

EET 438B
Sequential Control and Data Acquisition
Laboratory 2
Programming Advanced Control Structures in LabVIEW

Laboratory Learning Objectives

1. Use the Palettes of the LabVIEW programming environment to efficiently produce programs
2. Use the program debugging tools to monitor program execution and variable values.
3. Insert text-based program code into a LabVIEW program using formula nodes.
4. Utilize the Help menu to gain knowledge of instructions and find code examples.
5. Create programs using logic tests that implement on/off control.
6. Explain how a state machine operates using WHILE and CASE structures in LabVIEW.
7. Write a program that implements a control sequence utilizing logical testing and a simple state machine structure.

Theoretical Background

Computerized control systems need to make logical decision regarding input variables and execute complex control programs. On/off controllers, such as thermostats, use either hardware or software implementations of comparators. These devices compare an input value to a predetermined set point value and produce a logical output. The output is a voltage value for hardware implementations and a Boolean value for the software versions. Comparators types include inverting, non-inverting with or without hysteresis and window comparators. Comparators with hysteresis make output changes based on the previous output state along with the current input level.

A state machine is a software structure that can handle complex control problems by dividing the control actions into a sequence of events. The sequence of events may execute in a specific order based on the inputs to the system or randomly based on user interface actions. In either case, the state machine provides a powerful tool for implementing computerized control actions.

Comparator Circuits and Software Realizations

OP AMPS without feedback are one type of hardware-based comparator. There are also dedicated hardware devices that provide comparator functionality. Figure 1 shows inverting and non-inverting voltage comparators implemented using OP AMPs. The accompanying input/output diagrams show their functionality.

The inverting comparator implements the following logic in hardware.

$$\begin{aligned} V_{in} < V_{ref}, & V_O = +V_{sat} \\ V_{in} \geq V_{ref}, & V_O = -V_{sat} \end{aligned}$$

The open loop OP AMP circuit amplifies the voltage difference between the + and - terminals with a very high gain. When the value of V_{in} is just slightly less (1-2 mV) than V_{ref} , a positive

difference occurs and the amplifier output rises to its positive saturation voltage. This is approximately $+V_{cc}-1.5V$ for most common OP AMP circuits. When V_{in} is above V_{ref} then the output goes to the negative saturation voltage of $-V_{cc}+1.5V$ typically. The transition from positive to negative output levels signifies that the input level is above the reference level. A voltage divider or a Zener diode can produce this reference level in practice.

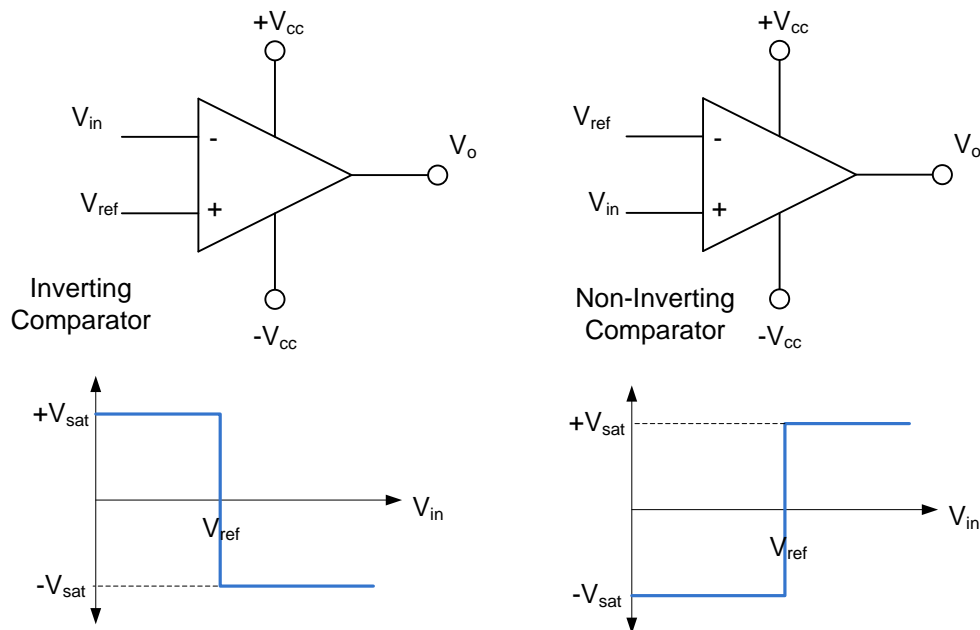


Figure 1. OP AMP Implementations of Voltage Comparators Along With Input/Output Responses.

