PlcLib(Arduino) User Guide

Author: W. Ditch

Last updated: 28th September, 2017

Software Version: *1.4*

1 Contents

1 Contents 2
2 Introduction 5
2.1 Software Development Methods 5
2.2 Supported Hardware 6
3 Installing the Software 8
3.1 First Time Installation 8
3.1.1 Loading and Running Your First Application 8
3.2 Updating to a New Version 9
4 Configuring the Hardware 10
5 Getting Started with Ladder Logic 11
5.1 Single Bit Input and Output 12
5.2 Performing Boolean Operations 13
6 Latching Outputs 18
6.1 Latch with Discrete Components 18
6.2 Using the Latch Command 19
6.3 Using the Set and Reset Commands 20
7 Edge Triggered Pulses 22
7.1 Creating an Edge Triggered Bistable 23
8 Inputting from a Keypad 25
8.1 Hardware Connections 25
8.2 Software 26
9 Using Time Delays 29
9.1 Producing a Turn-on Delay 29
9.2 Switch Debouncing 30
9.3 Creating a Turn-off Delay 31
9.4 Creating a Fixed Duration Pulse 32
10 Producing Repeating Waveforms 33
10.1 Manual Pulse Creation 33
10.2 Using the timerCycle() Command 34
11 Counting and Counters 36
11.1 Up Counter 36
11.2 Down Counter 37

11.3 Up/Down Counter 38
11.4 Debugging Counter-based Applications 39
12 Shifting and Rotating Binary Data 41
12.1 Creating and Using Shift Registers 41
12.2 Rotating Data 44
13 Working with Analogue Signals 47
13.1 Controlling LED Brightness using PWM 47
13.2 Controlling the Speed and Direction of a Motor 47
14 Position Control Using Servos 50
15 Comparing Analogue Values 51
15.1 Software-based Comparison of Analogue Values 51
15.2 A Simple Comparator Application 52
16 Instruction List Programming 54
17 Function Block Diagrams 55
17.1 Constructing Working Systems 55
17.2 Application 1: A Simple Alarm 56
17.3 Application 2: Alarm with Flashing 'Armed' LED 57
17.4 Creating User Defined Function Blocks 58
18 Sequential Function Charts 60
18.1 Creating Sequences 60
18.2 Branching and Converging 63
19 Developing Timed SFC-based Applications 65
19.1 Time-base Transitions 65
19.2 Application 1: Traffic Light Controller Application 67
19.3 Application 2: Running Light Display 69
20 Structured Text 74
20.1 Using Program Structures 74
21 Advanced Concepts 76
21.1 How the Software Works 76
21.2 Using Variables in Programs 77
21.2.1 Using the scanValue Variable 77
21.3 Working with Custom Variables 78
21.4 Using Variables with Complex Logic Circuits 79

22 Stack-based Storage and Logic ... 81
22.1 Block Logic Operations ... 82
23 Defining Custom IO Allocations ... 85
23.1 Preconfigured I/O Allocations ... 85
23.2 Case Study: Creating a Custom IO Allocation ... 85
24 Strengths and Limitations of the Software ... 88
25 Command Reference ... 89
25.1 General Configuration ... 89
25.2 Single Bit Digital Input / Output ... 89
25.3 Combinational Logic ... 90
25.4 Analogue Signal Input / Output ... 90
25.5 Comparing Analogue Signals ... 91
25.6 Latches ... 91
25.7 Timers ... 92
25.8 Edge Triggered Pulses ... 93
25.9 Counters ... 94
25.10 Shift Registers ... 95
25.11 Stack and Block Logic ... 97
26 Frequently Asked Questions ... 98
27 Revision History ... 100
28 Appendix A – Serial Monitor (Experimental Feature) ... 101
28.1 Introduction ... 101
28.2 Using the Serial Monitor ... 101
28.3 Technical Operation ... 104

2 Introduction

The *plcLib* library allows you to develop 'PLC-style' control-oriented software applications for the Arduino and compatibles.

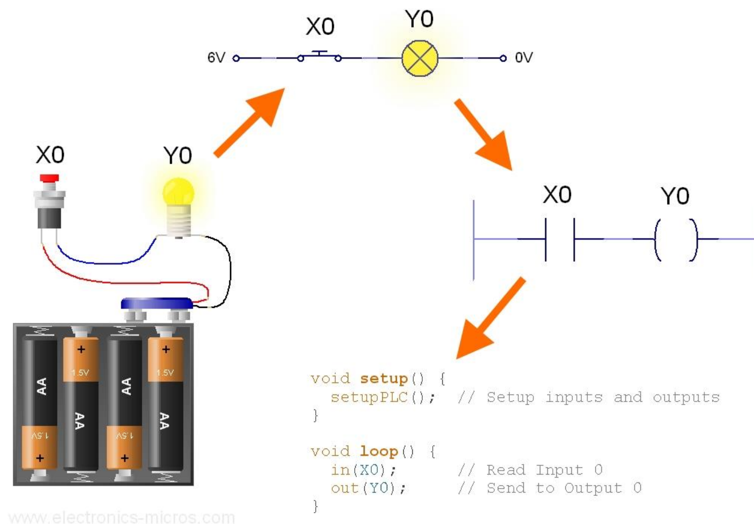


Figure 1. Converting a simple electrical circuit into a ladder diagram and then into a simple program.

A wide range of example sketches will also become available once you have installed the library. The above program – which is probably the simplest that does something useful – will then be available from the pull down menu of the Arduino IDE by selecting **File > Examples > plcLib > InputOutput > BareMinimum**.

2.1 Software Development Methods

In the most familiar design method, an electrical circuit containing elements such as switches and lamps is first represented in graphical form as a ladder diagram – which must then be transformed into a text-based Arduino program (or *sketch*), before being compiled and downloaded in the normal way. You can also describe a system as a block diagram (function block diagram), or a sequence based system (sequential function chart), all of which can be mixed with native C/C++ code (structured text), so you are free to use the design approach which best suits the problem.

The software is supplied as an installable Arduino library, which is *included* in the normal way at the start of your program. A range of text-based PLC-style commands then become available for use in your programs.

Note: *Unlike a modern commercial PLC, the software does not currently support graphical program entry, simulation, or run-time monitoring. Programs must be entered using the standard Arduino IDE. Diagrams provided in the User Guide are intended to illustrate the design process, and are linked to example sketches provided with the library.*

Software features supported by the current plcLib version include inputs (digital / analogue), outputs (digital / PWM / servo), Boolean logic operators, latches, timers, and repeating waveforms. Further

details are available in following sections of the *User Guide*, including download and installation instructions.

2.2 Supported Hardware

The plcLib library may be used with a variety of hardware, including *prototype boards*, *I/O shields* and even *Arduino compatible PLCs*.

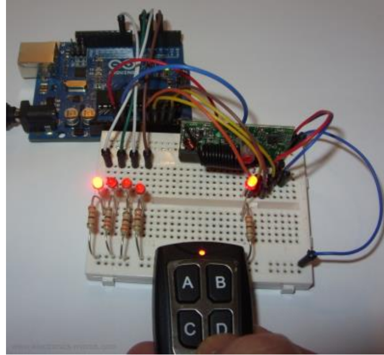


Figure 2. Using a prototype board to connect a remote-control receiver to an Arduino Uno.

The *Arduino Motor shield* is supported as standard, allowing speed and direction control of up to two DC motors.

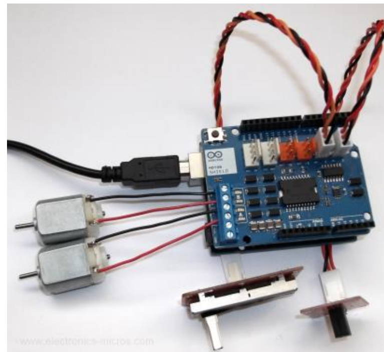


Figure 3. Controlling up to two motors with an Arduino Motor Shield.

Modular construction and experimentation systems such as *Grove* allow a wide range of input and output devices to be connected, without the need for prototype board wiring or soldering.

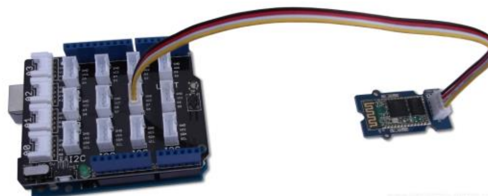


Figure 4. A Grove base shield and serial Bluetooth module connected to an Arduino Uno.

Version 1.2 or later of the software is supplied with a wide range of sample configurations for commonly available hardware, including Controllino and Industrial Shields Arduino compatible PLCs. Details of supported hardware is given in the Defining Custom IO Allocations section.

Figure 5. A typical Programmable Logic Controller (PLC). Image © istockphoto.com/Igor Mazej.

Custom *I/O configurations* may also be developed for other hardware, not supported as standard. This process is illustrated later, using the Velleman I/O shield for Arduino as a case study.

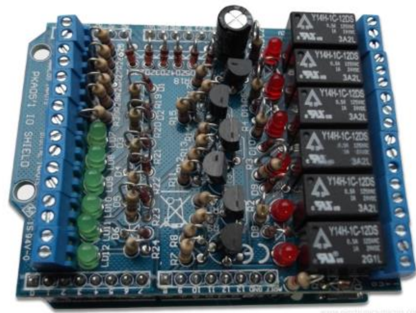


Figure 6. The Velleman Input/Output Shield offers a useful range of pre-configured inputs and outputs.

3 Installing the Software

Please follow the appropriate instructions below, depending on whether you are installing the software for the first time, or upgrading an existing version.

3.1 First Time Installation

The plcLib software is supplied as a ZIP file which can be downloaded and then added into your Arduino IDE in the normal way. With the software installed you'll be able to browse through the example programs, and create PLC-style sketches of your own.

Exact details of the installation process depends on the version of Arduino IDE installed. For Arduino 1.6.4 or later, select the **Sketch > Include Library > Add .Zip Library** option and browse to the Zip file downloaded earlier.

If you're using Version 1.0.5 of the Arduino IDE, then select **Sketch > Import Library... > Add Library...** from the pull-down menu, browse to find the previously downloaded Zip file.

The installation process is explained fully in this Arduino guide.

3.1.1 Loading and Running Your First Application

With the software installed, browse to the examples section and load the *Bare Minimum* sketch.

```
#include <plcLib.h>

/* Programmable Logic Controller Library for the Arduino and Compatibles

   Bare Minimum - Single bit digital input and output

   Connections:
   Input - switch connected to input X0 (Arduino pin A0)
   Output - LED connected to output Y0 (Arduino pin 3)

   Software and Documentation:
   https://github.com/wditch/plcLib

*/

void setup() {
  setupPLC();  // Setup inputs and outputs
}

void loop() {
  in(X0);      // Read Input 0
  out(Y0);     // Send to Output 0
}
```

Listing 1. *Bare Minimum (Source: File > Examples > plcLib > InputOutput > BareMinimum)*

Note: *All examples shown are available from the* ***File > Examples*** *section of the Arduino IDE.*

The above sketch reads a switch connected to input X0 (pin A0 on the Arduino) and sends the switch state to output Y0 (pin 3 on the Arduino). To see it working, you can either wire up a prototype board circuit, or use any of the range of supported hardware, as explained in the Configuring the Hardware section.

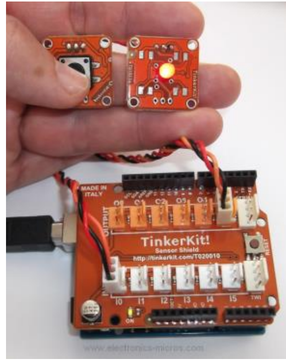


Figure 7. Testing the Bare Minimum program by using a modular experimentation system.

3.2 Updating to a New Version

New versions of the plcLib software are released periodically. The recommended upgrade procedure is to firstly remove the existing version of the library and then install the new one, as described below.

1. Identify the location of the library files on your computer, which will be in a *libraries* folder beneath the *Arduino Sketchbook* location. Exact details may vary but it will typically be something like *[My Documents]/Arduino/libraries*, with a *plcLib* folder existing beneath this if the software has been previously installed. You can check the path to the Sketchbook folder by selecting **File > Preferences** from the Arduino IDE.
2. Close the Arduino IDE, and then delete the plcLib folder identified above. (It is a good idea to make a backup copy of any existing files before deleting them – just in case you make a mistake.)
3. Reinstall the library as explained earlier.

The next section introduces the hardware arrangement used by the software.

4 Configuring the Hardware

A basic set of inputs and outputs is enabled by default. This default configuration is selected by firstly 'including' the plc library file (**#include <plcLib.h>**) and secondly by calling the **setupPLC()** function from within the *setup()* section of your sketch, as was seen in the example sketch of Listing 1.

At a minimum, the software defines four inputs *X0*, *X1*, *X2* and *X3* (analogue inputs *A0–A3*) and four outputs *Y0*, *Y1*, *Y2* and *Y3* (pins 3, 5, 6, and 9).

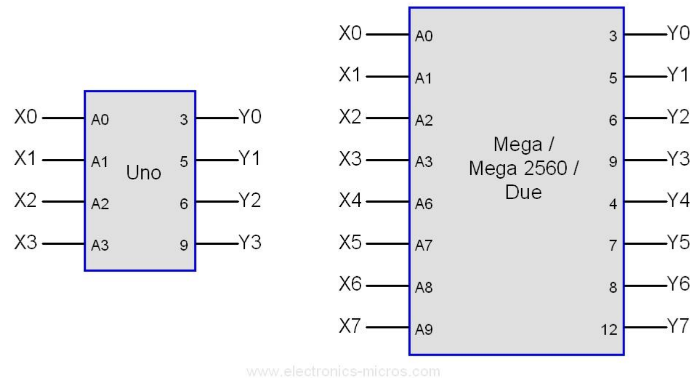


Figure 8. Default Input / Output allocations for common Arduino boards

Additional pins are allocated for larger Arduino boards (*Mega*, *Mega 2560* or *Due*), as shown at the right, giving 8 inputs and 8 outputs in total.

The main features of this hardware layout are explained below:

- Inputs are capable of reading either digital or analogue values.
- Outputs can produce either digital, PWM or servo values.
- Arduino pins with duplicate functions have been avoided wherever possible, to minimise hardware conflicts.
- Data directions of inputs and outputs are automatically configured and outputs are initially disabled (based on the assumption that 0='off' and 1 = 'on')

Note: *Most of the supplied example files make use of the standard I/O configuration.*

If this default arrangement proves unsuitable, then a number of alternative custom hardware setups are available – or you may prefer to create your own. Please see the Defining Custom IO Allocations section for more details.

The next section introduces the use of a *ladder diagram* to describe the arrangement and operation of a simple system, and its conversion into *ladder logic* based program. Ladder logic concepts are further developed in subsequent sections of the plcLib User Guide.

5 Getting Started with Ladder Logic

The PLC design method often starts with an electrical circuit, or block diagram, which is then redrawn as a *ladder diagram*, and then converted into an Arduino sketch before being compiled and downloaded in the normal way. The figure below illustrates the process.

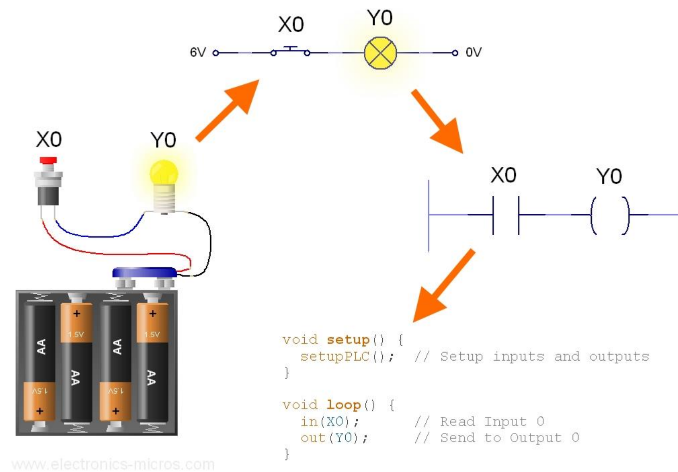


Figure 9. Converting an electrical circuit into a ladder diagram and then into a ladder logic program.

The name 'ladder diagram' comes from the superficial resemblance to a physical ladder, with vertical power rails at each side, and horizontal circuit branches called *rungs* connected between the rails. More complex ladder diagrams have a series of rungs, each of which represents a separate circuit.

Ladder diagrams are an adaptation of an earlier technology called *relay logic*, in which switches and relays are used to control industrial circuits. A simple relay logic circuit is shown below.

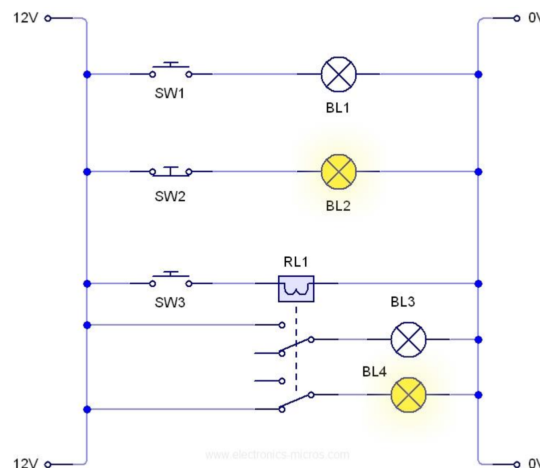


Figure 10. A simple Relay Logic circuit.

Notice the positive power rail at the left, and negative at the right. Switches SW1 and SW2 are push-to-make and push-to-break types, causing their associated lamps to be lit when the switches are pressed or released, respectively. Switch SW3 is connected to relay coil RL1 and the changeover relay contacts are

then linked to lamps BL3 and BL4. The relay contacts are arranged so that only one lamp is lit at any time.

