directory already within the FBDK directory, change the header to package fb.rt.student) or create the sub directories rutgers and b1 within the rt directory. Before continuing on, make sure to input the proper information also for Compilers.

The DB_SELECT_1 function block will not require INIT and INITO events; it will also not need the data input QI and the data output QO. First, delete the two data inputs. Then, click on the execution control chart for the function block and delete the INIT state and click OK to delete any connections with it. Afterwards, delete the INIT algorithm by double clicking on the INIT algorithm on the left side of the screen and select Delete. After completing all of this, delete both the INIT and INITO events.

For the DB_SELECT_1 function block, 5 data inputs and a single data output are necessary. Go back to the Interface entry and right click on the function block to create a new data input. All of the data inputs and the data output for DB_SELECT_1 will work with the data type WSTRING; be sure to define WSTRING as the data type for all data. Create the first data input and name it DB; this data input will control which database is being used. Create a second data input, and name it TABLE. This data input will specify which table in the database the function block will use. Create a third data input named SELECT_Attribute, which specifies the column in the database table that data will be retrieved from based on the database query. Create a fourth data input named WHERE_Attrribute, which specifies the WHERE condition for the database query. Create the fifth and final data input named INPUT_VAL, which will be similar to the TC_AGENT data input INPUT from Tutorial 6. For this function block, this data input will eventually be the value of LINK_ID that will be checked in the Agent database. Create a data output named OUTPUT_VAL, which will be the value of L_STATUS in the Agent database. Connect the REQ event with the INPUT_VAL data input and connect the CNF event with the OUTPUT_VAL data output. The basic function should now look similar to Figure 7.4.

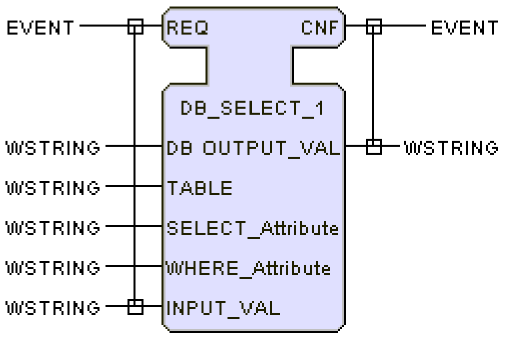


Figure 7.4 Partially Configured DB_SELECT_1 Function Block

Notice that only one of the data inputs is connected with the REQ event. The other four data inputs are not connected to the REQ event on purpose. By not connecting the data with the events, this means the database, table, and SELECT and WHERE constraints will have to be given during the design of the system. Having to code each exact constraint in the Java algorithm requires changing the actual Java code, recompiling the function block, and reloading the system numerous times anytime a change is made. Instead, by setting up the function block in this way, the user can right click on the data input, select Connect to assign it a base value (double quotes for WSTRING), and double click on the resulting double quotes to assign the necessary database name, table, and SELECT and WHERE constraints. This way, the function block can be run without having to recompile the Java algorithm or reload the entire system. This, in addition to the Java algorithm, makes this DB_SELECT_1 function block a database query function block.

For the execution control chart, the only change that needs to be made is the algorithm. Set the REQ algorithm to use Java as its language by double clicking on it at the bottom of the list on the left and selecting Java as its language. Do not put anything into the REQ algorithm, leave it blank for now. Everything should be completed and the execution control chart should look like Figure 7.5.

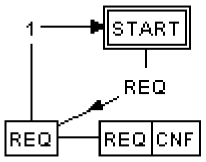


Figure 7.5 Execution Control Chart

Everything that had to be done to configure TC_AGENT function block is now complete. Save the DB_SELECT_1 function block as a Java file in the directory specified for the Compiler Info header. While no algorithm has been added yet, the file should be compiled successfully. Save the DB_SELECT_1 function block as a XML file, and then exit the function block editor. Open up the function block editor and open the system configuration DB_Select and insert the DB_SELECT_1 function block into the workspace (use any name for the function block, here it will be called DBSelect). Connect the TC_AGENT function block with Input and Output. The framework is now fully completed; it should look like Figure 7.6. Save the framework and close the software.