The non-inverting comparator implements the logic below, which is the inverse of the inverting device.

$$\begin{aligned} V_{in} < V_{ref}, & \quad V_O = -V_{sat} \\ V_{in} \geq V_{ref}, & \quad V_O = +V_{sat} \end{aligned}$$

For this circuit, the transition from the lowest voltage to the highest indicates an input value greater than the reference value.

A logical test of the input using the comparison operators in LabVIEW produces that same result as the OP AMP hardware implementation. The LabVIEW program in Figures 2 and 3 show how to create non-inverting and inverting comparator action using a Boolean output and two numerical controls.

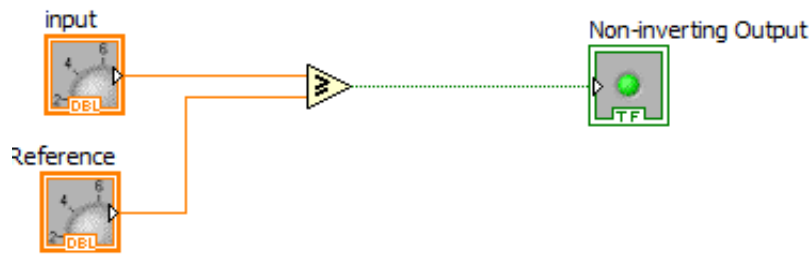


Figure 2, Non-Inverting Comparator In LabVIEW.

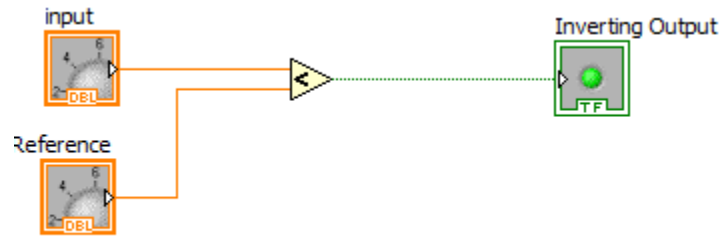


Figure 3, Inverting Comparator In LabVIEW.

Simple comparators can detect voltage levels, but are susceptible to spurious changes due to input noise or slight changes of input near the set point. Comparators with hysteresis reduce false output changes due to noise by adding a range of insensitivity to the comparator. This range is called deadband. In the hardware implementation using OP AMPs, the circuit feeds back a portion of the output to the reference input, which produces two transition points that depend on the input and the last output value. Figure 4 shows an inverting comparator with hysteresis. The values of resistors R_1 and R_2 determine the amount of dead band. The difference between the lower trip point voltage, V_{LTP} and the upper trip point voltage, V_{UTP} is the hysteresis or dead band of the comparator.

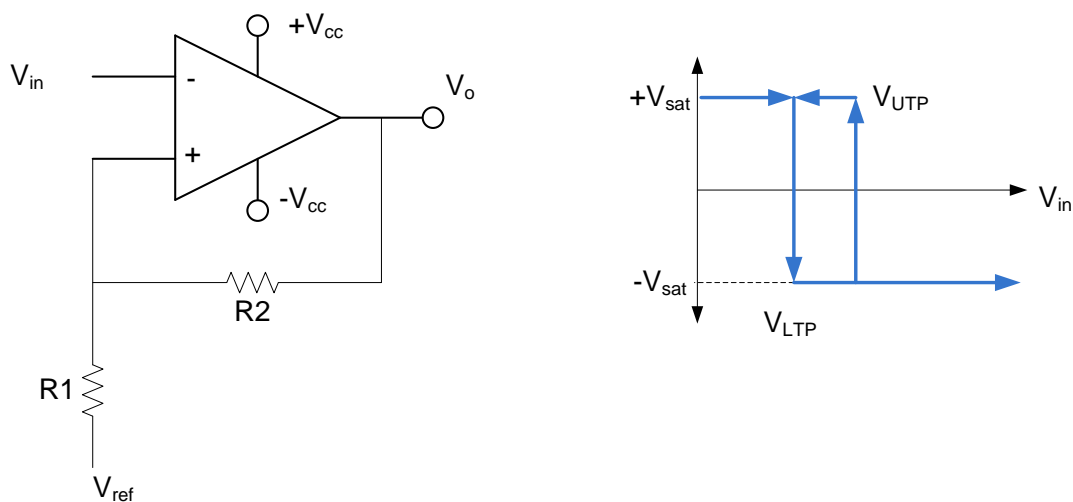


Figure 4. Inverting Comparator With Hysteresis,

Equations (1) and (2) give relationships for finding the values of V_{UTP} and V_{LTP} with specified resistor values.