Any program which make use of the plcLib software library must be entered as a standard Arduino text-based *sketch*, which PLC programmers may refer to as *instruction list* programming. With practice, the process of converting a ladder diagram into an Arduino sketch becomes relatively straightforward. A number of alternatives to the ladder logic design approach are available, including function block programming, sequential function charts and structured text, each of which is discussed separately.

Note: *These five programming methods (ladder diagram, instruction list, function block, sequential function chart and structured text) are defined by the Part 3 of the IEC 61131 standard (IEC 61131-3, also known in the UK as BS EN 61131-3), which deals with programming languages for programmable controllers.*

5.1 Single Bit Input and Output

The plcLib software allows single bit inputs inputs and outputs to be controlled in either *normally off* or *normally on* forms. A normally on input is equivalent to a *push-to-make* switch, and is represented by a pair of vertical lines in the ladder diagram. A *push-to-break* switch gives a normally closed connection and this is shown by adding a diagonal line between the vertical contacts. A similar arrangement is used for outputs which are shown either as a pair of curved lines (or even a complete circle), with a diagonal line added for an inverted output. The ladder diagram below shows the input and output of values in normal and inverted forms.

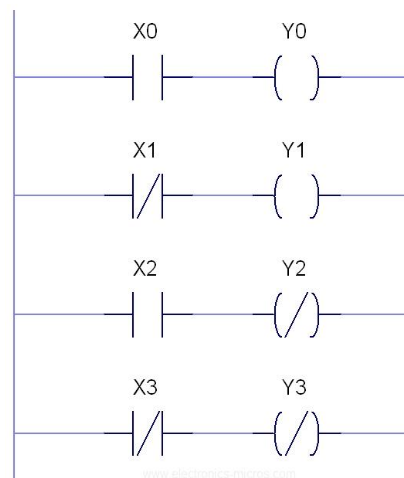


Figure 11. A ladder diagram showing methods of reading inputs and controlling outputs.

The ladder diagram may be easily converted to a text-based sketch, as shown below.

```
#include <plcLib.h>

/* Programmable Logic Controller Library for the Arduino and Compatibles

   Digital Input / Output - Single bit I/O in normal and inverted forms

   Connections:
   Input - switch connected to input X0 (Arduino pin A0)
   Input - switch connected to input X1 (Arduino pin A1)
```

```
   Input - switch connected to input X2 (Arduino pin A2)
   Input - switch connected to input X3 (Arduino pin A3)
   Output - LED connected to output Y0 (Arduino pin 3)
   Output - LED connected to output Y1 (Arduino pin 5)
   Output - LED connected to output Y2 (Arduino pin 6)
   Output - LED connected to output Y3 (Arduino pin 9)

   Software and Documentation:
   https://github.com/wditch/plcLib

*/

void setup() {
  setupPLC();  // Setup inputs and outputs
}

void loop() {
  in(X0);      // Read Input 0
  out(Y0);     // Send to Output 0

  inNot(X1);   // Read Input 1 (inverted)
  out(Y1);     // Send to Output 1

  in(X2);      // Read Input 2
  outNot(Y2);  // Send to Output 2 (inverted)

  inNot(X3);   // Read Input 3 (inverted) and send to Output 3 (inverted)
  outNot(Y3);  // (The double negative cancels out)
}
```

Listing 2. *Digital Input / Output (Source: File > Examples > plcLib > inputOutput > DigitalInputOutput)*

The PLC software repeatedly calculates each rung of the ladder diagram in sequence – from left to right and top to bottom – by 'scanning' the inputs, performing calculations, and outputting the results; a process known as the *scan cycle*. Each rung of the ladder diagram is effectively a separate task, and the computer shares its processing power between these tasks in a repeating sequence. The high processing speed makes it appear that the PLC is performing several activities at the same time.

Inputs may be connected in series or parallel to create simple Boolean Logic (combinational logic) functions, as discussed in the next section.

5.2 Performing Boolean Operations

Boolean logic functions such as AND and OR may be achieved using series / parallel arrangements of switch contacts. For example, two switches in series will give an AND function, since both switches must be closed to complete the circuit. Similarly an OR function may be achieved by two switches connected in parallel, as closing one or more switches will allow power to flow to the next stage.

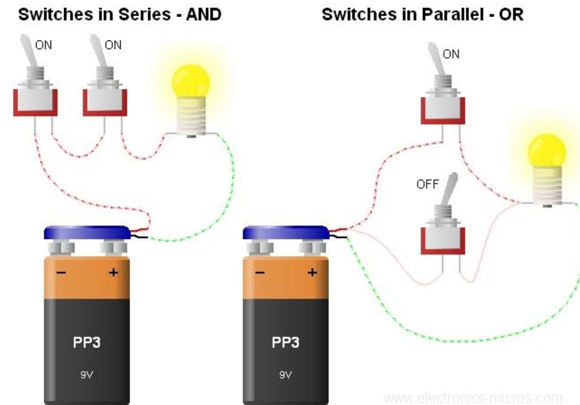


Figure 12. Switches in series and parallel perform logical AND and OR functions.

The basic Boolean logic functions AND, OR, XOR and NOT may be represented in ladder diagram form by using series / parallel combinations of input switches and output contacts, as shown below.

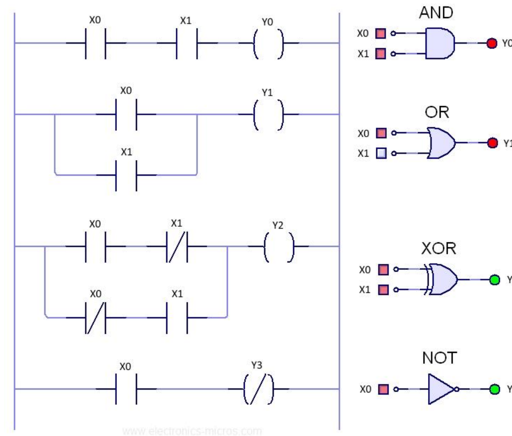


Figure 13. Boolean logic ladder logic functions and their equivalent logic functions.

This ladder diagram arrangement may be easily coded, as shown in the following sketch.

```
#include <plcLib.h>

/* Programmable Logic Controller Library for the Arduino and Compatibles

   AND, OR, XOR and Not - Basic Boolean Logic Functions

   Connections:
   Input - switch connected to input X0 (Arduino pin A0)
   Input - switch connected to input X1 (Arduino pin A1)
   Output - ANDed Output - LED connected to output Y0 (Arduino pin 3)
   Output - ORed Output - LED connected to output Y1 (Arduino pin 5)
   Output - XORed Output - LED connected to output Y2 (Arduino pin 6)
   Output - Inverted Output - LED connected to output Y3 (Arduino pin 9)

   Software and Documentation:
   https://github.com/wditch/plcLib

*/

void setup() {
  setupPLC();  // Setup inputs and outputs
}
```

```
void loop() {
  in(X0);       // Read Input 0
  andBit(X1);   // AND with Input 1
  out(Y0);      // Send result to Output 0

  in(X0);       // Read Input 0
  orBit(X1);    // OR with Input 1
  out(Y1);      // Send result to Output 1

  in(X0);       // Read Input 0
  xorBit(X1);   // XOR with Input 1
  out(Y2);      // Send result to Output 2

  in(X0);       // Read Input 0
  outNot(Y3);   // Send inverted result to Output 3
}
```

Listing 3. *AND, OR, XOR and Not functions (Source: File > Examples > plcLib > Logic > AndOrXorNot*

If active-low outputs are required then NAND, NOR and XNOR equivalent functions may be created in ladder logic.

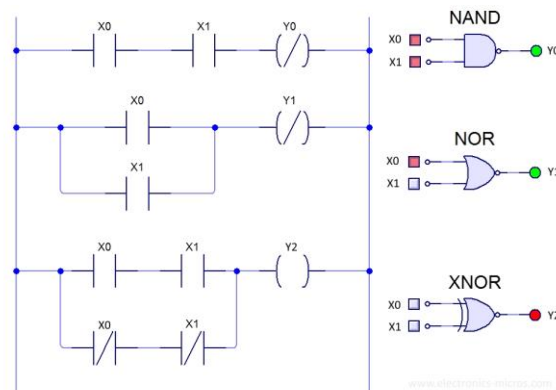


Figure 14. NAND, NOR and XNOR ladder logic circuits and their equivalent logic functions.

Coding of these functions is achieved by replacing the **out()** instructions of the previous example with the negative logic **outNot()** equivalent, as shown below.

```
#include <plcLib.h>

/* Programmable Logic Controller Library for the Arduino and Compatibles

   NAND, NOR, and XNOR - Boolean Logic functions with inverted outputs

   Connections:
   Input - switch connected to input X0 (Arduino pin A0)
   Input - switch connected to input X1 (Arduino pin A1)
   Output - NAND Output - LED connected to output Y0 (Arduino pin 3)
   Output - NOR Output - LED connected to output Y1 (Arduino pin 5)
   Output - XNOR Output - LED connected to output Y2 (Arduino pin 6)

   Software and Documentation:
   https://github.com/wditch/plcLib
*/

void setup() {
  setupPLC();   // Setup inputs and outputs
}

void loop() {
                // NAND
```

```
  in(X0);       // Read Input 0
  andBit(X1);   // AND with Input 1
  outNot(Y0);   // Send result to Output 0 (inverted)

                // NOR
  in(X0);       // Read Input 0
  orBit(X1);    // OR with Input 1
  outNot(Y1);   // Send result to Output 1 (inverted)

                // XNOR
  in(X0);       // Read Input 0
  xorBit(X1);   // XOR with Input 1
  outNot(Y2);   // Send result to Output 2 (inverted)
}
```

Listing 4. *NAND, NOR, and XNOR functions. (Source: File > Examples > plcLib > Logic > NandNorXnor)*

Logical operations may also be performed with one or more of the inputs inverted, as seen here.

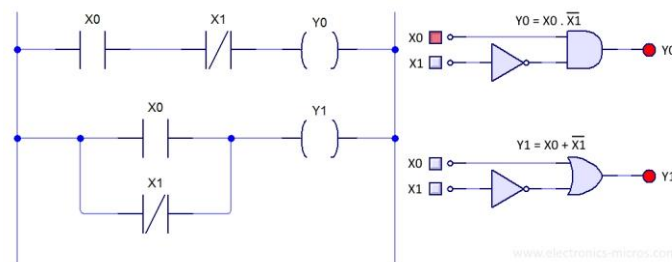


Figure 15. Performing Boolean operations involving inverted input signals.

An equivalent sketch is shown below.

```
#include <plcLib.h>

/* Programmable Logic Controller Library for the Arduino and Compatibles

   Inverted Input Logic - Boolean logic operations using inverted inputs
                          (equivalent to normally closed input switches)

   Connections:
   Input - switch connected to input X0 (Arduino pin A0)
   Input - switch connected to input X1 (Arduino pin A1)
   Output - ANDed output - LED connected to output Y0 (Arduino pin 3)
   Output - ORed output - LED connected to output Y1 (Arduino pin 5)

   Software and Documentation:
   https://github.com/wditch/plcLib
*/

void setup() {
  setupPLC();         // Setup inputs and outputs
}

void loop() {
  in(X0);             // Read Input 0
  andNotBit(X1);      // AND with Input 1 (inverted)
  out(Y0);            // Send result to Output 0

  in(X0);             // Read Input 0
  orNotBit(X1);       // OR with Input 1 (inverted)
  out(Y1);            // Send result to Output 1
}
```

Listing 5. *Inverted input logic (Source: File > Examples > plcLib > Logic > InvertedInputLogic)*

It is also possible to perform logical operations involving the state of output contacts, which in effect applies feedback from outputs to inputs. This of course is the basis of *sequential logic*, the simplest of which is the *Set-Reset latch* – to be considered next.

6 Latching Outputs

You can *latch* a momentary input, causing it to remain active (or *set*) until it needs to be cancelled (or *reset*). Three different approaches are available, as described in the following sections.

6.1 Latch with Discrete Components

The Set-Reset latch is one of the mainstays of electronics, and is the simplest of *sequential logic* circuits – most often seen as a pair of cross connected NAND or NOR gates. It is also quite easy to create a self latching circuit using just a *relay* (an electromagnetically operated switch) and a few other components, as seen in the following relay logic circuit.

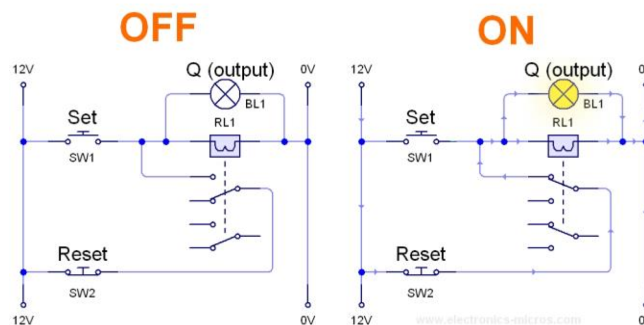


Figure 16. Discrete components have been used here to create a self-latching circuit.

Examining the circuit at the left, firstly notice that the two switches are wired in parallel, which is of course a logical OR arrangement. However, to begin with, current does not flow through either path! The upper branch is off, as the push-to-make switch is not pressed. The lower switch is a push-to-break type, but this path is also off, being blocked by the relay contacts which are in the off position (down).

A momentary press of the *Set* input switch allows power to reach the relay coil which activates, moving the relay contacts to their On position (up). Current now flows through the lower branch, so the relay remains enabled, even when the Set input is released. The relay remains active until the *Reset* input switch is pressed, hence disconnecting the relay coil and returning the relay contacts to their Off position.

Note: *Set Reset latches often have normal and inverted outputs. This function may easily be added, if required, by wiring a second lamp via the unused set of switch contacts, but with opposite polarity to the main output.*

An equivalent Arduino sketch is shown below.

```
#include <plcLib.h>

/* Programmable Logic Controller Library for the Arduino and Compatibles

   Latch using Discrete Components - Self latching circuit with Q and Not Q outputs

   Connections:
   Input - Set - switch connected to input X0 (Arduino pin A0)
   Input - Reset - switch connected to input X1 (Arduino pin A1)
   Output - Q - LED connected to output Y0 (Arduino pin 3)
   Output - NotQ - LED connected to output Y1 (Arduino pin 5)
```

```
   Software and Documentation:
   https://github.com/wditch/plcLib

*/

void setup() {
  setupPLC();              // Setup inputs and outputs
}

void loop() {
  in(X0);            // Read switch connected to Input 0 (Set)
  orBit(Y0);         // Self latch using Output 0 (Q)
  andNotBit(X1);     // Reset latch using Input 1 (Reset)
  out(Y0);           // Output to Output 0 (Q)

  in(Y0);            // Read Q output
  outNot(Y1);        // Produce inverted output on Output 1 (Not Q)
}
```

Listing 6. *Latch using Discrete Components (Source: File > Examples > plcLib > Latch > LatchDiscreteComponents)*

The above self-latching circuit works well, but is rather verbose. A simplified version is shown in the next section.

6.2 Using the Latch Command

The **latch()** command is functionally identical to the self-latching arrangement seen above, but requires a minimum of two lines of code. An equivalent function block diagram representation is shown below, the first variant having normal and inverted outputs, and the second with only a single normal output.

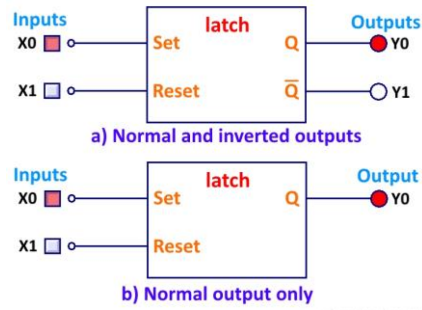


Figure 17. Set reset latch symbols, shown both with and without an inverted output.

The following sketch shows a latch with both normal and inverted outputs, although the latter two lines used to generate the inverted output are optional.

```
#include <plcLib.h>

/* Programmable Logic Controller Library for the Arduino and Compatibles

   Latch Command - Set Reset latch with Q and NotQ outputs, based on the 'latch' command

   Connections:
   Input - Set - switch connected to input X0 (Arduino pin A0)
   Input - Reset - switch connected to input X1 (Arduino pin A1)
   Output - Q - LED connected to output Y0 (Arduino pin 3)
   Output - NotQ - LED connected to output Y1 (Arduino pin 5)

   Software and Documentation:
   https://github.com/wditch/plcLib
```

```
*/

void setup() {
  setupPLC();           // Setup inputs and outputs
}

void loop() {
  in(X0);               // Read switch connected to Input 0 (Set input)
  latch(Y0, X1);        // Latch, Q = Output 0, Reset = Input 1

  in(Y0);               // Read Q output and generate NotQ on Output 1
  outNot(Y1);           // (These two lines are optional)
}
```

Listing 7. *Latch Command (Source: File > Examples > plcLib > Latch > LatchCommand)*

Notice that the *Set* input to the latch is taken from the preceding value from the same 'rung' of the ladder diagram (reading input *X0* in this case). The command takes two arguments which are the *Q* output and the *Reset* input respectively.

6.3 Using the Set and Reset Commands

An alternative method of creating a latched output is to use separate **set()** and **reset()** commands, as shown in the following sketch.

```
#include <plcLib.h>

/* Programmable Logic Controller Library for the Arduino and Compatibles

   Using Set and Reset commands to create a Set-Reset Latch

   Connections:
   Input - Set - switch connected to input X0 (Arduino pin A0)
   Input - Reset - switch connected to input X1 (Arduino pin A1)
   Output - Q - LED connected to output Y0 (Arduino pin 3)
   Output - NotQ - LED connected to output Y1 (Arduino pin 5)

   Software and Documentation:
   https://github.com/wditch/plcLib

*/

void setup() {
  setupPLC();           // Setup inputs and outputs
}

void loop() {
  in(X0);               // Read switch connected to Input 0 (Set input)
  set(Y0);              // Set Y0 to 1 if X0 = 1, leave Y0 unaltered otherwise

  in(X1);               // Read switch connected to X1
  reset(Y0);            // Clear Y0 to 0 if X1 = 1, leave Y0 unaltered otherwise
}
```

Listing 8. *Set and Reset commands (Source: File > Examples > plcLib > Latch > SetResetCommands)*

The set() and reset() commands are available in Version 0.7 or later of the plcLib library.