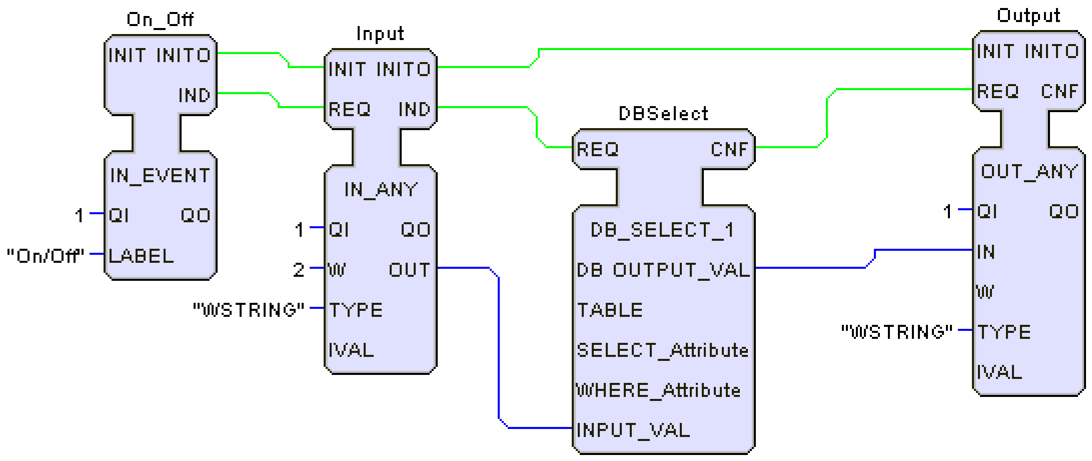


Figure 7.6 Framework with DB_SELECT_1 Basic Function Block

Step 3: Editing the DB_SELECT_1 Algorithm – Part I

When the Java file was saved, it's saved into the same file path as defined in the package. To find that file, follow the path C:\ -> fbdk -> lib -> fb -> rt -> rutgers-> b1. Open up the b1 folder for all file types and you can see the Java file and class for DB_SELECT_1. Open up the DB_SELECT_1.java file by using Notepad. With Notepad, the Java code written for the basic function block DB_SELECT_1 can be seen and edited. Notice the code already in the file. It represents the events, data, and connections programmed for this function block type. Scroll all the way down to the end of the file and you will find the following lines, shown in Figure 7.7.

```
  /** ALGORITHM REQ IN Java*/
public void service_REQ(boolean qi){

}
```

Figure 7.7 Program Code

In between those two brackets is where the code for the REQ algorithm goes. When creating the basic function block, nothing was put in for the algorithm. At this point, we can put in our own code and have the function block call other Java files. For reading a database, there must be a module that actually performs the reading function as well as a module that controls connections to the Agent database. In addition, the database, its tables, and the columns in the database will have to be defined as variables given that the specification for each will come from the function block. To accomplish this, consider the following list of Java modules called.

DB_SELECT_1.java is the main program which will call:

- A module which will read from a database. To do this, it will call:
    - A module which will connect and disconnect the database from the ReadDB module. That will require:
        - The Agent database, that needs to be defined.

The best way to accomplish this is to work from the bottom of the list and up. That way no important steps such as defining the database are missed. Furthermore, the modules that will be called in other programs will already be written and have names to be called by.

Step 4: Connecting to a Variable Database

First, make sure you have a database set up in Access called Agent with a table called LINK_STATUS. In that table there should be the data given in Figure 7.2. Save it to the same directory as the DB_SELECT_1.java file. Give it a data source name after saving it, as done previously in Tutorial 6. If you have done Tutorial 6 previously, you may already have that database available and with a data source name. Copy the Agent.mdb file and paste it in the same directory. Then, rename the file Agent2.mdb and define it with a data source name (see Tutorial 6 for more details). This Agent2 database will be used for the purposes of testing the DB_SELECT_1 function block later.

In Tutorial 6, in addition to a Java algorithm for TC_AGENT, two modules were also programmed in Java. For the purposes of this tutorial, we will modify those modules to make the formerly static attributes involving the use the database variable. Open up ConnectToDataBase.java, immediately save it as ConnectToDataBaseGen.java (make sure "All Files" is listed as Save as type in the resulting window or the file will be saved as a text file instead of a Java file), and close Notepad. This will ensure that the original file remains unchanged and not cause errors in other programs and function blocks that use that file. Open up ConnectToDataBaseGen.java in Notepad and add the necessary code such that the file will open and close connections to a variable database name (assuming the inputted database name already has a defined data source). The detailed code for the file ConnectToDataBaseGen.java is listed in section 7.4.A.