$$V_{UTP} = +V_{sat} \left[\frac{R1}{R1 + R2} \right] + V_{ref} \left[\frac{R2}{R1 + R2} \right] \quad (1)$$

$$V_{LTP} = -V_{sat} \left[\frac{R1}{R1 + R2} \right] + V_{ref} \left[\frac{R2}{R1 + R2} \right] \quad (2)$$

Where: $+V_{sat}$ = positive saturation voltage of OP AMP,
 $-V_{sat}$ = negative saturation voltage of OP AMP, May be zero.
 V_{ref} = comparator reference voltage.

Taking the difference between the trip point voltage equations gives an equation for finding the hysteresis voltage. Equation (3) shows this formula.

$$V_h = 2 \cdot V_{sat} \left[\frac{R1}{R1 + R2} \right] \quad (3)$$

Where: V_h = the hysteresis voltage value ($V_{UTP} - V_{LTP}$)

Specifying a value of hysteresis voltage, a saturation voltage and a value of R1 in (3) determines the value of R2 in a design. These values can then be used in (1) along with a specified value of upper trip point voltage to find the reference voltage required.

Software realizations of comparators with hysteresis are possible but require the storage of the previous state of the comparator's output. A software implementation must satisfy the following logical expressions.

$$(X \leq UTP) \text{ AND } (Y' = \text{False}), \quad Y = \text{True}$$

$$(X \leq LTP) \text{ AND } (Y' = \text{True}), \quad Y = \text{False}$$

$$\text{and} \quad LTP = UTP - h$$

Where: X = the comparator input variable
 Y = the comparator output variable
 Y' = the previous output of the comparator
 UTP = comparator upper trip point value
 LTP = comparator lower trip point value
 h = value of hysteresis

A WHILE loop that uses a shift register will provide previous output state storage for the software implementation of the comparator with hysteresis.

Figure 5 shows the block diagram of an inverting comparator with hysteresis created in LabVIEW. The comparator tests the input, X_{in} , against the value of UTP modified by the hysteresis value, h . The comparator output, Y , is TRUE when the inequality is satisfied and the shift register saves it for further use. A TRUE output lights the indicator LED that is also the comparator output. On the next iteration of the loop, the previous value of Y is inverted and converted to an integer value of 0. This multiplies the value of hysteresis, which is then subtracted from the UTP value. As long as the comparator output remains TRUE, the UTP value remains unchanged since zero multiplies the value of hysteresis. When the input exceeds the UTP, the logical test $X \leq UTP$ returns a FALSE, extinguishing the indicating LED. On the next loop iteration, the shift register saved $Y=FALSE$ that is then inverted and converted to a numeric value of 1. This causes the hysteresis value to be subtracted from the UTP giving the LTP. The LTP is now used to test the X_{in} value. This implementation also displays the trip point value on the front panel so the change in trip point is visible while the program is running.

The comparator code inside the loop can be used in another program but the program must include a loop so the previous the value of output is available.