A letter is added to the standard output symbol indicating whether the output is a 'set' or 'reset' type, as shown in the equivalent ladder diagram.

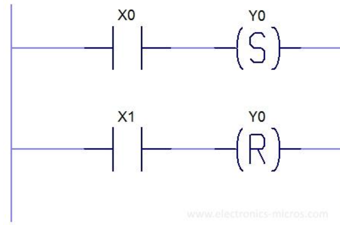


Figure 18. A latch implemented using Set and Reset outputs.

This method allows separate logic to control the enabling and disabling of a latched output, which is often convenient. A sequential function chart is a typical application, in which these commands may be used to control the transitions between steps in a sequence.

7 Edge Triggered Pulses

Version 1.1 of the plcLib software introduces the ability to generate edge triggered pulses, the output of which are active for a single scan cycle only.

This can be useful in a range of scenarios, such as triggering an output once only, ensuring an output lasts for a shorter duration than an input, or preventing a latch-based system from being locked in one position due to a faulty external switch.

A simple edge triggered system is shown below.

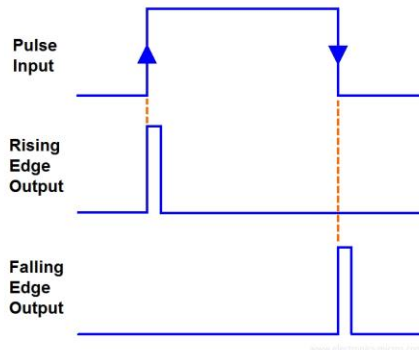


Figure 19. Brief output pulses may be generated from the rising or falling edges of an input waveform.

This may be represented as shown in the following ladder diagram, with rising or falling edges on input *X0* being used to momentarily set outputs *Y0* and *Y1*, respectively.

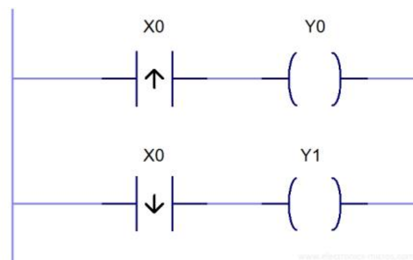


Figure 20. Output pulses on Y0 and Y1 are generated by rising or falling edges of input waveform X0.

An equivalent sketch is shown below.

```
#include <plcLib.h>

/* Programmable Logic Controller Library for the Arduino and Compatibles

   Generate rising-edge and falling-edge pulses using a digital input

   Connections:
   Clock input - switch connected to input X0 (Arduino pin A0)
   Rising-edge output - LED connected to output Y0 (Arduino pin 3)
   Falling-edge output - LED connected to output Y1 (Arduino pin 5)

   Software and Documentation:
   https://github.com/wditch/plcLib

*/

Pulse pulse1;              // Create a pulse object called 'pulse1'
```

```
void setup() {
  setupPLC();                 // Setup inputs and outputs
}

void loop() {
  in(X0);                     // Read switch connected to Input 0 and
  pulse1.inClock();           // connect it to the clock input

  pulse1.rising();            // Read rising edge
  out(Y0);                    // Output rising edge for 1 scan cycle only

  pulse1.falling();           // Read falling edge
  out(Y1);                    // Output falling edge for 1 scan cycle only

  delay(50);                  // Slow down scan cycle to enable viewing pulses
                              // (remove this delay in the final code)
}
```

Listing 9. *Generating rising- and falling-edge pulses (Source: File > Examples > plcLib > Pulse > PulseInput)*

Examining the above listing, a pulse object called 'pulse1' is first created. External input *X0* is then connected to the clock input of the previously created pulse. Finally, rising and falling edge signals are used to drive outputs *Y0* and *Y1* respectively.

Output pulses last for a single scan cycle only, so a 50 millisecond time delay has been added to the above sketch to 'stretch' the output pulses long enough to be visible when connected to LEDs. This delay should be removed after testing to avoid slowing down the scan cycle.

7.1 Creating an Edge Triggered Bistable

A simple set reset latch may be created using several different methods, as demonstrated in the Latching Outputs section. Given the potential for faulty input switches mentioned earlier, it is useful to consider what would happen to a latch-based circuit if both the *Set* and *Reset* inputs were simultaneously activated, or if one of the external input switches became stuck in the 'On' position?

These potential issues may to a large extent be overcome by using edge triggered signals to drive the latch inputs, as shown in the following ladder diagram.

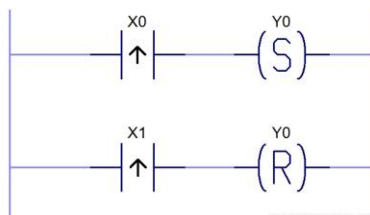


Figure 21. Using edge triggered inputs to set or reset an output.

An equivalent sketch is shown below.

```
#include <plcLib.h>

/* Programmable Logic Controller Library for the Arduino and Compatibles

   Set or reset an output using edge-triggered inputs

   Connections:
```

```
   Set input - switch connected to input X0 (Arduino pin A0)
   Reset input - switch connected to input X1 (Arduino pin A1)
   Latched output - LED connected to output Y0 (Arduino pin 3)

   Software and Documentation:
   https://github.com/wditch/plcLib

*/

Pulse setPulse;              // Create a pulse object to set the latch
Pulse resetPulse;            // Create a pulse object to reset the latch

void setup() {
  setupPLC();               // Setup inputs and outputs
}

void loop() {
  in(X0);                   // Read switch connected to Input 0 and
  setPulse.inClock();       // connect it to the set pulse clock input

  in(X1);                   // Read switch connected to Input 1 and
  resetPulse.inClock();     // connect it to reset pulse clock input

  setPulse.rising();        // Read rising edge of X0
  set(Y0);                  // Set Y0 to 1 on using rising edge of X0

  resetPulse.rising();      // Read rising edge of X1
  reset(Y0);                // Reset Y0 to 0 on using rising edge of X1
}
```

Listing 10. *Set or reset an output using edge-triggered inputs (Source: File > Examples > plcLib > Pulse > SetResetEdge)*

8 Inputting from a Keypad

A matrix keypad may be used as an input device, allowing PLC outputs to be turned on or off when particular keys are pressed. To illustrate the concept, a standard membrane keypad will be used, together with a freely downloadable *keypad library*, to drive a series of latched LED outputs.

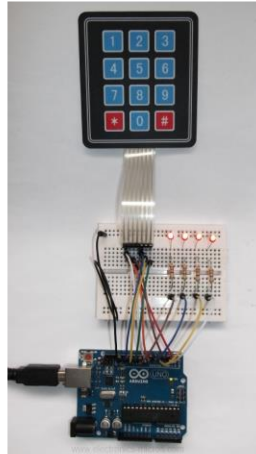


Figure 22. A test circuit and sketch, with keypad inputs and latched LED outputs.

Note: *Before trying the examples, you'll need to download and install the* [Arduino Keypad library](). *The keyboard example files are available with Version 0.6 or later of the* plcLib software*.*

8.1 Hardware Connections

We'll be using an Arduino Uno main board for the examples, which the plcLib software defines as having four inputs (X0–X3) and four outputs (Y0–Y3). The chosen keypad is an Adafruit membrane keypad with 12 keys, arranged as a matrix with four rows and three columns, hence requiring a further seven IO pins in total. The following image shows the selected *input / output allocation* in diagrammatic form.

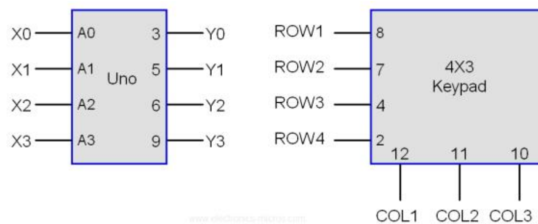


Figure 23. IO allocation for the Arduino Uno and membrane keypad.

A possible prototype board wiring arrangement is shown below (drawn using [Fritzing]()).

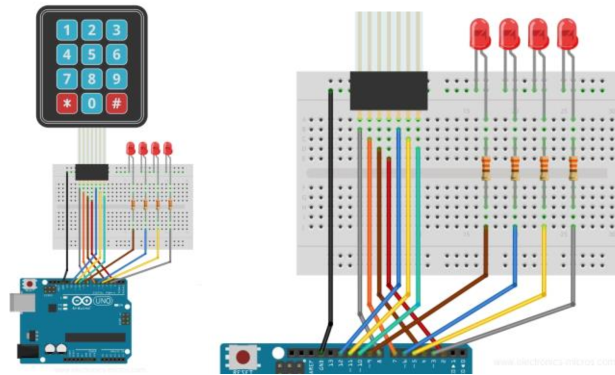


Figure 25. Keypad and LED connections (left), prototype board wiring detail (right).

8.2 Software

The following listing takes a fairly standard keypad input program, and uses a custom **latchKey()** function to enable or disable an output bit based on momentary presses of the allocated Set and Reset keys.

```
/* Programmable Logic Controller Library for the Arduino and Compatibles

   Matrix Keypad with On/Off Control of Latched LED Outputs

   Arduino Uno Pin Connections:
   Input X0 - unused (Arduino pin A0)
   Input X1 - unused (Arduino pin A1)
   Input X2 - unused (Arduino pin A2)
   Input X3 - unused (Arduino pin A3)
   Output - LED connected to output Y0 (Arduino pin 3)
   Output - LED connected to output Y1 (Arduino pin 5)
   Output - LED connected to output Y2 (Arduino pin 6)
   Output - LED connected to output Y3 (Arduino pin 9)

   4X3 Matrix Keypad Connections:
   ROW1 - Arduino pin 8
   ROW2 - Arduino pin 7
   ROW3 - Arduino pin 4
   ROW4 - Arduino pin 2
   COL1 - Arduino pin 12
   COL2 - Arduino pin 11
   COL3 - Arduino pin 10

   Software and Documentation:
   https://github.com/wditch/plcLib

*/

#include <Keypad.h>  // Include the Keypad library
#include <plcLib.h>  // Include the plcLib library

char keyPress = 0;   // Holds the currently pressed key value (if any)
const byte ROWS = 4; // Keypad has four rows
const byte COLS = 3; // Keypad has three columns

// Define the Keymap
char keys[ROWS][COLS] = {
  {'1','2','3'},
  {'4','5','6'},
  {'7','8','9'},
  {'*','0','#'}
};
```

```
// Connect keypad ROW0, ROW1, ROW2 and ROW3 to these Arduino pins.
byte rowPins[ROWS] = { 8, 7, 4, 2 };
// Connect keypad COL0, COL1 and COL2 to these Arduino pins.
byte colPins[COLS] = { 12, 11, 10 };

// Create the Keypad
Keypad kpd = Keypad( makeKeymap(keys), rowPins, colPins, ROWS, COLS );

void setup() {
  setupPLC();                 // Define input / output pins
}

void loop()
{
  keyPress = kpd.getKey();    // Read key pressed (if any)

  latchKey('1', '2', Y0);     // Keyboard latch, Set = '1', Reset = '2',
                              // output = Y0 (pin 3)

  latchKey('3', '4', Y1);     // Keyboard latch, Set = '3', Reset = '4',
                              // output = Y1 (pin 5)

  latchKey('5', '6', Y2);     // Keyboard latch, Set = '5', Reset = '6',
                              // output = Y2 (pin 6)

  latchKey('7', '8', Y3);     // Keyboard latch, Set = '7', Reset = '8',
                              // output = Y3 (pin 9)
}

// Keypad-based Set-Reset latch, outputting to a pin.
unsigned int latchKey(char en, char dis, int outPin) {
  if(keyPress) {
    if (keyPress == en) {
      digitalWrite(outPin, HIGH);
    }
    if (keyPress == dis) {
      digitalWrite(outPin, LOW);
    }
  }
}
```

Listing 11. *Matrix Keypad with On/Off Control of Latched LED Outputs (Source: File > Examples > plcLib > Keypad > LedOnOff)*

The overall effect of the four latchKey() commands in the above sketch is to activate LEDs connected to outputs *Y0*, *Y1*, *Y2* or *Y3* when keys '1', '3', '5' or '7' are pressed, with keys '2', '4', '6' and '8' turning the corresponding outputs off again.

A slight change to the switch allocations allows a single input key to act as a master reset, as shown in the following extract.

```
void loop()
{
  keyPress = kpd.getKey();    // Read key pressed (if any)

  latchKey('1', '*', Y0);     // Keyboard latch, Set = '1', Reset = '*',
                              // output = Y0 (pin 3)

  latchKey('2', '*', Y1);     // Keyboard latch, Set = '2', Reset = '*',
                              // output = Y1 (pin 5)

  latchKey('3', '*', Y2);     // Keyboard latch, Set = '3', Reset = '*',
                              // output = Y2 (pin 6)

  latchKey('4', '*', Y3);     // Keyboard latch, Set = '4', Reset = '*',
                              // output = Y3 (pin 9)
}
```

Listing 12. *Master reset key modification (Source: File > Examples > plcLib > Keypad > LedStop)*

It is also possible to store the latch output bit in a user defined variable, rather than sending it directly to an output pin. An example based on this approach is given in the **LedVariable** example sketch, while the underlying approach is discussed in the Using Variables in Programs section.

9 Using Time Delays

The plcLib software has commands to produce time delays based on the activation or deactivation of an input signal (the **timerOn()** and **timerOff()** commands), and can also produce an output pulse of fixed duration (the **timerPulse()** command). All commands accept two parameters which are firstly the name of the elapsed timer variable and secondly the required time delay in milliseconds.

*Avoid using the Arduino's own **delay()** command wherever possible, as this halts the PLC scan cycle for the duration of the delay, and only supports a single delay being active at any time. See the* [Advanced Concepts] *section for more details.*

9.1 Producing a Turn-on Delay

The **timerOn()** command delays activating an output until the input has been continuously active for the specified period of time in milliseconds.

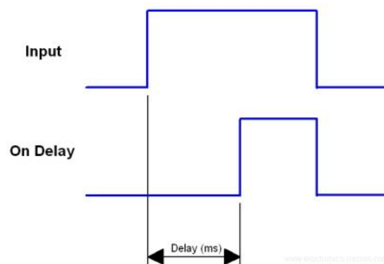


Figure 26. An on-delay timer provides delayed activation of an output..

A timer-based system may be be conveniently represented using a *block diagram* or [function block diagram], as shown in the following example.

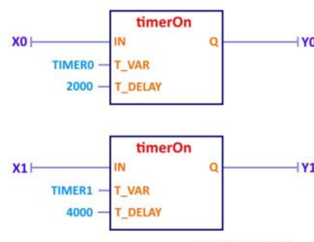


Figure 27. A pair of on-delay timers produce separate delays, controlled by two different inputs.

The two on-delay timers provide independent activation time delays of 2 seconds and 4 seconds on Outputs 0 and 1, respectively. Output 0 becomes active after a signal connected to Input 0 has been active for 2 seconds, while Output 1 requires Input 1 to have been active for a period of 4 seconds. The equivalent sketch is shown below.

```
#include <plcLib.h>

/* Programmable Logic Controller Library for the Arduino and Compatibles

   Turn-on Delay - Delays turning an output on after an input is applied

   Connections:
```

```
   Input - switch connected to input X0 (Arduino pin A0)
   Input - switch connected to input X1 (Arduino pin A1)
   Output with 2 s delay - LED connected to output Y0 (Arduino pin 3)
   Output with 4 s delay - LED connected to output Y1 (Arduino pin 5)

   Software and Documentation:
   https://github.com/wditch/plcLib

*/

unsigned long TIMER0 = 0;  // Variable to hold elapsed time for Timer 0
unsigned long TIMER1 = 0;  // Variable to hold elapsed time for Timer 1

void setup() {
  setupPLC();              // Setup inputs and outputs
}

void loop() {
  in(X0);                  // Read Input 0
  timerOn(TIMER0, 2000);   // 2 second delay
  out(Y0);                 // Output to Output 0

  in(X1);                  // Read Input 1
  timerOn(TIMER1, 4000);   // 4 second delay
  out(Y1);                 // Output to Output 1
}
```

Listing 13. *Turn-on Delay (Source: File > Examples > plcLib > TimeDelays > DelayOn)*

Notice that each timer makes internal use of an elapsed timer variable of type 'unsigned long', which must be declared at the start of the program.

9.2 Switch Debouncing

A common problem with switch- or keypad-based systems is *contact bounce* in which a single key press causes the contacts to physically bounce several times – over a period of several milliseconds – before settling. This mechanical effect can be eliminated using an on-delay timer having a suitable time delay, as shown in the following example.

```
#include <plcLib.h>

/* Programmable Logic Controller Library for the Arduino and Compatibles

   Switch Debounce - Delays a switch input by 10 ms to remove contact bounce

   Connections:
   Input - switch connected to input X0 (Arduino pin A0)
   Output - debounced output - LED connected to output Y0 (Arduino pin 3)

   Software and Documentation:
   https://github.com/wditch/plcLib

*/

unsigned long TIMER0 = 0;  // Define variable used to hold timer 0 elapsed time

void setup() {
  setupPLC();              // Setup inputs and outputs
}

void loop() {
  in(X0);                  // Read Input 0
  timerOn(TIMER0, 10);     // 10 ms delay
  out(Y0);                 // Output to Output 0
}
```

__Listing 14.__ Switch Debounce (Source: File > Examples > plcLib > TimeDelays > SwitchDebounce)

9.3 Creating a Turn-off Delay

A turn-off delay timer becomes active immediately and then delays turning-off for a given period after the controlling input is removed.