Step 5: Modifying SQL to Read from a Variable Database

Now that we have written Java module can connect to any available database based off its name, we can write a Java module that will use it to read information from that database. A similar Java module was written for this purpose in Tutorial 6 called Read.java, so we shall modify it to account for the new variables. Open up Read.java, save it as ReadDB.java (make sure "All Files" is listed as Save as type in the resulting window), and close Notepad. Again, this ensures that the contents of Read.java are not changed and does not affect other programs and function blocks that use it.

Now open up ReadDB.java in Notepad to make the necessary changes to the code. In this module, it is imperative that the ConnectToDataBaseGen.java module is run first and connects to the database that the user will define in the DB_SELECT_1 function block. In addition, the SQL command must be edited to include the new variables. The line that is equal to String query is in the SQL language. SQL language, among other uses, is used to retrieve information from a database based on a set of conditions. The original query selects the table LINK_STATUS and retrieves the value of the column L_STATUS where the LINK_ID is equal to the inputted integer poll. Whatever poll is equal to, the program will find that value in the LINK_ID column in the LINK_STATUS table, and then locate the corresponding value in the L_STATUS column. Originally, the table name and the names of the columns for the conditions were already defined within the SQL command. For this function block, those names become variables. Therefore, modify the command to make the name of the table and the conditions to be variables that the user will define in the DB_SELECT_1 function block. The detailed code for ReadDB.java is listed in section 7.4.B.

Step 6: Writing the DB_SELECT_1 REQ algorithm.

Now that the ReadDB.java and ConnectToDataBaseGen.java files are created and in the same directory as the DB_SELECT_1.java file, now the code can be written for the REQ algorithm for the DB_SELECT_1 basic function block. Open the DB_SELECT_1 function block and double click on the REQ algorithm at the bottom of the left side of the screen. A window will pop up with a large, blank text space at the bottom. In this text space, the Java code for the REQ algorithm will be implemented here. This code must be written such that it will take the data coming into the function block, call the ReadDB module, which will call the ConnectToDataBaseGen module, check for the corresponding for the inputted INPUT_VAL in the column specified by the WHERE_Attribute, and output the value from SELECT_Attribute to the OUTPUT_VAL of the function block. All that is necessary to do this is to equate a variable to the INPUT_VAL.value (the value of the data coming into the INPUT data input), call the ReadDB module (which will take care of everything but the output), and equate the OUTPUT.value (the value of the data that will come from the OUTPUT data output) from a string variable. Make sure that the ReadDB module is called not just with the value of INPUT_VAL, but also the values of DB, TABLE, SELECT_Attribute, and WHERE_Attribute. The complete, detailed code for the REQ algorithm of the DB_SELECT_1 is shown in section 7.4.C.

Save the DB_SELECT_1.java file and the file will automatically be compiled. If there are any errors in either of the two modules, go back to Notepad, fix them, and recompile them. If the error is with the code specifically for the REQ algorithm, double click on the REQ algorithm and fix the errors in the text space available. If there are no errors, save DB_SELECT_1 as a XML file, close the program, open up the program again, and load the DB_Select system configuration. The program needs to be closed so that the DB_SELECT_1 function block in the system will work with the most recent compiled Java class file. Click on RES1 to bring up the framework in the resource, and right click on the data inputs DB, TABLE, SELECT_Attribute, and WHERE_Attribute separately and connect them with double quotes. Double click on each set of double quotes to define the parameters. When finished, the function block should look like Figure 7.8.

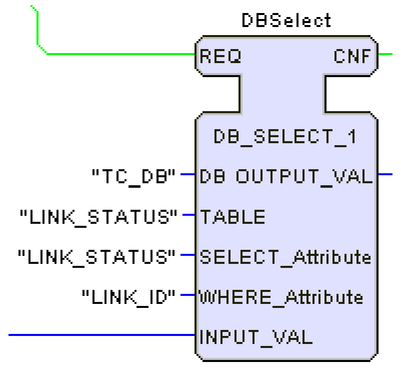


Figure 7.8 Fully Configured DB_SELECT_1 Function Block in Framework

When finished defining each parameter, save the system configuration, and launch the program. Type in a LINK_ID such as 12 or 14, click on the On/Off button, and check the resulting value with the values in the Agent database. If they match, then the DB_SELECT_1 function block and the program work properly. To further test it, close the Java pop-up window (not the function block editor) and open up Agent2 (which we defined earlier) in Access and give some new numeric values to L_STATUS. Go back to the resource RES1 in the function block editor, double click on the string next to DB to redefine the parameter, and change the parameter to "Agent2." Launch the program again and input different LINK_ID values for various resulting outputs. If it works properly, the outputs from the user-inputted LINK_ID should match with the L_STATUS values for each LINK_ID in the Agent2 database.