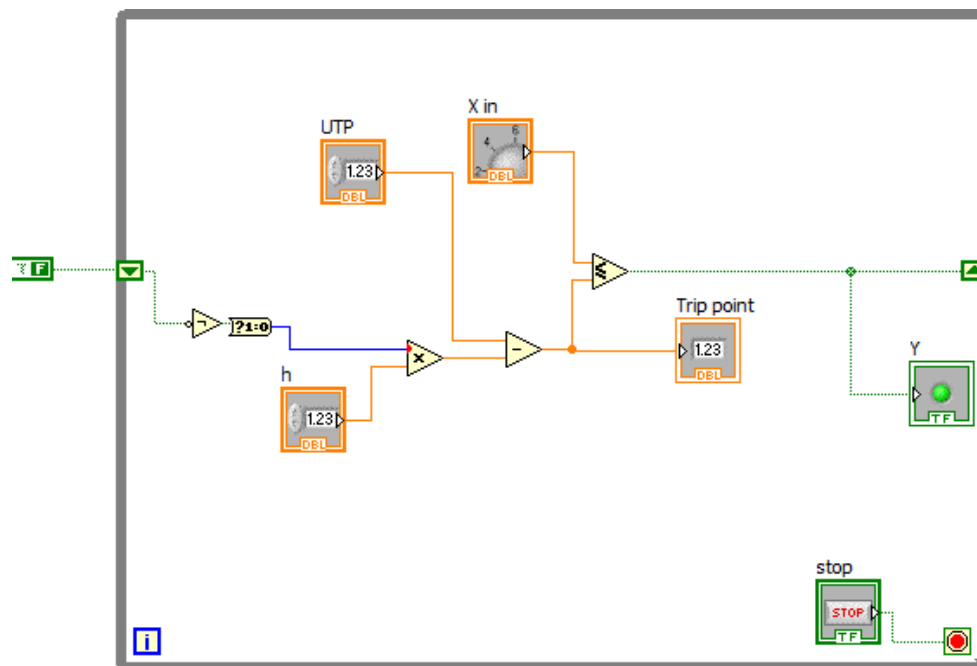


Figure 5. Block Diagram of an Inverting Comparator With Hysteresis.

Window comparators detect a range of input values between lower trip and upper trip points. It can also detect when an input is in one of three ranges of input defined by the lower and upper trip points. Figure 6 shows the input/output diagram and a schematic diagram for a window voltage comparator constructed using OP AMPs and a logic gate. The circuit consists of

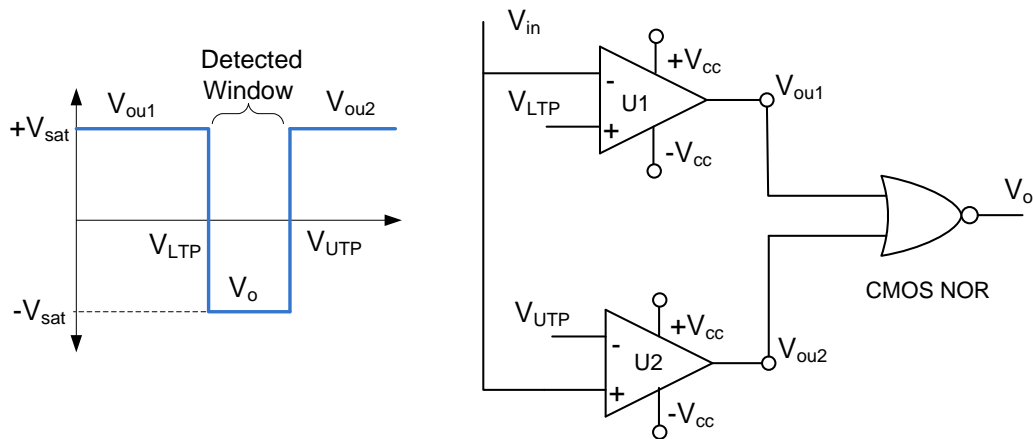


Figure 6. Window Comparator With Input/Output Diagram.

of U1, an inverting comparator, U2 a non-inverting comparator and a CMOS NOR gate. The comparators U1 and U2 share a common input voltage. The output of U1, V_{ou1} remains high as long as the input is less than V_{LTP} . The output of U2, V_{ou2} transitions from low to high at V_{UTP} . The range between the trip voltages of both devices defines the detection window. A CMOS NOR gate logically adds the outputs of U1 and U2 resulting in the input/output response. Table 1 shows the output logic for all possible combinations of comparator input.

Table 1 Window Comparator Logic

Condition	V_{ou1}	V_{ou2}	Desired Output	V_{ou1} Logic Level	V_{ou2} Logic Level	NOR Logic
$V_{LTP} < V_{in} < V_{UTP}$	$-V_{sat}$	$-V_{sat}$	V_o	0	0	1
$V_{in} \geq V_{UTP}$	$-V_{sat}$	$+V_{sat}$	0	0	1	0
$V_{in} \leq V_{LTP}$	$+V_{sat}$	$-V_{sat}$	0	1	0	0
Not Possible	$+V_{sat}$	$+V_{sat}$	0	1	1	0

Connecting outputs V_{ou1} and V_{ou2} to other circuits along with the overall output will allow the circuit to detect three ranges determined by the upper and lower trip point values.

A software implementation of the window comparator uses two logical tests to define the detection window. Figure 7 show the block diagram of a LabVIEW program that produces window comparator functionality. The $V_{in} \leq LTP$ test simulates the action of U1 in the hardware and the $V_{in} \geq UTP$ the action of U2. The logic function combines the two outputs to only register when the input is between the two set points. Having $LTP > UTP$ prevents proper operation. Figure 8 show the program that detects the three ranges defined by the two set points. Adding indicators to the greater/less than outputs displays when the input is within each range.