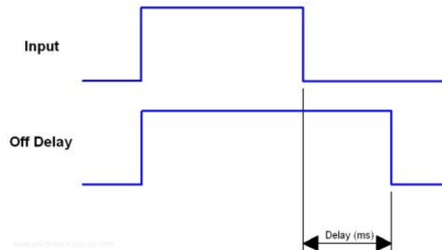


Figure 28. An off-delay causes an output to remain active for a fixed duration after the input is removed.

An application of the turn-off delay is 'pulse stretching' in which a brief input signal is expanded to have a minimum pulse width. The following function block diagram shows an off-delay timer with a 2 second turn-off delay.

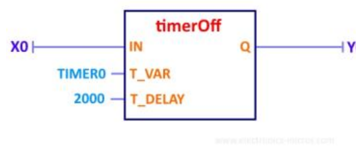


Figure 29. An off-delay timer with a 2 second delay.

The equivalent sketch is given below.

```
#include <plcLib.h>

/* Programmable Logic Controller Library for the Arduino and Compatibles

   Turn-off Delay - Delays turning an output off after an input is removed

   Connections:
   Input - switch connected to input X0 (Arduino pin A0)
   Output - LED connected to output Y0 (Arduino pin 3)

   Software and Documentation:
   https://github.com/wditch/plcLib

*/

unsigned long TIMER0 = 0;  // Variable to hold elapsed time for Timer 0

void setup() {
  setupPLC();               // Setup inputs and outputs
}

void loop() {
  in(X0);                   // Read Input 0
  timerOff(TIMER0, 2000);   // 2 second turn-off delay
  out(Y0);                  // Output to Output 0
}
```

__Listing 15.__ Turn-off Delay (Source: File > Examples > plcLib > TimeDelays > DelayOff)

9.4 Creating a Fixed Duration Pulse

The **timerPulse()** command (available with Version 1.0 or later) creates a fixed width output pulse when triggered by a brief input pulse. This is equivalent to an electronic *monostable* circuit.

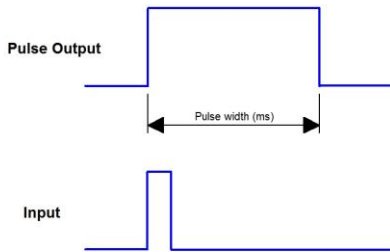


Figure 30. A fixed-width pulse is created by the timerPulse() command..

An example sketch is shown below.

```
#include <plcLib.h>

/* Programmable Logic Controller Library for the Arduino and Compatibles

   Fixed Pulse - Activates an output for a fixed period after a momentary input is applie
d

   Connections:
   Input - switch connected to input X0 (Arduino pin A0)
   Output with 2 s pulse - LED connected to output Y0 (Arduino pin 3)

   Software and Documentation:
   https://github.com/wditch/plcLib

*/

unsigned long TIMER0 = 0;      // Variable to hold elapsed time for Timer 0

void setup() {
  setupPLC();                  // Setup inputs and outputs
}

void loop() {
  in(X0);                      // Read Input 0
  timerPulse(TIMER0, 2000);    // 2 second pulse
  out(Y0);                     // Output to Output 0
}
```

Listing 16. *Fixed width pulse (Source location: File > Examples > plcLib > TimeDelays > FixedPulse)*

Note: *In Version 1.1 or later, operation of the timerPulse command is modified to be edge triggered rather than level triggered. Hence the timer is re-triggered by the first low to high transition of the trigger input after the previous pulse has ended. (This now agrees with the standard behaviour in IEC standard 61131-3.)*

The next section extends the time delay concepts introduced here to create continuously repeating waveforms.

10 Producing Repeating Waveforms

A repeating pulse may be defined using the duration of the *low* and *high* components of the waveform.

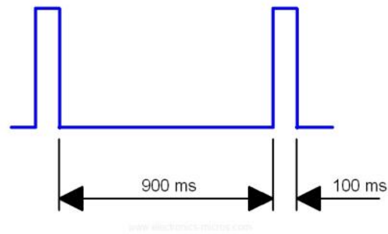


Figure 31. A repeating pulse may be defined in terms of the durations of the low and high sections.

The time taken for one complete cycle is called the *periodic time* (or *period*), which is obtained by adding the durations of the low and high components (a duration of 1 second in the above example).

Repeating pulses may either be created manually, from simpler components, or by using a dedicated command.

10.1 Manual Pulse Creation

A pair of cross-connected on-delay timers may be used to create a repeating pulse, as shown in the following example.

```
#include <plcLib.h>

/* Programmable Logic Controller Library for the Arduino and Compatibles

   Repeating pulse using a pair of linked on-delay timers

   Connections:
   Input - switch connected to input X0 (Arduino pin A0)
   Output - LED connected to output Y0 (Arduino pin 3)


   Software and Documentation:
   https://github.com/wditch/plcLib

*/

unsigned long TIMER0 = 0;  // Variable to hold elapsed time for Timer 0
unsigned long TIMER1 = 0;  // Variable to hold elapsed time for Timer 1

void setup() {
  setupPLC();              // Setup inputs and outputs
}

void loop() {
  in(X0);                  // Read Input 0 (enable)
  andNotBit(Y0);           // And with inverted output
  timerOn(TIMER0, 900);    // 900 ms (0.9 s) delay
  set(Y0);                 // Set Output 0 on time-out

  in(X0);                  // Read Input 0 (enable)
  andBit(Y0);              // And with output
  timerOn(TIMER1, 100);    // 100 ms (0.1 s) delay
  reset(Y0);               // Reset Output 0 on time-out
}
```

Listing 17. *Repeating pulse using a pair of linked on-delay timers (Source: File > Examples > plcLib > Waveforms > PulsedOutputManual)*

With the program running, it will be seen that continuously pressing the *enable* input (*X0*) causes the output (*Y0*) to turn on for 0.1 s, then off for 0.9 s in a repeating sequence.

The program makes use of a pair of on-delay timers, plus a set-reset latch based around output *Y0*. The first timer is initially enabled, assuming output *Y0* is zero, and the enable input is also on. The first timer output goes high after 900 ms, which in turn sets output *Y0* to 1. At this point, the second on-delay timer is enabled and begins to count. After a further 100 ms, the second timer output becomes set, which causes output *Y0* to be cleared, and the whole cycle begins again.

This is an example of a very simple Finite State Machine (FSM), which is the basis of sequential function charts.

Don't worry if the above example seems rather complex, as this same functionality is provided by the **timerCycle()** command, discussed next.

10.2 Using the timerCycle() Command

The **timerCycle()** command may be used to produce a repeating pulse waveform, which could for example be used to repeatedly flash an LED. (As the command name suggests, this is sometimes called a *cycle timer*.)

An equivalent function block diagram symbol is shown below.

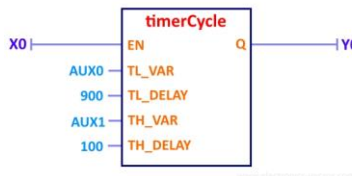


Figure 32. A cycle timer produces a repeating pulse waveform, when enabled.

The command requires four parameters to be specified for each waveform. Parameters 1 and 3 must be specified as variables (of type *unsigned long*) and are used internally by the plcLib software to measure the elapsed time for the low and high sections of the waveform respectively. Associated low and high pulse widths are defined in the 2nd and 4th parameters. The following example illustrates the process.

```
#include <plcLib.h>

/* Programmable Logic Controller Library for the Arduino and Compatibles

   Pulsed Output - Creating a repeating pulse using the timerCycle command

   Connections:
   Input - Enable input connected to input X0 (Arduino pin A0)
   Output - Pulse Waveform on LED connected to output Y0 (Arduino pin 3)

   Software and Documentation:
   https://github.com/wditch/plcLib

*/
```

```
// Variables:
unsigned long AUX0 = 0;                 // Pulse low timer variable
unsigned long AUX1 = 0;                 // Pulse high timer variable


void setup() {
  setupPLC();                           // Define inputs and outputs
}

void loop() {
  in (X0);                              // Read Enable input (1 = enable)
  timerCycle(AUX0, 900, AUX1, 100);     // Repeating pulse, low = 0.9 s, high = 0.1 s
                                        // (hence period = 1 second)
  out(Y0);                              // Send pulse waveform to Output 0
}
```

Listing 18. *Pulsed Output - Creating a repeating pulse using the timerCycle command (Source: File > Examples > plcLib > Waveforms > PulsedOutput)*

Note: *A typical application is the creation of a repeating 'alarm armed' flashing pulse, as demonstrated in the* ***File > Examples > plcLib > Applications > AlarmWithArmedStatus*** *example sketch.*

11 Counting and Counters

Version 0.8 of the software introduces *counters*, which can activate an output once a predetermined number of events have occurred. A counter 'object' may be configured to count *up*, *down*, or a combination of both.

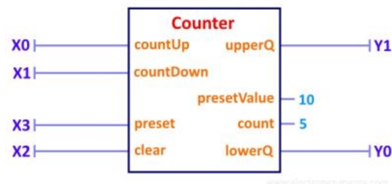


Figure 33. A counter object may be configured to count up, down, or both.

Each counter has four inputs, and four outputs, shown at the left and right of above diagram, respectively. In practice, only a subset of these may be required, depending on the type of counter required.

11.1 Up Counter

At its simplest, an up counter counts pulses received on its *countUp* input, activating the 'finished' or *upperQ* output when the required number of input pulses have been detected. To illustrate the process, the following example creates an up counter which counts from 0 to 10, driven by switch presses applied to input *X0*.

```
#include <plcLib.h>

/* Programmable Logic Controller Library for the Arduino and Compatibles

   Up Counter - Counts 10 pulses on Input 0 with switch debounce

   Connections:
   Count input - switch connected to input X0 (Arduino pin A0)
   Clear input - switch connected to input X1 (Arduino pin A1)
   Preset input - switch connected to input X2 (Arduino pin A2)
   Lower Q output - LED connected to output Y0 (Arduino pin 3)
   Upper Q output - LED connected to output Y1 (Arduino pin 5)

   Software and Documentation:
   https://github.com/wditch/plcLib

*/

Counter ctr(10);            // Final count = 10, starting at zero
unsigned long TIMER0 = 0;  // Define variable used to hold timer 0 elapsed time

void setup() {
  setupPLC();               // Setup inputs and outputs
}

void loop() {
  in(X0);                   // Read Input 0
  timerOn(TIMER0, 10);      // 10 ms switch debounce delay
  ctr.countUp();            // Count up

  in(X1);                   // Read input X1
  ctr.clear();              // Clear counter (counter at lower limit)

  in(X2);                   // Read input X2
```

```
  ctr.preset();             // Preset counter (counter at upper limit)

  ctr.lowerQ();             // Display Count Down output on Y0
  out(Y0);

  ctr.upperQ();             // Display Count Up output on Y1
  out(Y1);
}
```

Listing 19. *Up Counter (Source: File > Examples > plcLib > Counters > CountUp)*

Examining the above listing, the first step is to use the *Counter* command to create a counter object *ctr*, at the same time setting the final value to 10, which is equivalent to the *presetValue* property of the counter. The counter defaults to being an *up* counter, so the internal *count* value is initially zero, causing the *lowerQ* output to be activated. Pulses applied to the *X0* input are linked to the *countUp* input, causing the internal *count* value to be incremented by each 0 to 1 transition. After 10 pulses, the *count* becomes equal to the upper limit or *preset value*, causing the *upperQ* output to be enabled.

A commonly encountered problem with counter circuits is *switch bounce*, where the switch contacts 'bounce' several times (over a period of a few milliseconds) before settling. This in turn may cause a single switch press to update the counter several times in rapid succession. To avoid this effect, notice that a 10 millisecond on-delay timer has been linked in series with the *X0* switch connected to the *countUp* input.

It may also be necessary to manually force the counter to go back to the start, which is achieved in the above example by connecting switch input *X1* to the *clear* input of the counter. The *preset* input (connected to switch input *X2*) performs a similar function, but this time setting the internal count to the final or preset value. Hence pressing the switches connected to inputs *X1* or *X2* will cause the *lowerQ* or *upperQ* outputs to be immediately activated, respectively.

11.2 Down Counter

The creation of a *down counter* is quite similar, as shown in the following listing.

```
#include <plcLib.h>

/* Programmable Logic Controller Library for the Arduino and Compatibles

   Down Counter - Counts 5 pulses on Input 0 with switch debounce

   Connections:
   Count input - switch connected to input X0 (Arduino pin A0)
   Clear input - switch connected to input X1 (Arduino pin A1)
   Preset input - switch connected to input X2 (Arduino pin A2)
   Lower Q output - LED connected to output Y0 (Arduino pin 3)
   Upper Q output - LED connected to output Y1 (Arduino pin 5)

   Software and Documentation:
   https://github.com/wditch/plcLib

*/

Counter ctr(5, 1);         // Counts down, starting at 5
unsigned long TIMER0 = 0;  // Define variable used to hold timer 0 elapsed time

void setup() {
  setupPLC();              // Setup inputs and outputs
}

void loop() {
```

```
  in(X0);                    // Read Input 0
  timerOn(TIMER0, 10);       // 10 ms switch debounce delay
  ctr.countDown();           // Count down

  in(X1);                    // Read input X1
  ctr.clear();               // Clear counter (counter at lower limit)

  in(X2);                    // Read input X2
  ctr.preset();              // Preset counter (counter at upper limit)

  ctr.lowerQ();              // Display Count Down output on Y0
  out(Y0);

  ctr.upperQ();              // Display Count Up output on Y1
  out(Y1);
}
```

Listing 20. *Down Counter (Source: File > Examples > plcLib > Counters > CountDown)*

The first difference is in the creation of the Counter object *ctr*, using the **Counter ctr(5, 1);** command. As well setting the preset value to 5, the optional second parameter has a value of 1, causing the initial count value to be set equal to the upper limit, which is suitable for a down counter. (Omitting the second parameter, or setting it to zero inialises the count to zero). Input *X0* is then connected to the *countDown* input, while output *Y0* is driven by *lowerQ*. Hence *Y0* will go high after 5 pulses have been received on input *X0*.

11.3 Up/Down Counter

It is straightforward to combine these two approaches in order to create a device capable of counting up or down, as shown in the example below.

```
#include <plcLib.h>

/* Programmable Logic Controller Library for the Arduino and Compatibles

   Up-down Counter - Counts up or down in the range 0-10

   Connections:
   Count up input - switch connected to input X0 (Arduino pin A0)
   Count down input - switch connected to input X1 (Arduino pin A1)
   Clear input - switch connected to input X2 (Arduino pin A2)
   Preset input - switch connected to input X3 (Arduino pin A3)
   Lower Q output - LED connected to output Y0 (Arduino pin 3)
   Upper Q output - LED connected to output Y1 (Arduino pin 5)

   Software and Documentation:
   https://github.com/wditch/plcLib

*/

Counter ctr(10);             // Counts up or down in range 0-10, starting at zero
unsigned long TIMER0 = 0;    // Define variable used to hold timer 0 elapsed time
unsigned long TIMER1 = 0;    // Define variable used to hold timer 1 elapsed time

void setup() {
  setupPLC();                // Setup inputs and outputs
}

void loop() {
  in(X0);                    // Read Input 0
  timerOn(TIMER0, 10);       // 10 ms switch debounce delay
  ctr.countUp();             // Count up

  in(X1);                    // Read Input 1
  timerOn(TIMER1, 10);       // 10 ms switch debounce delay
```

```
  ctr.countDown();          // Count down

  in(X2);                   // Read input X1
  ctr.clear();              // Clear counter (counter at lower limit)

  in(X3);                   // Read input X2
  ctr.preset();             // Preset counter (counter at upper limit)

  ctr.lowerQ();             // Display Count Down output on Y0
  out(Y0);

  ctr.upperQ();             // Display Count Up output on Y1
  out(Y1);
}
```

Listing 21. *Up-down counter. (Source: File > Examples > plcLib > Counters> CountUpDown)*

The above counter has separate count-up and count-down inputs linked to input switches *X0* and *X1* respectively, and each counter input has its own debounced switch input.

11.4 Debugging Counter-based Applications

The output of a counter is activated once the defined number of pulses have been detected, but the internal count value is not normally visible to the user. In effect, the counter is either 'finished' or 'not finished'. It may sometimes be useful to view the internal count for debugging purposes, as shown in the following example.

```
#include <plcLib.h>

/* Programmable Logic Controller Library for the Arduino and Compatibles

   Up-down Counter - Counts up or down in the range 0-10
   (Sends current count to serial monitor)

   Connections:
   Count up input - switch connected to input X0 (Arduino pin A0)
   Count down input - switch connected to input X1 (Arduino pin A1)
   Clear input - switch connected to input X2 (Arduino pin A2)
   Preset input - switch connected to input X3 (Arduino pin A3)
   Lower Q output - LED connected to output Y0 (Arduino pin 3)
   Upper Q output - LED connected to output Y1 (Arduino pin 5)

   Software and Documentation:
   https://github.com/wditch/plcLib

*/

Counter ctr(10);            // Counts up or down in range 0-10, starting at zero
unsigned long TIMER0 = 0;   // Define variable used to hold timer 0 elapsed time
unsigned long TIMER1 = 0;   // Define variable used to hold timer 1 elapsed time

void setup() {
  setupPLC();               // Setup inputs and outputs
  Serial.begin(9600);       // Open serial connection over USB
}

void loop() {
  in(X0);                   // Read Input 0
  timerOn(TIMER0, 10);      // 10 ms switch debounce delay
  ctr.countUp();            // Count up

  in(X1);                   // Read Input 1
  timerOn(TIMER1, 10);      // 10 ms switch debounce delay
  ctr.countDown();          // Count down

  in(X2);                   // Read input X1
```

```
  ctr.clear();              // Clear counter (counter at lower limit)

  in(X3);                   // Read input X2
  ctr.preset();             // Preset counter (counter at upper limit)

  ctr.lowerQ();             // Display Count Down output on Y0
  out(Y0);

  ctr.upperQ();             // Display Count Up output on Y1
  out(Y1);

  Serial.println(ctr.count());  // Send count to serial port
  delay(100);
}
```

Listing 22. *Up-down counter with debugging. (Source: File > Examples > plcLib > Counters > CountUpDownDebug)*

To debug the application, simply open the *Serial Monitor* while the sketch is running. This will repeatedly display the internal count value, with the repetition interval controlled by the delay command towards the end of the listing. (However, don't forget to remove the debugging code, once your application is working as intended.)