7.3 Summary

This tutorial has shown the following sequential steps for designing and using a database query function block. To create a database query function block:

1. Set up a system as a framework for the function block.
2. Create a new basic function block and define its compiler information.
3. Define the interface of the composite function blocks: events, data inputs, and data outputs.
4. Connect data inputs and outputs with corresponding events. Be sure not to link events to data inputs that the user will have to configure.
5. Make sure the Execution Control Chart has all the necessary states and transitions. Especially make sure that the algorithm for the event is defined as Java.
6. Save the function block as Java, compile it, and then save it as XML.
7. Insert this function block into the system framework. Save the framework and close the program.
8. Write the necessary Java modules that the function block will call. Be sure to make the name of the database, the table, and the conditions variable since the user will specify that in the function block.
9. Make sure that any databases that can/will be used have a defined data source.

10. Open up the program and the function block, double click on the algorithm name, and implement the Java code for that specific algorithm.
11. Save the function block as a Java file, where the Java file will automatically be compiled. If there are no errors, save the function block again as an XML file. Close the program, and then open it again and load the system configuration.
12. Before launching the program, configure the function block with the name of the database to be used, the name of the table that will be used, and the names of the columns to check for and retrieve data from.

EXERCISE: The DB_SELECT_1 function block retrieves information from a user defined database. It is possible, however, to update a database in the same way. Defining a database, table, and conditions to be used is done exactly the same way it was done in this tutorial. However, additional inputs and a new SQL command are necessary for updating a database. Create a new function block called DB_UPDATE_1 and write or edit the necessary Java algorithms and modules to accomplish this task. (Note: The answer to this exercise, complete with Java code, is in section 7.4.D.)

EXERCISE: The DB_SELECT_1 function block only returns one value from a column in a user defined database. However, it is possible to select and return more than one value in an SQL query. SQL queries can be modified to select information from more than one column. In addition, the WHERE condition in an SQL query does not have to be a value of a column. An entire SQL query for that WHERE condition, a sub-query, can further specify the constraint. Create a new function block called DB_NEST_1_SELECT_3 and write or edit the necessary Java algorithms and modules so that the function block can accomplish the following: 1) return 3 values from 3 separate columns and 2) instead of a solitary where value, implement a sub-query in its place. Be sure to have the SQL query select all three columns at one time. Also, the sub-query only needs to have one value for the WHERE condition. (Note: The answer to this exercise, complete with Java code, is in section 7.4.E.)

7.4.A Code for the Module ConnectToDataBaseGen.java

```
package fb.rt.rutgers.b1;
import java.sql.*;
import java.util.*;

public class ConnectToDataBaseGen{
        private String db = new String();
        protected Connection connection;
        public ConnectToDataBaseGen(String dbName){
                db = dbName;
        }
        public void openConnection(){
                try{
                        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
                        String url ="jdbc:odbc:" + db;
                        connection = DriverManager.getConnection(url);
                }
                catch(ClassNotFoundException cnfe){
                        System.err.println(cnfe);
                }
                catch(SQLException sqle){
                        System.err.println(sqle);
                }
        }
        public void closeConnection(){
                if(connection != null)
                try{
                        connection.close();
                        connection = null;
                }
                catch(SQLException sqle){
                        System.err.println(sqle);
                }
        }
}
```

7.4.B Code for the Module ReadDB.java

```
package fb.rt.rutgers.b1;
import java.sql.*;
import java.util.*;

class ReadDB extends ConnectToDataBaseGen{
        public ReadDB(String dbName){
                super(dbName);
        }
        public String read_database(int poll, String Table, String Select, String Where){
                try{
                        openConnection();
                        Statement statement = connection.createStatement();
                        String query = "SELECT " + Select + " FROM " + Table + "
WHERE " + Where + "=" + poll;
                        ResultSet rs = statement.executeQuery(query);
                if(rs.next()){
                        String result = rs.getString(Select);
                        closeConnection();
                        return result;
                }
                else{
                        return null;
                }
                }
                catch(SQLException sqle){
                        System.err.println(sqle);
                        return null;
                }
        }
}
```