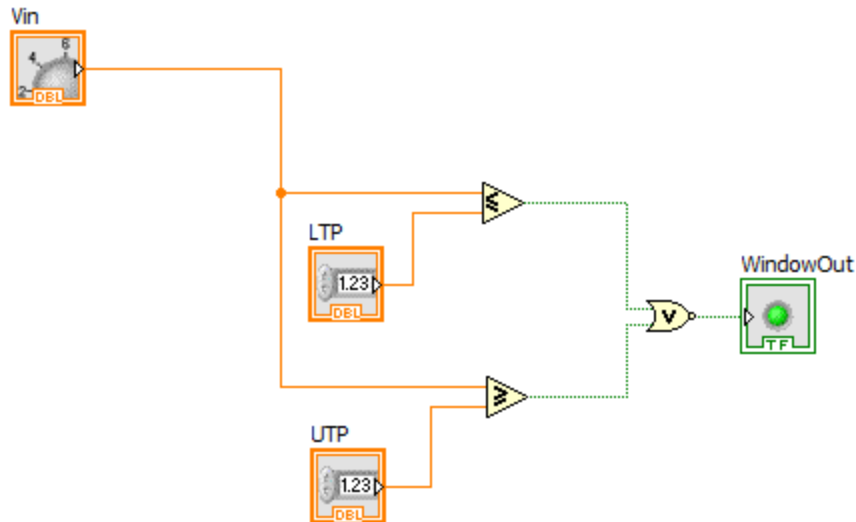


Figure 7. Window Comparator Logic Programmed In LabVIEW.

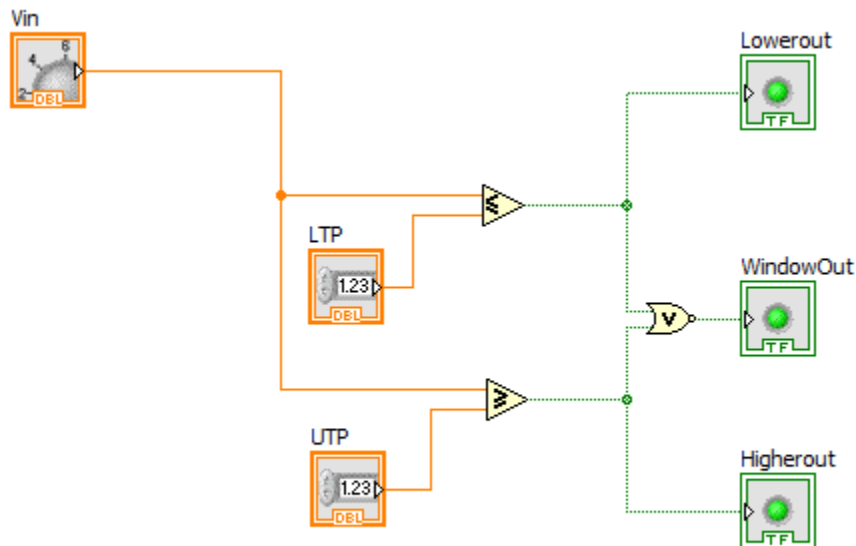


Figure 8. Window Comparator With Upper and Lower Range Indication Implemented Using LabVIEW.

State Machines

Complex sequential processes lend themselves to automation using computer-based control. These processes can be as simple as the control timer on a home washing machine or clothes dryer or be a multistage articulated robotic arm in an industrial plant. The systems both require control structures that produce a structured program. The state machine is a programming structure that is ideally suited for these applications and many others. This programming tool also finds application in programming user interfaces where a number of actions can occur randomly based on user interaction with the program.

The key characteristic of state machine systems is that the underlying process must have discernible steps or stages or the system must respond to events occurring randomly outside the program. This last case occurs in user interface design.

Three parts make up a typical state machine program structure. Figure 9 shows a block diagram of the major sections of a state machine program. A state machine program begins with an