12 Shifting and Rotating Binary Data

Version 0.9 of the software introduces *shift registers* which may be used to shift binary data to the left or right, by one bit position at a time.

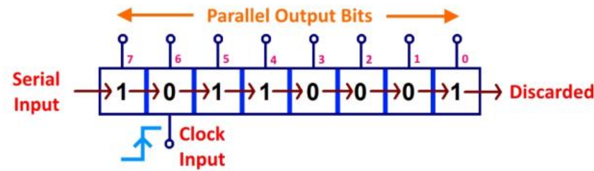


Figure 34. A simple shift register moves data one place to the right on each rising edge of the Clock.

The above example shows a simple 8-bit shift register, which shifts data one position to the right on each rising edge of the Clock input. New data is shifted in at the left side, while data is discarded as it leaves at the right. The content of individual bits may be read from output connections at the top.

Considering the numerical content of the shift register, shifting data to the right is equivalent to division by two, since the most significant bit is at the left. Conversely, shifting to the left would be equivalent to multiplication by two.

12.1 Creating and Using Shift Registers

The first step is to use the **Shift** command to create a shift register *object*, which has a fixed data width of 16-bits. The initial content of the register may optionally be set at creation. For example, the command **Shift shift1(0x8888);** creates a shift register object called *shift1* with an initial hexadecimal content of 0x8888, which is equivalent to 1000 1000 1000 1000 in binary.

Several other configuration tasks are also required, including definition of the *serial data input*, *clock input* (shifting data to the right in this case), *reset input* and any *parallel output* connections, as illustrated by the following function block symbol.

Figure 35. A function block representation of the shift register.

The actual coding of the shift register is given by the following sketch.

```
#include <plcLib.h>

/* Programmable Logic Controller Library for the Arduino and Compatibles

   Shift register: Shift data to the right

   Connections:
   Serial Input - switch connected to input X0 (Arduino pin A0)
   Clock Input - switch connected to input X1 (Arduino pin A1)
```

```
   Reset Input - switch connected to input X2 (Arduino pin A2 )
   Output - LED connected to output Y0 (Arduino pin 3)
   Output - LED connected to output Y1 (Arduino pin 5)
   Output - LED connected to output Y2 (Arduino pin 6)
   Output - LED connected to output Y3 (Arduino pin 9)

   Software and Documentation:
   https://github.com/wditch/plcLib

*/

Shift shift1(0x8888);      // Create a 16 bit shift register with initial value 0x8888

                           //        Bit      1 1 1 1    1 1
                           //     Positions   5 4 3 2    1 0 9 8    7 6 5 4    3 2 1 0
                           //                 | | | |    | | | |    | | | |    | | | |
                           //  X0 -> -> -> -> 1 0 0 0 -> 1 0 0 0 -> 1 0 0 0 -> 1 0 0 0 ->
                           //                 | | | |
                           //  X1 -> Clock    | | | |
                           //                 | | | |
                           //  X2 -> Reset    | | | |
                           //
                           //                 Y Y Y Y
                           //      Outputs    3 2 1 0

unsigned long TIMER0 = 0;  // Define variable used for switch debounce

void setup() {
  setupPLC();              // Setup inputs and outputs
}

void loop() {

  in(X0);                  // Read input to shift register from X0
  shift1.inputBit();

  in(X1);                  // Shift Right on rising edge of input X1
  timerOn(TIMER0, 10);     // 10 ms switch debounce delay on X1
  shift1.shiftRight();

  in(X2);                  // Reset the shift register value to zero if X2 = 1
  shift1.reset();

  shift1.bitValue(15);      // Send bit 15 value to output Y3
  out(Y3);

  shift1.bitValue(14);      // Send bit 14 value to output Y2
  out(Y2);

  shift1.bitValue(13);      // Send bit 13 value to output Y1
  out(Y1);

  shift1.bitValue(12);      // Send bit 12 value to output Y0
  out(Y0);
}
```

Listing 23. *Shift register – shifting data to the right (Source: File > Examples > plcLib > ShiftRotate > ShiftRight)*

Data is shifted right by the rising edge of the clock input taken from a switch connected to input *X1*, with new data shifted-in at the left taken from input switch *X0*. (Notice that the potential problem of *contact bounce* on the clock input is avoided by the use of a 10 ms switch debounce delay.) Input swicth *X2* provides a reset input, clearing the shift register to zero when pressed. Parallel outputs are taken from bits 15–12, allowing any newly inputted serial data to be immediately visible on the outputs.

It is straightforward to modify the above sketch to shift data to the left, as shown below.

```
#include <plcLib.h>

/* Programmable Logic Controller Library for the Arduino and Compatibles

   Shift register: Shift data to the left

   Connections:
   Serial Input - switch connected to input X0 (Arduino pin A0)
   Clock Input - switch connected to input X1 (Arduino pin A1)
   Reset Input - switch connected to input X2 (Arduino pin A2)
   Output - LED connected to output Y0 (Arduino pin 3)
   Output - LED connected to output Y1 (Arduino pin 5)
   Output - LED connected to output Y2 (Arduino pin 6)
   Output - LED connected to output Y3 (Arduino pin 9)

   Software and Documentation:
   https://github.com/wditch/plcLib

*/

Shift shift1(0x1111);      // Create a 16 bit shift register with initial value 0x1111

                           //      Bit      1 1 1 1    1 1
                           //   Positions   5 4 3 2    1 0 9 8    7 6 5 4    3 2 1 0
                           //               | | | |    | | | |    | | | |    | | | |
                           //            <- 0 0 0 1 <- 0 0 0 1 <- 0 0 0 1 <- 0 0 0 1 <-
X0
                           //                                                | | | |
                           //  X1 -> Clock                                   | | | |
                           //                                                | | | |
                           //  X2 -> Reset                                   | | | |
                           //
                           //                                                Y Y Y Y
                           //                                     Outputs    3 2 1 0

unsigned long TIMER0 = 0;  // Define variable used for switch debounce

void setup() {
  setupPLC();              // Setup inputs and outputs
}

void loop() {

  in(X0);                  // Read input to shift register from X0
  shift1.inputBit();

  in(X1);                  // Shift Left on rising edge of input X1
  timerOn(TIMER0, 10);     // 10 ms switch debounce delay on X1
  shift1.shiftLeft();

  in(X2);                  // Reset the shift register value to zero if X2 = 1
  shift1.reset();

  shift1.bitValue(3);      // Send bit 3 value to output Y3
  out(Y3);

  shift1.bitValue(2);      // Send bit 2 value to output Y2
  out(Y2);

  shift1.bitValue(1);      // Send bit 1 value to output Y1
  out(Y1);

  shift1.bitValue(0);      // Send bit 0 value to output Y0
  out(Y0);
}
```

Listing 24. *Shift register – shifting data to the left (Source: File > Examples > plcLib > ShiftRotate > ShiftLeft)*

The main differences are firstly the connection of the clock (switch *X1*) to the *shiftLeft* input, and secondly the connection of parallel outputs to bits 3–0 of the shift register. (The latter allows data shifted-in at the right to be immediately visible on outputs, as it moves to the left).

12.2 Rotating Data

Adding a feedback connection to our shift register from output to input allows data to be rotated in a continuous loop, as shown below.

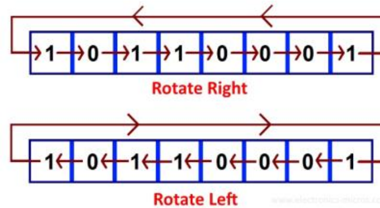


Figure 36. Adding a feedback connection causes data to be rotated right or left in a continuous loop.

In fact, the feedback connection may be taken from any convenient output bit, hence creating a circulating bit pattern of any desired width. As an example, the following sketch uses shift register bit 12 as the feedback connection to the serial input (i.e. the 4th bit from the left), to rotate a 4-bit data word to the right.

```
#include <plcLib.h>

/* Programmable Logic Controller Library for the Arduino and Compatibles

   Shift register: Rotate data to the right

   Connections:
   Clock Input - switch connected to input X0 (Arduino pin A0)
   Reset Input - switch connected to input X1 (Arduino pin A1)
   Output - LED connected to output Y0 (Arduino pin 3)
   Output - LED connected to output Y1 (Arduino pin 5)
   Output - LED connected to output Y2 (Arduino pin 6)
   Output - LED connected to output Y3 (Arduino pin 9)

   Software and Documentation:
   https://github.com/wditch/plcLib

*/

Shift shift1(0x8000);      // Create a shift register with initial value 0x8000
                           // (Leftmost 4 bits are rotated to the right)

                           //    Outputs         Y Y Y Y
                           //                    3 2 1 0
                           //
                           //      Bit           1 1 1 1
                           //   Positions        5 4 3 2
                           //                    | | | |
                           //                 -> 1 0 0 0 ->
                           //                 |           |
                           //                 |           |
                           //  X0 -> Clock    -<--<--<--<--
                           //
                           //  X1 -> Reset

unsigned long TIMER0 = 0;  // Define variable used for switch debounce

void setup() {
```

```
  setupPLC();               // Setup inputs and outputs
}

void loop() {

  shift1.bitValue(12);     // Read input to shift register from bit 12 at RHS
  shift1.inputBit();

  in(X0);                  // Rotate Right on rising edge of input X0
  timerOn(TIMER0, 10);     // 10 ms switch debounce delay on X0
  shift1.shiftRight();

  in(X1);                  // Reset the shift register value to zero if X1 = 1
  shift1.reset();

  shift1.bitValue(15);      // Send bit 15 value to output Y3
  out(Y3);

  shift1.bitValue(14);      // Send bit 14 value to output Y2
  out(Y2);

  shift1.bitValue(13);      // Send bit 13 value to output Y1
  out(Y1);

  shift1.bitValue(12);      // Send bit 12 value to output Y0
  out(Y0);
}
```

Listing 25. *Shift register – rotate data to the right (Source: File > Examples > plcLib > ShiftRotate > RotateRight)*

An equivalent sketch to rotate a 4-bit data word to the left would use bit 3 (the 4th bit from the right) as the feedback connection, as shown in the following example.

```
#include <plcLib.h>

/* Programmable Logic Controller Library for the Arduino and Compatibles

   Shift register: Rotate data to the left

   Connections:
   Clock Input - switch connected to input X0 (Arduino pin A0)
   Reset Input - switch connected to input X1 (Arduino pin A1)
   Output - LED connected to output Y0 (Arduino pin 3)
   Output - LED connected to output Y1 (Arduino pin 5)
   Output - LED connected to output Y2 (Arduino pin 6)
   Output - LED connected to output Y3 (Arduino pin 9)

   Software and Documentation:
   https://github.com/wditch/plcLib

*/

Shift shift1(0x0001);      // Create a shift register with initial value 0x0001
                           // (Rightmost 4 bits are rotated to the left)

                           //    Outputs         Y Y Y Y
                           //                    3 2 1 0
                           //      Bit
                           //   Positions        3 2 1 0
                           //                    | | | |
                           //                 <- 0 0 0 1 <-
                           //                 |           |
                           //                 |           |
                           //  X0 -> Clock    ->-->-->-->--
                           //
                           //  X1 -> Reset

unsigned long TIMER0 = 0;  // Define variable used for switch debounce
```

```
void setup() {
  setupPLC();               // Setup inputs and outputs
}

void loop() {

  shift1.bitValue(3);        // 'Feedback' input to shift register from bit 3 at LHS
  shift1.inputBit();

  in(X0);                   // Rotate Left on rising edge of input X0
  timerOn(TIMER0, 10);      // 10 ms switch debounce delay on X0
  shift1.shiftLeft();

  in(X1);                   // Reset the shift register value to zero if X1 = 1
  shift1.reset();

  shift1.bitValue(3);       // Send bit 3 value to output Y3
  out(Y3);

  shift1.bitValue(2);       // Send bit 2 value to output Y2
  out(Y2);

  shift1.bitValue(1);       // Send bit 1 value to output Y1
  out(Y1);

  shift1.bitValue(0);       // Send bit 0 value to output Y0
  out(Y0);
}
```

Listing 26. *Shift register – rotate data to the left (Source: File > Examples > plcLib > ShiftRotate > RotateLeft)*

13 Working with Analogue Signals

The **inAnalog()** command reads an analogue input. This may then be used to control a continuously variable output value, such as the brightness of an LED, the speed of a motor, or the position of a servo. Scaling of inputs and outputs is performed automatically by the plcLib software.

*The **scanValue** global variable holds the inputted value, which is in the range 0-1023 for an analogue input. This is automatically scaled in the range 0-255 or 0-179 when used to control a PWM or servo output, respectively.*

13.1 Controlling LED Brightness using PWM

Linking the **inAnalog()** command to the **outPWM()** command, allows a PWM output to be controlled from an analogue input. As an example, the following sketch reads a potentiometer connected to input *X0* and produces a repeating pulse waveform with a variable duty cycle on output *Y0*.

```
#include <plcLib.h>

/* Programmable Logic Controller Library for the Arduino and Compatibles

   PWM (Pulse Width Modulation) - Analogue control of a PWM output

   Connections:
   Input - potentiometer connected to input X0 (Arduino pin A0)
   Output - LED connected to output Y0 (Arduino pin 3)

   Software and Documentation:
   https://github.com/wditch/plcLib

*/

void setup() {
  setupPLC();       // Setup inputs and outputs
}

void loop() {
  inAnalog(X0);     // Read Analogue Input 0
  outPWM(Y0);       // Send to Output 0 as PWM waveform
}
```

Listing 27. *Analogue control of a PWM output (Source: File > Examples > plcLib > InputOutput > PWM)*

13.2 Controlling the Speed and Direction of a Motor

The *Arduino Motor Shield* is based on a commercially available H-bridge driver IC, allowing the speed and direction of up to two DC motors to be controlled. A possible hardware connection is shown below.

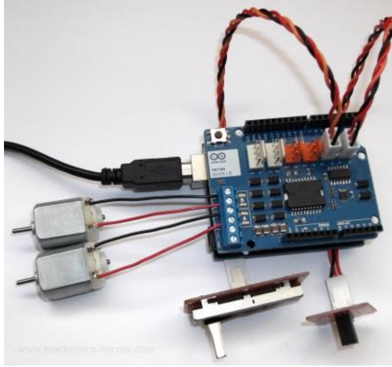


Figure 37. An Arduino Motor Shield allows up to two motors to be controlled.

Two motor channels are available – *Channel A* and *Channel B* – and the shield also supports a small number of Tinkerkit compatible input and output connectors (plus two TWI / I2C interfaces not considered here). The above image shows a linear potentiometer connected to pin *X2* (*A2*), used to control the motor speed via PWM, and a tilt switch controlling the motor direction, linked to pin *X3* (*A3*).

The following example controls the speed and direction of a motor connected to Channel A.

```
#include <plcLib.h>

/* Programmable Logic Controller Library for the Arduino and Compatibles

   Motor Channel A - Simple Motor Control on Arduino Motor Shield Channel A

   Connections:
   Input - Speed - potentiometer connected to input X2 (Arduino pin A2)
   Input - Direction - switch connected to input X3 (Arduino pin A3)
   Output - Channel A Direction (DIRA) - Arduino pin 12
   Output - Channel A PWM (PWMA) - Arduino pin 3
   Output - Channel A Brake (BRAKEA) - Arduino pin 9

   Software and Documentation:
   https://github.com/wditch/plcLib

*/

// Variables
unsigned int RUN = 0;  // Disable brake

void setup() {
  setupPLC();           // Setup inputs and outputs

                        // Turn off Channel A Brake
  in(RUN);              // Read RUN variable (0 = brake off)
  out(BRAKEA);          // Output to BRAKEA
}

void loop() {
                        // Control direction of motor
  in(X3);               // Read switch connected to Input 3
  out(DIRA);            // Output to DIRA (motor direction)

                        // Read Analog Input 2 and send to Channel A PWM
  inAnalog(X2);         // Read from potentiometer connected to Analogue Input 2
  outPWM(PWMA);         // Output to PWMA (motor speed)
}
```

Listing 28. *Simple Motor Control on Arduino Motor Shield Channel A (Source: File > Examples > plcLib > Motor > MotorChannelA)*

Note: *See the* ***Motor*** *section of the example files for an equivalent sketch to control a motor connected to Channel B.*

Notice that a number of predefined variables are available to simplify the coding of motor control software as far as possible, as summarised below:

- *BRAKEA* and *BRAKEB* enable or disable the brake on each channel (0 = brake off).
- *DIRA* and *DIRB* set the forward or reverse direction of each motor.
- *PWMA* and *PWMB* control the motor speed of their related channels.