7.4.C Code for the Function Block DB_SELECT_1.java

```
int in = 0;
String Table;
String Select;
String Where;
try{
        in = Integer.parseInt(""+INPUT_VAL.value);
}
catch(NumberFormatException e){
        System.err.println(e);
}
Table = TABLE.value;
Select = SELECT_Attribute.value;
Where = WHERE_Attribute.value;
ReadDB readDB = new ReadDB(""+DB.value);
try{
        String out = readDB.read_database(in, Table, Select, Where);
        OUTPUT_VAL.value = out;
        CNF.serviceEvent(this);
}
catch(NumberFormatException e){
        System.err.println(e);
}
```

7.4.D DB_UPDATE_1 Interface & Java Code

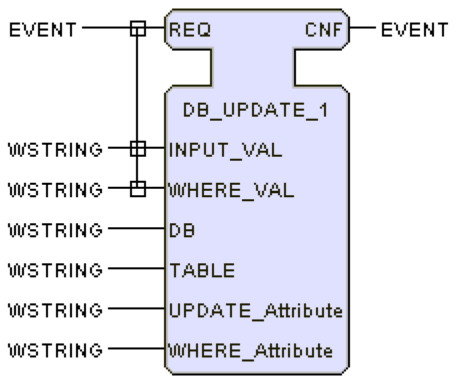


Figure 7.9 Interface of DB_UPDATE_1 Function Block

The DB_UPDATE_1 function block uses the file ConnectToDataBaseGen.java, detailed in section 7.4.A. It uses its own Java module called WriteDB. The WriteDB.java code is as follows:

```
package fb.rt.rutgers.b1;
import java.sql.*;
import java.util.*;

class WriteDB extends ConnectToDataBaseGen{
        public WriteDB(String dbName){
                super(dbName);
        }
        public String write_database(int in1, int in2, String Table, String Update, String
Where){
                try{
                        openConnection();
                        Statement statement = connection.createStatement();
                        String update = "UPDATE " + Table + " SET " + Update + " = " +
in1 + " WHERE " + Where + "=" + in2;
                        statement.executeUpdate(update);
                        closeConnection();
                        String OK = "OK";
                        return OK;
                }
                catch(SQLException sqle){
                        System.err.println(sqle);
                        return null;
                }
        }
}
```

The Java algorithm for the REQ state (the only non-initial state needed) of the DB_UPDATE_1 function block uses the WriteDB Java module and is shown below. This code is to be entered in the text space for the REQ algorithm (the only algorithm needed) in the DB_UPDATE_1 function block.

```
int in1 = 0; //Input
int in2 = 0; //Update
String Table;
String Update;
String Where;
try{
        in1 = Integer.parseInt(""+INPUT_VAL.value);
}
catch(NumberFormatException e){
        System.err.println(e);
}
try{
        in2 = Integer.parseInt(""+WHERE_VAL.value);
}
catch(NumberFormatException e){
        System.err.println(e);
}
Table = TABLE.value;
Update = UPDATE_Attribute.value;
Where = WHERE_Attribute.value;
WriteDB writeDB = new WriteDB(""+DB.value);
try{
        String out = writeDB.write_database(in1, in2, Table, Update, Where);
        CNF.serviceEvent(this);
}
catch(NumberFormatException e){
        System.err.println(e);
}
```