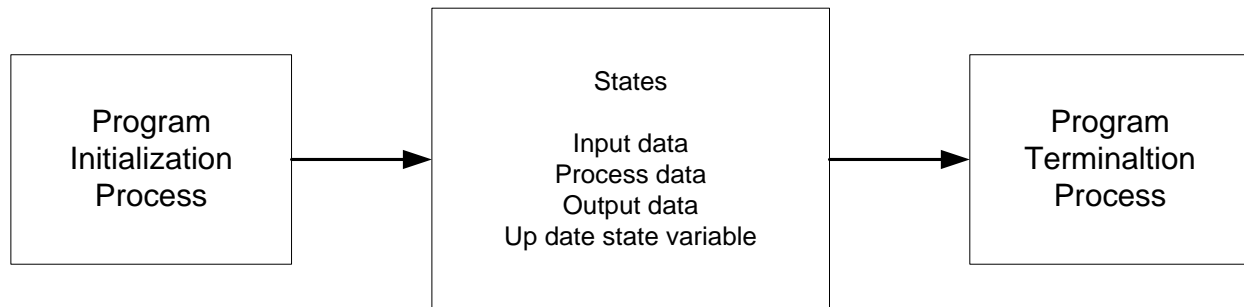


Figure 9. Overall Structure of a State Machine Program

initialization routine. This section of code defines the program variables and set the starting values. This code only executes once when the program starts. The most important action of this program section is that the state variables are set to their initial values correctly. A state variable is a variable that indicates the current and future states of the program. Changing the state variable moves the program through a sequence of steps in which different sections of code executes. The actions of a typical initialization routine include:

1. Initialing state variable
2. Setting all outputs to a starting value
3. Updating state variable to move to next state

The initialization process may include placing the program into an idle state to wait for user or system input before further action takes place. The program will execute the code in the idle state until other inputs enter the program. An idle state is not necessary in all state machine designs but is typically seen in user interface construction.

A state machine program enters into state processing after completing the initialization code. States are the individual stages or steps in a process. Lighting a pattern of LED in a specified order can be programmed using a state machine. Each LED pattern would represent a state in the program code. Programmers should define the states using meaningful descriptions before beginning the coding process. Using integers in the program to define the state is the simplest way to code the stages into the state machine. Figure 10 shows a graphical way to design a state machine prior to coding it. This figure is called a state diagram. The state diagram shows how the stages proceed through the process and how the variable, I, controls the interaction between the states.

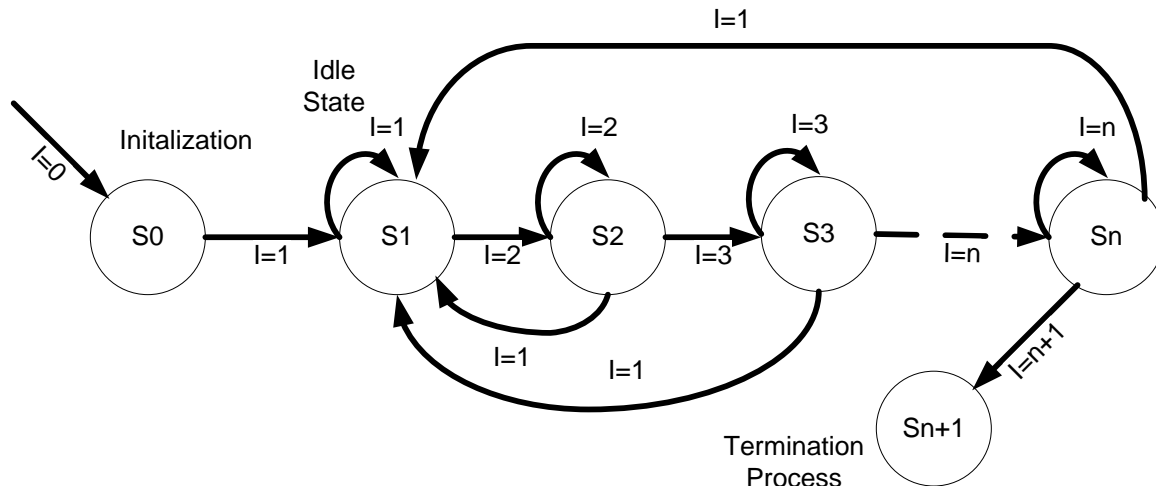


Figure 10. State Diagram Showing Integer State Variable and Interactions

The circles in the diagram represent the program states. These states map to descriptions of the state function. For example, the state S2 may represent a tank fill stage in a more complex process. The arrows indicate the direction process action takes with the state variable I representing state variable value necessary to enter the next program state. If the state variable remains unchanged the program repeats the code used to define the actions of a state. The loop back arrows in the diagram represent this activity. The process may require the program to jump between states. The arrows in the state diagram that point back to the idle state represent this action. The code in each state may include:

- 1.) Reading inputs
- 2.) Performing calculations based on inputs and current values
- 3.) Update output
- 4.) Modify state variable to continue program processing

The final section of the state machine is the termination routine. This code executes only once in the program before the program ends. The state S_{n+1} represents this action in the state diagram of Figure 10. The state machine code will update all output to final values, save all data and program conditions, and perform an orderly and safe process shutdown before ending program execution.