14 Position Control Using Servos

Controlling a servo is slightly more complex, due to the need to work with the Servo library which is supplied as a standard part of the Arduino environment. The **outServo()** replaces the **outPWM()** command used previously, and is defined locally, as shown below.

```
#include <Servo.h>
#include <plcLib.h>

/* Programmable Logic Controller Library for the Arduino and Compatibles

   Single Servo - Controlling the position of a servo using a potentiometer

   Connections:
   Input - potentiometer connected to input X0 (Arduino pin A0)
   Output - servo connected to output Y0 (Arduino pin 3)

   Software and Documentation:
   https://github.com/wditch/plcLib

*/

Servo servo1;           // Create a servo object to control a single servo

extern int scanValue;  // Link to scanValue defined in PLC library file

void setup() {
  setupPLC();           // Setup inputs and outputs
  servo1.attach(Y0);    // Attaches Servo 1 to Output 0
}

void loop() {
  inAnalog(X0);         // Read potentiometer connected to input 0
  outServo(servo1);     // Output to Servo 1 (connected to Output 0)
}

// The outServo function is defined locally
int outServo(Servo &ServoObj) {
  ServoObj.write(map(scanValue, 0, 1023, 0, 179));
  return(scanValue);
}
```

Listing 29. *Single Servo (Source: File > Examples > plcLib > InputOutput > ServoSingle)*

Note: *See the* ***InputOutput*** *section of the example files for a dual servo control sketch.*

15 Comparing Analogue Values

Analogue comparison commands provide equivalent functionality to an electronic *comparator* circuit, giving a *true* / *false* result based on which of the two tested analogue signals is the larger.

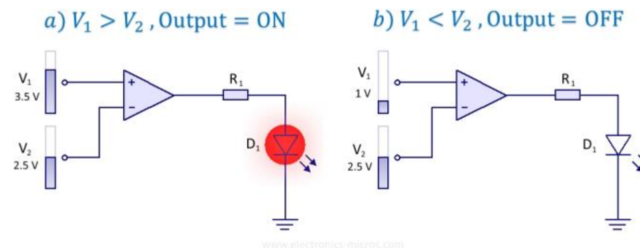


Figure 38. A comparator 'tests' the relative magnitude of two analogue inputs.

The circuit symbol for an electronic comparator is triangular, with two analogue inputs at the left and a single digital output at the left. A *high* output is produced if the voltage applied to the upper V^+, input terminal is greater than that connected to the lower, V^- input (called the *non-inverting input* and the *inverting input*, respectively). Otherwise, a *low* output is produced.

15.1 Software-based Comparison of Analogue Values

Comparing analogue values in software is a three-step process:

1. Input the first analogue value.
2. Compare this with a second value.
3. Output the comparison result to a digital output.

The two available variations of the comparison operation are **compareGT()** and **compareLT()** which test whether an input is either greater than or less than a reference value, respectively.

The following example tests whether the analogue voltage on input *X0* is greater than that on input *X1*, setting output *Y0* if this is true.

```
#include <plcLib.h>

/* Programmable Logic Controller Library for the Arduino and Compatibles

   Comparator - Greater than test between two input pins

   Connections:
   Analogue Input - potentiometer connected to input X0 (Arduino pin A0)
   Analogue Input - potentiometer connected to input X1 (Arduino pin A1)
   Digital Output - LED connected to output Y0 (Arduino pin 3)

   Software and Documentation:
   https://github.com/wditch/plcLib

*/

void setup() {
  setupPLC();         // Setup inputs and outputs
}

void loop() {
```

```
  inAnalog(X0);       // Read Analogue Input 0
  compareGT(X1);      // X0 > X1 ?
  out(Y0);            // Y0 = 1 if X0 > X1, Y0 = 0 otherwise

}
```

Listing 30. *Greater than test between two input pins using a comparator (Source: File > Examples > plcLib > AnalogCompare > GreaterThan)*

It is also possible to compare an analogue input against a fixed reference, as shown in the following example.

```
#include <plcLib.h>

/* Programmable Logic Controller Library for the Arduino and Compatibles

   Comparator - Less than test between an input and a fixed threshold

   Connections:
   Analogue Input - potentiometer connected to input X0 (Arduino pin A0)
   Digital Output - LED connected to output Y0 (Arduino pin 3)

   Software and Documentation:
   https://github.com/wditch/plcLib

*/

unsigned int threshold = 500;   // Analogue threshold = 500

void setup() {
  setupPLC();                   // Setup inputs and outputs
}

void loop() {
  inAnalog(X0);                 // Read Analogue Input 0
  compareLT(threshold);         // X0 < 500?
  out(Y0);                      // Y0 = 1 if X0 < 500, Y0 = 0 otherwise

}
```

Listing 31. *Less than comparator test between an input and a fixed threshold (Source: File > Examples > plcLib > AnalogCompare > LessThanThreshold)*

15.2 A Simple Comparator Application

In this example, a pair of comparators are used to test whether an analogue input is either above an upper threshold (greater than 3.5 V), or below a lower threshold (less than 1.5 V), as shown below.

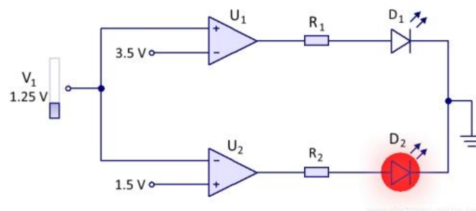


Figure 39. A dual-threshold comparator indicates if the input is above or below an allowed 'window'.

The upper comparator causes its associated LED to light if the input voltage, V_1 is greater than a fixed threshold of 3.5 V. Conversely, the lower comparator gives an output if its input voltage is less than its threshold voltage of 1.5 V. (Carefully note the polarity of each comparator input connection to understand how 'greater than' or 'less than' tests are achieved.)

An equivalent sketch is given below, which makes use of both forms of comparator command, together with calculated threshold values based on a 5 V power supply.

```
#include <plcLib.h>

/* Programmable Logic Controller Library for the Arduino and Compatibles

   Comparator - Maximum / minimum test using fixed threshold values

   Connections:
   Analogue Input - potentiometer connected to input X0 (Arduino pin A0)
   Digital Output - 'High' LED connected to output Y0 (Arduino pin 3)
   Digital Output - 'Low' LED connected to output Y1 (Arduino pin 5)

   Software and Documentation:
   https://github.com/wditch/plcLib

*/

unsigned int lowLimit = 307;    // Analogue lower threshold = 307
                                // (30% of 1024 = 1024 * 0.3 = 307)
                                // Lower threshold voltage = Vsupply * 0.3
                                // (1.5 V if Vsupply = 5 V)

unsigned int highLimit = 717;   // Analogue lower threshold = 717
                                // (70% of 1024 = 1024 * 0.7 = 717)
                                // Upper threshold voltage = Vsupply * 0.7
                                // (3.5 V if Vsupply = 5 V)

void setup() {
  setupPLC();                   // Setup inputs and outputs
}

void loop() {
  inAnalog(X0);                 // Read Analogue Input 0
  compareGT(highLimit);         // X0 > upper threshold?
  out(Y0);                      // Y0 = 1 if X0 > 717, Y0 = 0 otherwise

  inAnalog(X0);                 // Read Analogue Input 0
  compareLT(lowLimit);          // X0 < lower threshold?
  out(Y1);                      // Y1 = 1 if X0 < 307, Y1 = 0 otherwise
}
```

Listing 32. *Maximum / minimum comparator test using fixed threshold values (Source: File > Examples > plcLib > AnalogCompare > MaxMin)*

Following sections introduce the range of commonly used PLC programming techniques, starting with *Instruction List* programming.

16 Instruction List Programming

Instruction list programming is a text based language, used to describe PLC programs, and is one of five methods specified by international standard IEC 61131-3, with the others being ladder diagram, function block diagram, sequential function chart and structured text.

Examples of instruction list commands include reading inputs, performing logical operations and controlling outputs, although details of command syntax tend to vary between manufacturers. A close relationship exists between instruction list and graphical design methods, including ladder diagrams and function block diagrams, as shown by the following illustration.

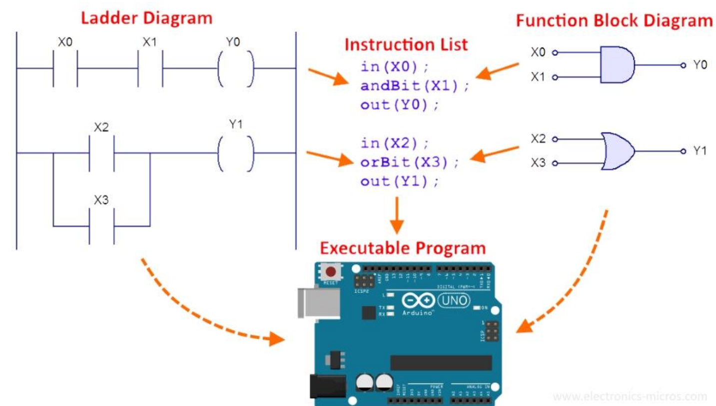


Figure 40. A close equivalence exists between instruction list and graphical design methods.

Instruction list is sometimes claimed to be a 'low level language', reminiscent of *assembly language*, but 'slightly lower level' (than a graphical method) might be more accurate.

Note: *Instruction list is not directly related to the microprocessor running the program, unlike assembly language or its close relative machine code (both of which are true 'low level' languages), and is likely to have been implemented using a high level language such as C or C++.*

Instruction list may be used as a program entry method in its own right, although this is less common in modern, commercially available PLCs.

17 Function Block Diagrams

A *function block diagram* allows an electronic system to be represented as a *block diagram*, which may then be translated into a text-based Arduino sketch in the usual way.

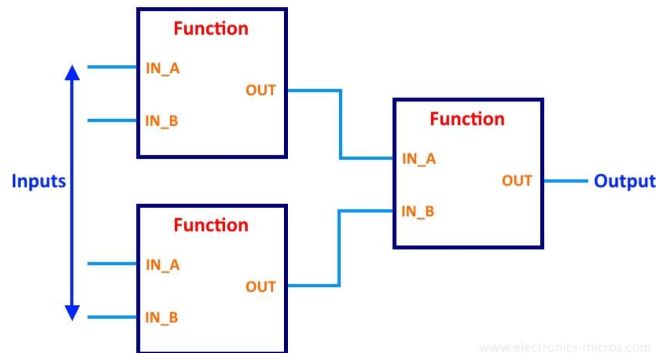


Figure 41. A block diagram can represent many types of electronic system.

Function block diagrams are typically made up of a series of boxes (often rectangular in shape), having inputs at the left and outputs at the right. External inputs are normally shown at the left side of the diagram and outputs at the right. Interconnections between these elements allow signals to progress generally from left to right, although in some cases, feedback signals, may also be present, returning signals back to an earlier stage for further processing.

A variety of function types are supported, including logic gates, comparators, latches, time delays and waveform generators.

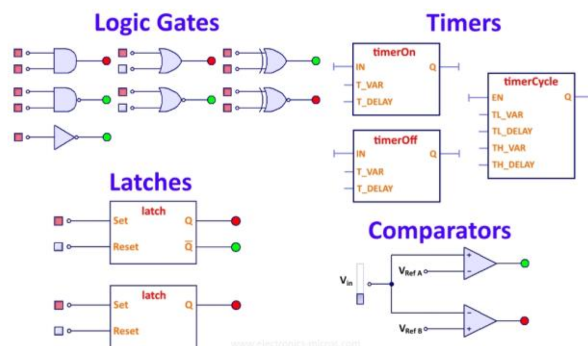


Figure 42. A variety of symbols may be used to represent electronic circuits.

17.1 Constructing Working Systems

Interconnections between blocks may use a variety of *signal types*, which are represented by the computer as an equivalent *data type*. For example, a simple *logic diagram*, consisting of interconnected logic gates, transfers single bit values (0 / 1) between elements. A more complex digital system may require some parameters to be specified as *integers* (*signed* or *unsigned*), or as *floating point* numbers (numbers which can take any value).

Clearly, care must be taken not to link incompatible signals when creating interconnections. However, note that the software is designed to be as 'forgiving' as possible. For example, an analogue input may be directly connected to a PWM output, or to a servo, without problems, and the software will automatically *scale* the values.

A good understanding of how the software 'solves the circuit' is required, to write an equivalent text-based *sketch*. The *Advanced Concepts* section explains how the software uses the *scanValue* variable to solve each rung of a ladder diagram, progressing from left to right across each rung, and from top to bottom in a repeating sequence called the *scan cycle*. In effect, the output of each command is temporarily stored in the *scanValue* variable and then reused as the first input to the next command. This works fine for simple networks, but it may be necessary to store temporary results for later reuse in more complex circuits.

The process of solving a system represented in block diagram form is similar to a ladder diagram. In summary, aim to work from left to right and from top to bottom. Try to allow the *scanValue* variable to automatically transfer signals between system blocks, wherever possible. You can use multiple left to right 'passes' – just like the 'rungs' of a ladder diagram – and temporary results may be stored and later retrieved, if necessary. The following two examples demonstrate the application of this design approach with some relatively simple circuits.

17.2 Application 1: A Simple Alarm

A simple alarm circuit may be constructed using an OR gate and a Set Reset latch, as shown by the following illustration.

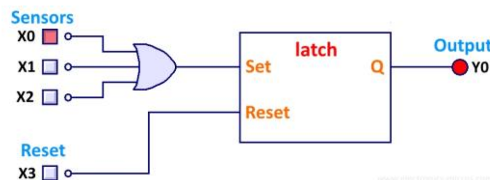


Figure 43. A simple alarm circuit with 3 input sensors, a latched alarm output and a reset input.

The alarm has three sensors, which are connected to a 3-input OR gate. If one or more of the sensors becomes active, this causes the output of the logic gate to go high, which in turn sets the latch and activates the alarm output (*Q*). The alarm will remain active due to the latch, even when the original alarm input is removed, but may be manually cancelled by activating the *Reset* input.

An equivalent sketch is shown below.

```
#include <plcLib.h>

/* Programmable Logic Controller Library for the Arduino and Compatibles

   Simple Alarm - A 3-input alarm circuit with a latched output and manual Reset input

   Connections:
   Input - Sensor 0 - switch connected to input X0 (Arduino pin A0)
   Input - Sensor 1 - switch connected to input X1 (Arduino pin A1)
   Input - Sensor 3 - switch connected to input X2 (Arduino pin A2)
   Input - Reset Alarm - switch connected to input X3 (Arduino pin A3)
   Output - Alarm Output - LED connected to output Y0 (Arduino pin 3)
```

```
   Software and Documentation:
   https://github.com/wditch/plcLib

*/

void setup() {
  setupPLC();         // Setup inputs and outputs
}

void loop() {
  in(X0);             // Read Sensor 0 (Input 0)
  orBit(X1);          // OR with Sensor 1 (Input 1)
  orBit(X2);          // OR with Sensor 2 (Input 2)

                      // Sensor result is used as Set input to latch
  latch(Y0, X3);      // Latch command, Q = Output 0, Reset = Input 3
}
```

Listing 33. *A simple alarm (Source: File > Examples > plcLib > Applications > SimpleAlarm)*

17.3 Application 2: Alarm with Flashing 'Armed' LED

This example is based on the previous application, but uses a cycle timer to produce a flashing *Armed* output when the alarm is active, but not triggered. The circuit diagram is shown below.

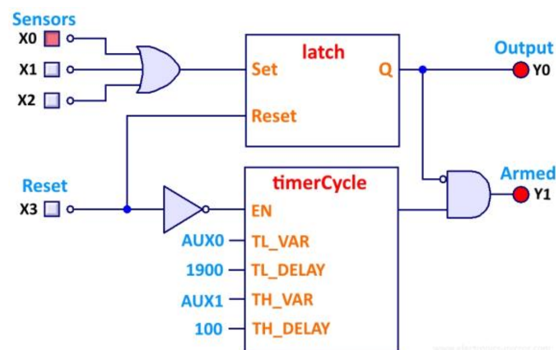


Figure 44. An improved alarm produces a pulse on Y1 when the alarm is armed but not triggered.

Operation of the OR gate and Set Reset latch is the same as the basic alarm described previously. The pulse generator (or *cycle timer*) is enabled when the *Reset* input is inactive (using the Not gate connected to input *X3*) and is configured to produce a brief pulse of 0.1 seconds duration every 2 seconds. The *Q* output of the cycle timer is connected to the *Armed* output (*Y1*) via an AND gate, which will disable the *Armed* output if the main alarm has been triggered. The equivalent software listing is shown below.

```
#include <plcLib.h>

/* Programmable Logic Controller Library for the Arduino and Compatibles

   Alarm with Armed Status LED - 3 input alarm controller with flashing Armed LED

   Connections:
   Input - Sensor 0 - switch connected to input X0 (Arduino pin A0)
   Input - Sensor 1 - switch connected to input X1 (Arduino pin A1)
   Input - Sensor 3 - switch connected to input X2 (Arduino pin A2)
   Input - Reset Alarm - switch connected to input X3 (Arduino pin A3)
   Output - Alarm Output - LED connected to output Y0 (Arduino pin 3)
   Output - Armed Output - LED connected to output Y1 (Arduino pin 5)
```

```
   Software and Documentation:
   https://github.com/wditch/plcLib

*/

// Timer Variables
unsigned long AUX0 = 0;      // Pulse low timer variable
unsigned long AUX1 = 0;      // Pulse high timer variable

void setup() {
  setupPLC();                // Setup inputs and outputs
}

void loop() {

  in(X0);                              // Read Sensor 0
  orBit(X1);                           // OR with Sensor 1
  orBit(X2);                           // OR with Sensor 2

                                       // Set input to latch taken from sensors
  latch(Y0, X3);                       // Latch command, Q = Output 0, Reset = Input 3

  inNot (X3);                          // Enable input (0 = enable)
  timerCycle(AUX0, 1900, AUX1, 100);   // Repeating pulse 1.9 s low, 0.1 s high.
  andNotBit(Y0);                       // Disable armed pulse if alarm is triggered
  out(Y1);                             // Armed pulse on output Y0
}
```

Listing 34. *Alarm with Armed Status LED (Source: File > Examples > plcLib > Applications > AlarmWithArmedStatus)*

17.4 Creating User Defined Function Blocks

You can extend the plcLib software by defining one or more text based *functions* (or *function blocks* to use PLC-specific terminology) 'locally' within a sketch, which may be useful if a required feature is not supported by the software, as standard. Any user created functions may then be *called* from the main program, acting just like built-in commands.