7.4.E DB_NEST_1_SELECT_3 Interface & Java Code

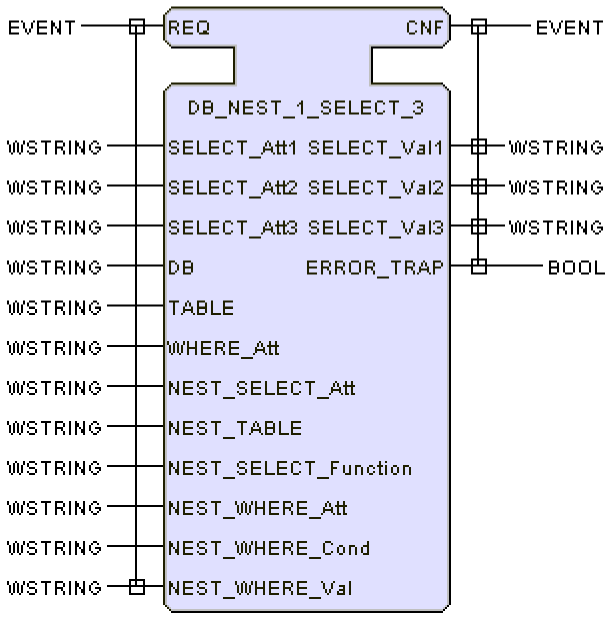


Figure 7.10 Interface of DB_NEST_1_SELECT_3 Function Block

The DB_NEST_1_SELECT_3 function block uses the file ConnectToDataBaseGen.java, detailed in section 7.4.A. It uses its own Java module called NestReadDB3. The NestReadDB3.java code is as follows:

```
package fb.rt.rutgers.b1;
import java.sql.*;
import java.util.*;

class NestReadDB3 extends ConnectToDataBaseGen{
        public NestReadDB3(String dbName){
                super(dbName);
System.out.println(dbName);
        }
        public String[] read_database(String Select1, String Select2, String Select3, String
Table, String WhereAtt, String NestSelect, String NestTable, String NestSelFunction,
String NestWhereAtt, String NestWhereCond, String NestWhereValS){
                String [] Array = new String[3];
                try{
                        openConnection();
                        Statement statement = connection.createStatement();
                        String query = "SELECT " + Select1 + ", " + Select2 + ", " +
Select3 + " FROM " + Table + " WHERE " + WhereAtt + " IN (SELECT " +
NestSelFunction + "(" + NestSelect + ") FROM " + NestTable + " WHERE " +
NestWhereAtt + " " + NestWhereCond + " " + NestWhereValS + ")";
System.out.println(query);
                        ResultSet rs = statement.executeQuery(query);
                if(rs.next()){
                        String result1 = rs.getString(Select1);
                        String result2 = rs.getString(Select2);
                        String result3 = rs.getString(Select3);
                        Array[0] = result1;
                        Array[1] = result2;
                        Array[2] = result3;
                        closeConnection();
                        return Array;
                }
                else{
                        Array[0] = "null";
                        Array[1] = "null";
                        Array[2] = "null";
                        return Array;
                }
                }
                catch(SQLException sqle){
                        System.err.println(sqle);
                        Array[0] = "null";
                        Array[1] = "null";
                        Array[2] = "null";
                        return Array;
                }
        }
}
```

The Java algorithm for the REQ state (the only non-initial state needed) of the DB_NEST_1_SELECT_3 function block uses the NestReadDB3 Java module and is shown below. This code is to be entered in the text space for the REQ algorithm (the only algorithm needed) in the DB_NEST_1_SELECT_3 function block.

```
        String Select1;
        String Select2;
        String Select3;
        String Table;
        String WhereAtt;
        String NestSelect;
        String NestTable;
        String NestSelFunction;
        String NestWhereAtt;
        String NestWhereCond;
        String NestWhereVal;
        Select1 = SELECT_Att1.value;
        Select2 = SELECT_Att2.value;
        Select3 = SELECT_Att3.value;
        Table = TABLE.value;
        WhereAtt = WHERE_Att.value;
        NestSelect = NEST_SELECT_Att.value;
        NestTable = NEST_TABLE.value;
        NestSelFunction = NEST_SELECT_Function.value;
        NestWhereAtt = NEST_WHERE_Att.value;
        NestWhereCond = NEST_WHERE_Cond.value;
        NestWhereVal = NEST_WHERE_Val.value;
        String NestWhereValS = "\"" + NestWhereVal + "\"";
        NestReadDB3 nestreadDB3 = new NestReadDB3("" + DB.value);
        String[] out = nestreadDB3.read_database(Select1, Select2, Select3, Table,
WhereAtt, NestSelect, NestTable, NestSelFunction, NestWhereAtt, NestWhereCond,
NestWhereValS);
        String SelectVal1 = out[0];
        String SelectVal2 = out[1];
        String SelectVal3 = out[2];
        if(SelectVal1.equals("null") || SelectVal2.equals("null") ||
SelectVal3.equals("null")){
                SELECT_Val1.value = "null";
                SELECT_Val2.value = "null";
                SELECT_Val3.value = "null";
                ERROR_TRAP.value = true;
        }
        else{
                SELECT_Val1.value = SelectVal1;
                SELECT_Val2.value = SelectVal2;
                SELECT_Val3.value = SelectVal3;
        ERROR_TRAP.value = false;
}
```