Figure 11 gives the pseudo code for the state diagram in Figure 10. The structure of a state machine uses two fundamental programming constructs: the WHILE loop and the CASE statement. Repeated IF-THEN statements can take the place of a CASE (SWITCH in C) statement. The basic function of the state machine code is to have only one section of instructions execute with the state variable being the determining variable. The pseudo code in Figure 11 shows the state machine implemented using IF-THEN statements with tests for each value of the state variable.

```

Begin

I=0 //initialize the state variable to enter state 0
// Place other initialization instructions here
I=1 // now increment the state variable and move to the while loop
WHILE I<>n+1 //repeat this loop until the program ends
//The following section contains state 1
IF I=1 THEN
//state 1 instructions go here
// read inputs, do processing, update state variable
// this code changes state variable based on some code test
IF TEST THEN
    I=2 //move to next state if TEST=true
ELSE
    I=1 //return to idle if TEST=false
ENDIF
END IF // This ends state 1 code section
IF I=2 THEN
//state 2 instructions go here
// read inputs, do processing, update state variable
// this code changes state variable based on some code test
IF TEST THEN
    I=3 //move to next state if TEST=true
ELSE
    I=1 //return to idle if TEST=false
ENDIF
END IF // This ends state 2 code section
IF I=3 THEN
//state 3 instructions go here
// read inputs, do processing, update state variable
// this code changes state variable based on some code test
IF TEST THEN
    I=n //move to next state if TEST=true
ELSE
    I=1 //return to idle if TEST=false
ENDIF
END IF // This ends state 3 code section
IF I=n THEN
//state n instructions go here
// read inputs, do processing, update state variable
// this code changes state variable based on some code test
IF TEST THEN
    I=n+1 //move to next state if TEST=true
ELSE
    I=1 //return to idle if TEST=false
ENDIF
END IF // This ends state n code section
END WHILE //this is the end of the WHILE loop
Save program data
Save program conditions
END //program

```

Figure 11. Pseudo Code for a State Machine

The pseudo code begins with initialization statements. The state variable, I, is updated to 1 as this section ends. The program then enters a WHILE loop with the exit condition set to the value of the termination state. The program will stay in this loop and execute the IF-THEN statements based on the value of the state variable. There is a section in each state's IF-THEN statement for the instruction necessary to process the action of the state followed by a test to determine if the code should jump back to the idle state or continue to the next stage of the process. If in the n-th state the test indicates the state variable should be updated to n+1, then the WHILE loop ends and the termination process code begins. This section of code handles saving program data and conditions and ends the program.

The CASE structure makes the code clear and simple. The CASE statement allows use of any data type, not just a Boolean as the decision variable. The simplest method is to let I be the CASE variable with the data type defined as integer.

Figure 12 shows a state machine template implemented in LabVIEW. This code makes use of the shift register to save the value of the state variable at each WHILE loop iteration. The CASE statement has five integer choices that represent the initialization process the active states and the termination process. The program initializes the state variable to zero. On the first iteration of the loop, the program enters state zero and would run any code place inside the CASE statement. Once completed the state variable is set to 1 and saved in the shift register. On the next iteration the CASE statement will enter the "1" case and execute the code written in the pane. Case 5 is the termination process. When the program enters this state, the code executes and saves all data and program variables. The case 5 code then increments the state variable to 6 which is a stop condition for the WHILE loop. A stop button can also end the program but would bypass the termination process code.