Note: *To create your own custom commands you will need to be reasonably confident with C / C++ programming, and have a basic knowledge of* [Arduino functions]*. You will also need to be familiar with the* [Advanced Concepts] *and* [Using Variables in Programs] *sections of the User Guide.*

The following example calls a *user defined function* to calculate the average value of two different analogue inputs, using the result to control the brightness of an LED.

```
#include <plcLib.h>

/* Programmable Logic Controller Library for the Arduino and Compatibles

   Average - Create a custom function to calculate the mean of two analogue inputs

   Connections:
   Input - Potentiometer connected to analogue input X0 (Arduino pin A0)
   Input - Potentiometer connected to analogue input X1 (Arduino pin A1)
   Output - LED connected to output Y0 (Arduino pin 3)

   Software and Documentation:
   https://github.com/wditch/plcLib

*/

unsigned int reading1;
unsigned int reading2;
extern int scanValue;  // Link to scanValue defined in PLC library file
```

```
void setup() {
  setupPLC();                             // Setup inputs and outputs
}

void loop() {
  reading1 = inAnalog(X0);                // Read analogue Input 0
  reading2 = inAnalog(X1);                // Read analogue Input 1
  scanValue = average(reading1, reading2);  // Set scanValue to average reading
  outPWM(Y0);                             // Send to Output 0
}

                                          // User defined function to calculate
                                          // the average of two values

unsigned int average(unsigned int value1, unsigned int value2) {
  int result;
  result = (value1 + value2) / 2;
  return result;
}
```

Listing 35. *Creating a user defined function block (Source: File > Examples > plcLib > Function Block > Average)*

The above sketch repeatedly reads the two analogue inputs, storing these values in two user defined variables, *reading1* and *reading2*. These are 'passed' to the *average()* function, which finds the mean of *reading1* and *reading2*, returning the result to the main program. This returned result is stored in the *scanValue* variable, which automatically makes the value available as an input to the next command in the *scan cycle*. The actual function definition is found towards the end of the listing.

Note: *The scanCycle variable must be initially defined as an external variable, as it is used internally by the plcLib library software.*

18 Sequential Function Charts

A *Sequential Function Chart* (SFC), allows a PLC to move through a series of steps under program control. The transition from one step to another occurs if the transition condition(s) are met, and each step may optionally produce one or more outputs. As the name suggests, SFCs are normally represented in diagrammatic form, as shown in the following example.

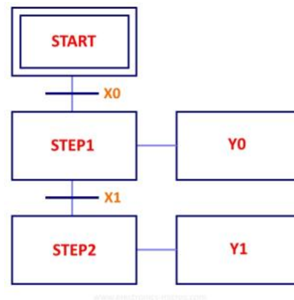


Figure 45. A simple three step program represented as a sequential function chart.

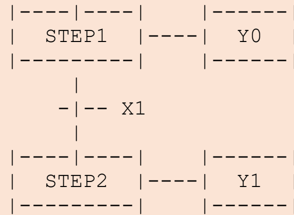
18.1 Creating Sequences

SFCs allow a sequence-based system to be represented in a concise form. Each step in the sequence is shown in a rectangular box at the left, with the first step having a double border. Transition conditions are shown next to horizontal lines between adjacent steps in the sequence. An active step may optionally produce one or more outputs and these are shown in a box to the right of the associated step.

In the above case, the system automatically begins in the *Start-up* step. Progression to *Step 1* occurs when input *X0* is pressed, causing output *Y0* to be displayed. With Y0 displayed, pressing input *X1* causes the system to progress to *Step 2*, activating output *Y1*.

An equivalent sketch is given below.

```
#include <plcLib.h>

/* Programmable Logic Controller Library for the Arduino and Compatibles

   Simple Sequence - A Three step sequence with push button control

   Connections:
   Input - switch connected to input X0 (Arduino pin A0)
   Input - switch connected to input X1 (Arduino pin A1)
   Output - LED connected to output Y0 (Arduino pin 3)
   Output - LED connected to output Y1 (Arduino pin 5)

   Software and Documentation:
   https://github.com/wditch/plcLib


   Sequential Function Chart

        |=========|
        |  START  |
        |=========|
             |
            -|-- X0
             |
```



```
        |----|----|    |------|
        |  STEP1  |----|  Y0  |
        |---------|    |------|
             |
           -|-- X1
             |
        |----|----|    |------|
        |  STEP2  |----|  Y1  |
        |---------|    |------|


   The Start step is active when the system is switched-on or reset
   Press X0 to activate Step 1, displaying Y0
   Next, press X1 to activate Step 2, displaying Y1

*/

                            // Define state names

unsigned int START = 1;     // Start-up state (START = 1 to automatically start here)
unsigned int STEP1 = 0;     // Step 1
unsigned int STEP2 = 0;     // Step 2

void setup() {
  setupPLC();               // Setup inputs and outputs
}

void loop() {
                            // Do state transitions

  in(START);                // Read Start-up state
  andBit(X0);               // AND with Step 1 transition input
  set(STEP1);               // Activate Step 1
  reset(START);             // Cancel Start-up state

  in(STEP1);                // Read Step 1
  andBit(X1);               // AND with Step 2 transition input
  set(STEP2);               // Activate Step 2
  reset(STEP1);             // Cancel Step 1


                            // Display current state

  in(STEP1);
  out(Y0);                  // Send to Output 0

  in(STEP2);
  out(Y1);                  // Send to Output 1

}
```

Listing 36. *A Simple Sequence (Source: File > Examples > plcLib > SequentialFunctionChart > SimpleSwitchSequence)*

The above listing consists of three key sections:

1: Variables are defined to represent each step in the sequence.

```
                            // Define state names

unsigned int START = 1;     // Start-up state (START = 1 to automatically start here)
unsigned int STEP1 = 0;     // Step 1
unsigned int STEP2 = 0;     // Step 2
```

A step is active if its associated variable is equal to 1, and inactive when zero. Hence, initial conditions at power-up or reset are defined based on the above allocated variable values.

2: Steps are activated in a predefined sequence, as each transition condition occurs.

```
                              // Do state transitions

in(START);                    // Read Start-up state
andBit(X0);                   // AND with Step 1 transition input
set(STEP1);                   // Activate Step 1
reset(START);                 // Cancel Start-up state

in(STEP1);                    // Read Step 1
andBit(X1);                   // AND with Step 2 transition input
set(STEP2);                   // Activate Step 2
reset(STEP1);                 // Cancel Step 1
```

Notice that transition conditions are ANDed with the current state, which ensures a transition input is only accepted when its associated step is active. The program should also cancel the previous step when the next step is activated, and this is achieved using the **set()** and **reset()** latch commands, in the above extract.

3: Output(s) are produced based on the active step(s).

```
                              // Display current state

in(STEP1);
out(Y0);                      // Send to Output 0

in(STEP2);
out(Y1);                      // Send to Output 1
```

Note: *This simple program has a single unique output for each step. A more complex system, such as a traffic light controller (see later), may require a custom 'mapping' between steps and outputs – typically implemented with Boolean OR functions.*

The above example may easily be extended to create a repeating sequence, as shown below.

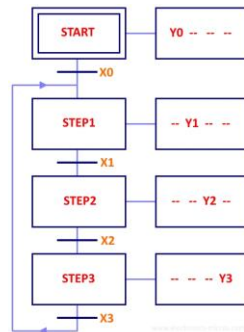


Figure 46. A simple repeating sequence *(Source: File > Examples > plcLib > SequentialFunctionChart > RepeatingSwitchSequence)*

The system powers up in the start-up step with output *Y0* active. Pressing *X0* activates Step 1, then inputs *X1*, *X2* and *X3* cause a repeating progression through Steps 1 – 3, with outputs *Y1* – *Y3* activated, respectively.

Note: *Program listings have been omitted in the remainder of this section in the interests of brevity, but are available from the Examples section in the Arduino IDE, with plcLib installed.*

18.2 Branching and Converging

A *parallel branch* allows several steps to be simultaneously activated, as shown in the following illustration.

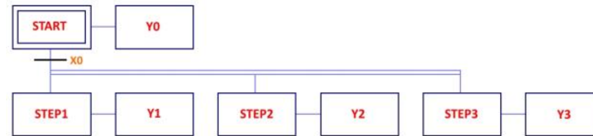


Figure 47. Activating several steps simultaneously with a parallel branch *(Source: File > Examples > plcLib > SequentialFunctionChart > ParallelSwitchBranch)*

The Start step is initially active with output *Y0* enabled. Pressing input *X0* simultaneously activates Steps 1 – 3, displaying outputs *Y1 – Y3*.

A parallel branch may be followed by a *simultaneous convergence*, as shown below.

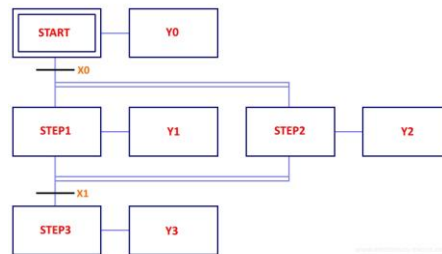


Figure 48. A simultaneous convergence following a parallel branch *(Source: File > Examples > plcLib > SequentialFunctionChart > ParallelSwitchBranchConverge)*

Note: *Any previous steps must be cancelled following a simultaneous convergence.*

A *selective branch* allows a multiple choice decision to be made in which a single step is activated from a range of possible options, as shown below.

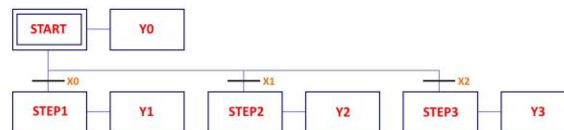


Figure 49. A selective branch activates a single step from a range of options *(Source: File > Examples > plcLib > SequentialFunctionChart > SelectiveSwitchBranch)*

A selective branch may be followed by a convergence, as shown below.

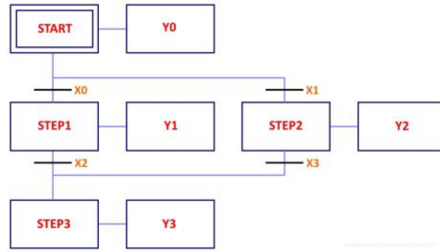


Figure 50. A selective branch followed by a convergence *(Source: File > Examples > plcLib > SequentialFunctionChart > SelectiveSwitchBranchConverge)*

The next section discusses the use of *delays* to produce time-based sequences, and the development of simple SFC-based applications.

19 Developing Timed SFC-based Applications

The previous section introduced basic concepts related to Sequential Function Charts (SFCs), but with transitions between steps initiated by switch presses. This section introduces the use of *on-delay timers* to produce timed transitions between steps, and then develops some simple SFC-based applications.

19.1 Time-base Transitions

The output of an on-delay timer becomes active (i.e. equal to 1) after the input to the timer has been continuously active for a predetermined time, as shown below.

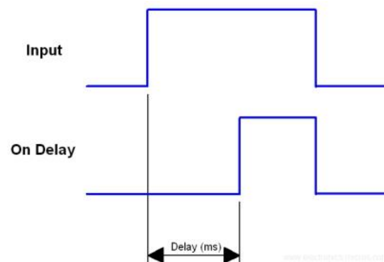


Figure 51. An on-delay produces a delayed activation of an input signal.

An existing switch-based transition may be converted to a time-based equivalent simply by replacing the switch transition with an on-delay timer, as shown in the following program extract.

```
in(START);              // Read Start-up state
timerOn(DELAY0, 2000);  // 2 second delay
set(STEP1);             // Activate Step 1
reset(START);           // Cancel Start-up state
```

In this case, the Start-up state is automatically set to 1 after the computer is switched-on or reset. This enables the timer, which begins counting (using the *DELAY0* user variable). After 2 seconds the timer output changes to a 1, causing the next step in the sequence to be activated, and the previous step cancelled.

A simple time-based sequential function chart is shown below.

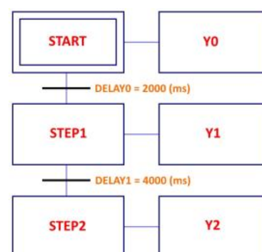


Figure 52. A simple time-based system represented as a sequential function chart.

The equivalent software listing is given below.

```
#include <plcLib.h>

/* Programmable Logic Controller Library for the Arduino and Compatibles

   Simple Timed Sequence - A three step sequence with timed transitions

   Connections:

   Output - LED connected to output Y0 (Arduino pin 3)
   Output - LED connected to output Y1 (Arduino pin 5)
   Output - LED connected to output Y2 (Arduino pin 6)

   Software and Documentation:
   https://github.com/wditch/plcLib


   Sequential Function Chart

        |=========|    |------|
        |  START  |----|  Y0  |
        |=========|    |------|
             |
            -|-- DELAY0 = 2000 (ms)
             |
        |----|----|    |------|
        |  STEP1  |----|  Y1  |
        |---------|    |------|
             |
            -|-- DELAY1 = 4000 (ms)
             |
        |----|----|    |------|
        |  STEP2  |----|  Y2  |
        |---------|    |------|


   The Start step is active when the system is switched-on or reset, displaying Y0
   Step 1 becomes active after 2 seconds, displaying Y1
   After a further 4 seconds Step 2 becomes active, displaying Y2

*/

                            // Define state names

unsigned int START = 1;     // Start-up state (START = 1 to automatically start here)
unsigned int STEP1 = 0;     // Step 1
unsigned int STEP2 = 0;     // Step 2

                            // Define time delay variables

unsigned long DELAY0 = 0;   // Variable to hold elapsed time for Step 0
unsigned long DELAY1 = 0;   // Variable to hold elapsed time for Step 1

void setup() {
  setupPLC();               // Setup inputs and outputs
}

void loop() {
                            // Do state transitions

  in(START);                // Read Start-up state
  timerOn(DELAY0, 2000);    // 2 second delay
  set(STEP1);               // Activate Step 1
  reset(START);             // Cancel Start-up state

  in(STEP1);                // Read Step 1
  timerOn(DELAY1, 4000);    // 4 second delay
  set(STEP2);               // Activate Step 2
  reset(STEP1);             // Cancel Step 1


                            // Display current state
```

```
  in(START);
  out(Y0);                  // Send to Output 0


  in(STEP1);
  out(Y1);                  // Send to Output 1

  in(STEP2);
  out(Y2);                  // Send to Output 2

}
```

Listing 37. *A Simple Timed Sequence (Source: File > Examples > plcLib > SequentialFunctionChart > SimpleTimedSequence)*

Having covered the main elements of sequential function charts, following sections develop some simple SFC-based applications.

19.2 Application 1: Traffic Light Controller Application

The first application demonstrates a time-based sequence for a single set of (UK) traffic lights, including an initial start-up state, as shown by the following sequential function chart.

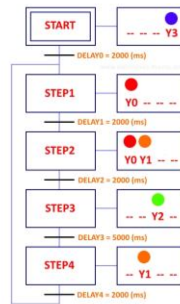
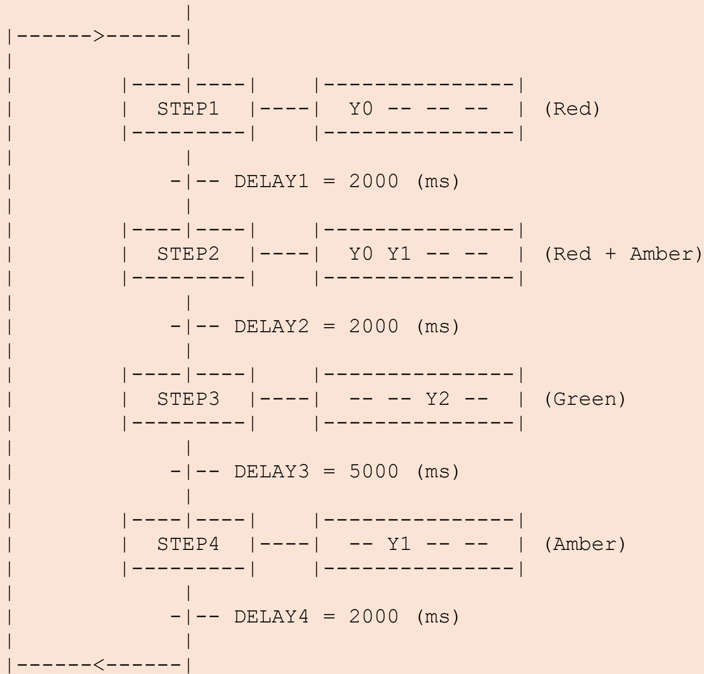


Figure 53. A (UK) traffic light sequence for a single set of lights.