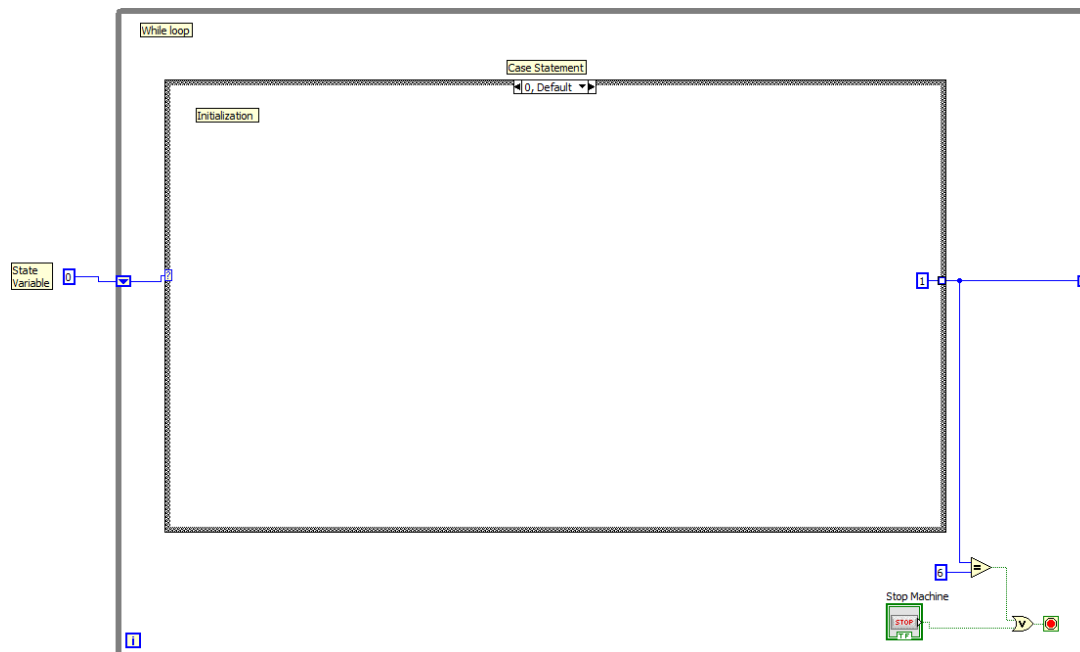


Figure 12. State Machine Template in LabVIEW.

Procedure

- 1.) Follow the link below to NI video training modules and review the these topic links:
Execution Structures and Passing Data Between Loop Iterations. <https://www.ni.com/getting-started/labview-basics/>. View the supplemental videos on creating state machines and using logic functions for control applications.
- 2.) Using the equations found in this lab handout, design an inverting comparator with hysteresis assuming that following values:

OP AMP: LM741

$V_{\text{sat}} = 13.5 \text{ V}$

$\pm V_{\text{cc}} = 15 \text{ V}$

$V_{\text{h}} = 0.75 \text{ V}$

$V_{\text{UTP}} = 5 \text{ V}$

$R1 = 1 \text{ k}\Omega$

Compute the values of $R2$ and V_{ref} to satisfy these design specifications. Show all calculations in a neat and organized way and save this work for later submission. Using a circuit simulator (e.g. Circuitmaker, Multisim, LTSpice) verify the circuit operation by running a transient analysis using a triangle wave. (See videos in D2L on using software) Verify the operation for both increasing and decreasing values of V_{in} to test the upper and lower trip point values. Printout the schematic and the circuit simulations and save them for later submission.

- 3.) Create a LabVIEW program that implements the hardware comparator designed in step 2 above. Verify operation with the lab instructor and printout both the front panel and block diagram for later submission.
- 4.) Create a state machine program in LabVIEW that will count 0-9 and repeat on a simulated seven segment display. There should be a 1 second delay between displayed counts. Pressing a stop button should end the counter operation and the program. Start the program using the template file located in D2L under Lab 2 materials. Figure A-1 in the appendix show a typical pinout for a hardware 7-segment display. Demonstrate the program operation to the lab instructor and print the front and back panels to document your work.
- 5.) Write a short report (3-4 pages double spaced) on how the state machine in 4 operates. Explain how each part of the program works and what each symbol of the block diagram represents.

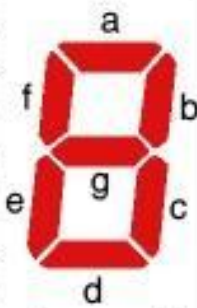
Lab 2 Assessment

Complete and submit the following items for grading and perform the listed actions to complete this laboratory assignment.

- 1.) Complete the online quiz over Lab 2 theory.
- 2.) Submit the calculations and simulations from the hysteresis comparator hardware design.
- 3.) Submit front and back panel printouts of LabVIEW software hysteresis comparator program
- 4.) Printouts of the front panel and block diagram of the counter program described in the procedure.
- 5.) Written program description of state machine as described in part 5 of the procedure.

Appendix

Outputs from the 4026 counter and display driver IC								
Count	a	b	c	d	e	f	g	h
0	●	●	●	●	●	●		●
1		●	●					●
2	●	●		●	●		●	●
3	●	●	●	●			●	●
4		●	●			●	●	●
5	●		●	●		●	●	
6	●		●	●	●	●	●	
7	●	●	●					
8	●	●	●	●	●	●	●	
9	●	●	●	●		●	●	



7-segment display

● = segment on. h is used to drive other counters.

Figure A-1. Configuration of a Hardware 7-Segment Display.