The associated sketch is given below.

```
#include <plcLib.h>

/* Programmable Logic Controller Library for the Arduino and Compatibles

   UK Traffic Lights - Timed Repeating Sequence

   Connections:
   Output - Red LED connected to output Y0 (Arduino pin 3)
   Output - Amber LED connected to output Y1 (Arduino pin 5)
   Output - Green LED connected to output Y2 (Arduino pin 6)
   Output - Blue LED connected to output Y3 (Arduino pin 9)

   Software and Documentation:
   https://github.com/wditch/plcLib



   Sequential Function Chart

         |=========|     |---------------|
         |  START  |----|  -- -- -- Y3  |  (Blue)
         |=========|     |---------------|
              |
            -|-- DELAY0 = 2000 (ms)
```




```
The system begins in the start-up state, proceeding to Red after 2 seconds
Red is displayed for 2 seconds
Red and Amber are displayed for 2 seconds
Green is displayed for 5 seconds
Amber is displayed for 2 seconds , after which operation resumes at the Red step

  Step (Variable)    Red (Y0) Amber (Y1) Green (Y2)  Blue (Y3)    Duration (Variable)
    0   (START)       -         -          -           X            2000 ms  (DELAY0)
    1   (STEP1)       X         -          -           -            2000 ms  (DELAY1)
    2   (STEP2)       X         X          -           -            2000 ms  (DELAY2)
    3   (STEP3)       -         -          X           -            5000 ms  (DELAY3)
    4   (STEP4)       -         X          -           -            2000 ms  (DELAY4)
    1
    2    etc.

*/
                            // Define step names

unsigned int START = 1;     // Start-up state (START = 1 to automatically start here)
unsigned int STEP1 = 0;     // Step 1
unsigned int STEP2 = 0;     // Step 2
unsigned int STEP3 = 0;     // Step 3
unsigned int STEP4 = 0;     // Step 4

                            // Define time delay variables

unsigned long DELAY0 = 0;   // Variable to hold elapsed time for Step 0
unsigned long DELAY1 = 0;   // Variable to hold elapsed time for Step 1
unsigned long DELAY2 = 0;   // Variable to hold elapsed time for Step 2
unsigned long DELAY3 = 0;   // Variable to hold elapsed time for Step 3
unsigned long DELAY4 = 0;   // Variable to hold elapsed time for Step 4

void setup() {
  setupPLC();               // Setup inputs and outputs
}

void loop() {
                            // Do timed step transitions

  in(START);                // Read start-up step
  timerOn(DELAY0, 2000);    // 2 second delay
  set(STEP1);               // Activate Step 1
```

```
  reset(START);              // Cancel Step 0

  in(STEP1);                 // Read Step 1
  timerOn(DELAY1, 2000);     // 2 second delay
  set(STEP2);                // Activate Step 2
  reset(STEP1);              // Cancel Step 1

  in(STEP2);                 // Read Step 2
  timerOn(DELAY2, 2000);     // 2 second delay
  set(STEP3);                // Activate Step 3
  reset(STEP2);              // Cancel Step 2

  in(STEP3);                 // Read Step 3
  timerOn(DELAY3, 5000);     // 5 second delay
  set(STEP4);                // Activate Step 4
  reset(STEP3);              // Cancel Step 3

  in(STEP4);                 // Read Step 4
  timerOn(DELAY4, 2000);     // 2 second delay
  set(STEP1);                // Activate Step 1 (again)
  reset(STEP4);              // Cancel Step 4


                             // Display outputs according to table

                             // Step  Red  Amber Green  Blue
                             //   0     -     -     -     x
                             //   1     x     -     -     -
                             //   2     x     x     -     -
                             //   3     -     -     x     -
                             //   4     -     x     -     -

                             // Display Blue if step = 0
  in(START);                 // Read Step 0
  out(Y3);                   // Send to Output 3 (Blue)

                             // Display Red if step = 1 OR 2
  in(STEP1);                 // Read Step 1
  orBit(STEP2);              // OR with Step 2
  out(Y0);                   // Send to Output 0 (Red)

                             // Display Amber if step = 2 OR 4
  in(STEP2);                 // Read Step 2
  orBit(STEP4);              // OR with Step 4
  out(Y1);                   // Send to Output 1 (Amber)

                             // Display Green if step = 3
  in(STEP3);                 // Read Step 3
  out(Y2);                   // Send to Output 2 (Green)

}
```

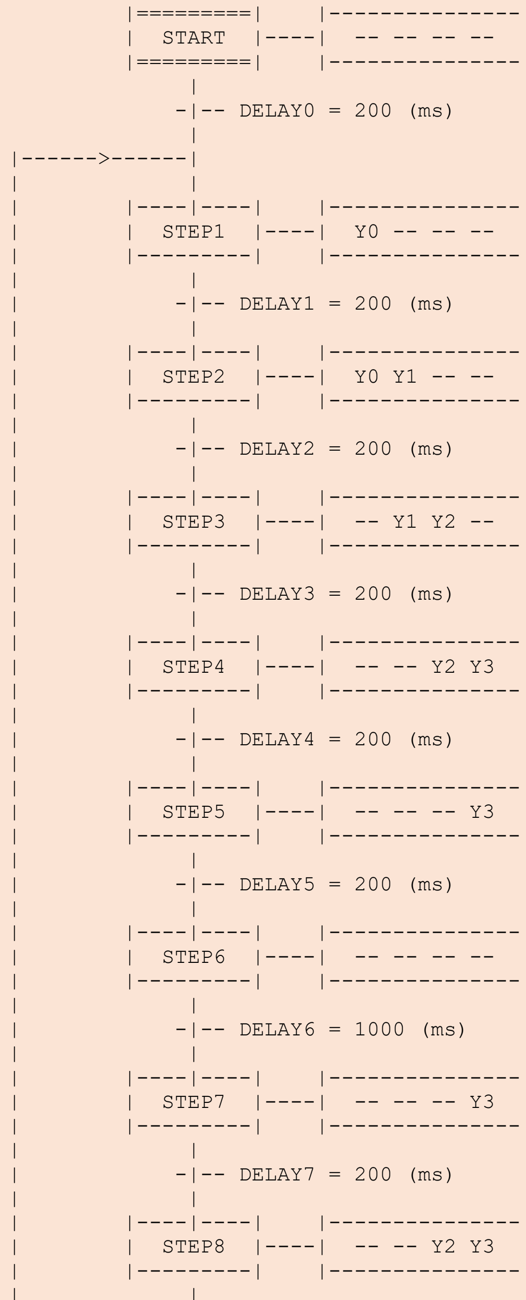
Listing 38. *UK Traffic Lights (Source location: File > Examples > plcLib > Applications> SingleTrafficLight)*

Notice the use of combinational logic instructions toward the end of the above listing, to create a 'mapping' between sequence steps and the outputs generated. Hence the Red LED is displayed if Step 1 or 2 is active, Amber is shown during Steps 2 or 4, and so on. This same technique is used in the next example.

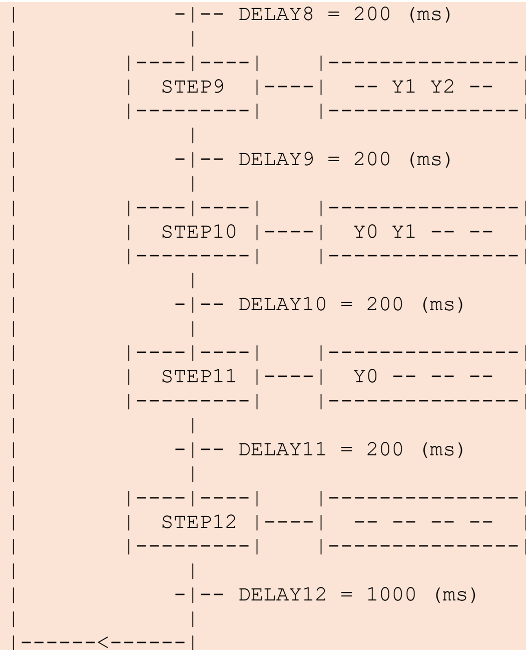
19.3 Application 2: Running Light Display

A simple running light display sequence gives the visual impression of moving one or more illuminated LEDs alternately to the left and right, as illustrated by the following sketch.

```
#include <plcLib.h>

/* Programmable Logic Controller Library for the Arduino and Compatibles

   Running Light Display - Timed Repeating Sequence

   Connections:
   Output - Red LED connected to output Y0 (Arduino pin 3)
   Output - Amber LED connected to output Y1 (Arduino pin 5)
   Output - Green LED connected to output Y2 (Arduino pin 6)
   Output - Blue LED connected to output Y3 (Arduino pin 9)

   Software and Documentation:
   https://github.com/wditch/plcLib

   Sequential Function Chart
```



```
         |=========|    |---------------|
         |  START  |----|  -- -- -- --  |
         |=========|    |---------------|
              |
             -|-- DELAY0 = 200 (ms)
              |
|------>------|
|             |
|        |----|----|    |---------------|
|        |  STEP1  |----|  Y0 -- -- --  |
|        |---------|    |---------------|
|             |
|            -|-- DELAY1 = 200 (ms)
|             |
|        |----|----|    |---------------|
|        |  STEP2  |----|  Y0 Y1 -- --  |
|        |---------|    |---------------|
|             |
|            -|-- DELAY2 = 200 (ms)
|             |
|        |----|----|    |---------------|
|        |  STEP3  |----|  -- Y1 Y2 --  |
|        |---------|    |---------------|
|             |
|            -|-- DELAY3 = 200 (ms)
|             |
|        |----|----|    |---------------|
|        |  STEP4  |----|  -- -- Y2 Y3  |
|        |---------|    |---------------|
|             |
|            -|-- DELAY4 = 200 (ms)
|             |
|        |----|----|    |---------------|
|        |  STEP5  |----|  -- -- -- Y3  |
|        |---------|    |---------------|
|             |
|            -|-- DELAY5 = 200 (ms)
|             |
|        |----|----|    |---------------|
|        |  STEP6  |----|  -- -- -- --  |
|        |---------|    |---------------|
|             |
|            -|-- DELAY6 = 1000 (ms)
|             |
|        |----|----|    |---------------|
|        |  STEP7  |----|  -- -- -- Y3  |
|        |---------|    |---------------|
|             |
|            -|-- DELAY7 = 200 (ms)
|             |
|        |----|----|    |---------------|
|        |  STEP8  |----|  -- -- Y2 Y3  |
|        |---------|    |---------------|
|             |
```



```
|            -|-- DELAY8 = 200 (ms)
|             |
|        |----|----|    |---------------|
|        |  STEP9  |----|  -- Y1 Y2 --  |
|        |---------|    |---------------|
|             |
|            -|-- DELAY9 = 200 (ms)
|             |
|        |----|----|    |---------------|
|        |  STEP10 |----|  Y0 Y1 -- --  |
|        |---------|    |---------------|
|             |
|            -|-- DELAY10 = 200 (ms)
|             |
|        |----|----|    |---------------|
|        |  STEP11 |----|  Y0 -- -- --  |
|        |---------|    |---------------|
|             |
|            -|-- DELAY11 = 200 (ms)
|             |
|        |----|----|    |---------------|
|        |  STEP12 |----|  -- -- -- --  |
|        |---------|    |---------------|
|             |
|            -|-- DELAY12 = 1000 (ms)
|             |
|------<------|

The system begins in the Start-up state, proceeding to Step 1 after 0.2 seconds
Steps 1 - 12 are executed in a repeating sequence with delay controlled transitions
Outputs Y0 - Y3 are displayed according to the following table

  Step (Variable)    Y0    Y1    Y2    Y3    Duration (Variable)
   0    (START)      -     -     -     -      200 ms   (DELAY0)
   1    (STEP1)      x     -     -     -      200 ms   (DELAY1)
   2    (STEP2)      x     x     -     -      200 ms   (DELAY2)
   3    (STEP3)      -     x     x     -      200 ms   (DELAY3)
   4    (STEP4)      -     -     x     x      200 ms   (DELAY4)
   5    (STEP5)      -     -     -     x      200 ms   (DELAY5)
   6    (STEP6)      -     -     -     -      1000 ms  (DELAY6)
   7    (STEP7)      -     -     -     x      200 ms   (DELAY7)
   8    (STEP8)      -     -     x     x      200 ms   (DELAY8)
   9    (STEP9)      -     x     x     -      200 ms   (DELAY9)
   10   (STEP10)     x     x     -     -      200 ms   (DELAY10)
   11   (STEP11)     x     -     -     -      200 ms   (DELAY11)
   12   (STEP12)     -     -     -     -      1000 ms  (DELAY12)

*/
                            // Define step names for Sequential Function Chart program

unsigned int START = 1;     // Start-up state (START = 1 to automatically start here)
unsigned int STEP1 = 0;     // Step 1
unsigned int STEP2 = 0;     // Step 2
unsigned int STEP3 = 0;     // Step 3
unsigned int STEP4 = 0;     // Step 4
unsigned int STEP5 = 0;     // Step 5
unsigned int STEP6 = 0;     // Step 6
unsigned int STEP7 = 0;     // Step 7
unsigned int STEP8 = 0;     // Step 8
unsigned int STEP9 = 0;     // Step 9
unsigned int STEP10 = 0;    // Step 10
unsigned int STEP11 = 0;    // Step 11
unsigned int STEP12 = 0;    // Step 12


                            // Define time delay variables

unsigned long DELAY0 = 0;   // Variable to hold elapsed time for Step 0
unsigned long DELAY1 = 0;   // Variable to hold elapsed time for Step 1
unsigned long DELAY2 = 0;   // Variable to hold elapsed time for Step 2
unsigned long DELAY3 = 0;   // Variable to hold elapsed time for Step 3
```

```
unsigned long DELAY4 = 0;   // Variable to hold elapsed time for Step 4
unsigned long DELAY5 = 0;   // Variable to hold elapsed time for Step 5
unsigned long DELAY6 = 0;   // Variable to hold elapsed time for Step 6
unsigned long DELAY7 = 0;   // Variable to hold elapsed time for Step 7
unsigned long DELAY8 = 0;   // Variable to hold elapsed time for Step 8
unsigned long DELAY9 = 0;   // Variable to hold elapsed time for Step 9
unsigned long DELAY10 = 0;   // Variable to hold elapsed time for Step 10
unsigned long DELAY11 = 0;   // Variable to hold elapsed time for Step 11
unsigned long DELAY12 = 0;   // Variable to hold elapsed time for Step 12

void setup() {
  setupPLC();               // Setup inputs and outputs
}

void loop() {
                            // Do timed step transitions

  in(START);                // Read startup step - Step 0
  timerOn(DELAY0, 200);     // 0.2 second delay
  set(STEP1);               // Activate Step 1
  reset(START);             // Cancel Step 0

  in(STEP1);                // Read Step 1
  timerOn(DELAY1, 200);     // 0.2 second delay
  set(STEP2);               // Activate Step 2
  reset(STEP1);             // Cancel Step 1

  in(STEP2);                // Read Step 2
  timerOn(DELAY2, 200);     // 0.2 second delay
  set(STEP3);               // Activate Step 3
  reset(STEP2);             // Cancel Step 2

  in(STEP3);                // Read Step 3
  timerOn(DELAY3, 200);     // 0.2 second delay
  set(STEP4);               // Activate Step 4
  reset(STEP3);             // Cancel Step 3

  in(STEP4);                // Read Step 4
  timerOn(DELAY4, 200);     // 0.2 second delay
  set(STEP5);               // Activate Step 5
  reset(STEP4);             // Cancel Step 4

  in(STEP5);                // Read Step 5
  timerOn(DELAY5, 200);     // 0.2 second delay
  set(STEP6);               // Activate Step 6
  reset(STEP5);             // Cancel Step 5

  in(STEP6);                // Read Step 6
  timerOn(DELAY6, 1000);    // 1 second delay
  set(STEP7);               // Activate Step 7
  reset(STEP6);             // Cancel Step 6

  in(STEP7);                // Read Step 7
  timerOn(DELAY7, 200);     // 0.2 second delay
  set(STEP8);               // Activate Step 8
  reset(STEP7);             // Cancel Step 7

  in(STEP8);                // Read Step 8
  timerOn(DELAY8, 200);     // 0.2 second delay
  set(STEP9);               // Activate Step 9
  reset(STEP8);             // Cancel Step 8

  in(STEP9);                // Read Step 9
  timerOn(DELAY9, 200);     // 0.2 second delay
  set(STEP10);              // Activate Step 10
  reset(STEP9);             // Cancel Step 9

  in(STEP10);               // Read Step 10
  timerOn(DELAY10, 200);    // 0.2 second delay
  set(STEP11);              // Activate Step 11
  reset(STEP10);            // Cancel Step 10
```

```
  in(STEP11);                 // Read Step 11
  timerOn(DELAY11, 200);      // 0.2 second delay
  set(STEP12);                // Activate Step 12
  reset(STEP11);              // Cancel Step 11

  in(STEP12);                 // Read Step 12
  timerOn(DELAY12, 1000);     // 1 second delay
  set(STEP1);                 // Activate Step 1
  reset(STEP12);              // Cancel Step 12


                              // Display outputs according to table

                              // Display Y0 if step = 1, 2, 10 or 11
  in(STEP1);                  // Read Step 1
  orBit(STEP2);               // OR with Step 2
  orBit(STEP10);              // OR with Step 10
  orBit(STEP11);              // OR with Step 11
  out(Y0);                    // Send to Output 0 (Red)

                              // Display Y1 if step = 2, 3, 9 or 10
  in(STEP2);                  // Read Step 2
  orBit(STEP3);               // OR with Step 3
  orBit(STEP9);               // OR with Step 9
  orBit(STEP10);              // OR with Step 10
  out(Y1);                    // Send to Output 1 (Amber)

                              // Display Y2 if step = 3, 4, 8 or 9
  in(STEP3);                  // Read Step 3
  orBit(STEP4);               // OR with Step 4
  orBit(STEP8);               // OR with Step 8
  orBit(STEP9);               // OR with Step 9
  out(Y2);                    // Send to Output 2 (Green)

                              // Display Y3 if step = 4, 5, 7 or 8
  in(STEP4);                  // Read Step 4
  orBit(STEP5);               // OR with Step 5
  orBit(STEP7);               // OR with Step 7
  orBit(STEP8);               // OR with Step 8
  out(Y3);                    // Send to Output 3 (Blue)

}
```

Listing 39. *Running Light Display (Source: File > Examples > plcLib > Applications > RunningLightDisplay)*

20 Structured Text

Software developed to use the PLC library may incorporate a wide range of standard program statements, such as variable assignments, calculations and user defined functions, plus a variety of program structures, including decisions and loops.

Possibilities for coding are largely limited by your imagination – and knowledge of C / C++ programming of course. There are a few caveats however, so a good understanding of the internal operation of the plcLib software, and its use of variables, is recommended before attempting to develop your own *structured text* programs.

Note: *Lots of information related to Arduino programming is available online, including the* Arduino Reference *and* Examples *pages.*

20.1 Using Program Structures

You can use a variety of program structures in your sketches, but care should be taken to avoid halting the PLC scan cycle, as this will in turn halt the processing of any other tasks.

Available program structures include: -

- ***if*** – conditional execution of a code section
- ***if ... else*** – conditional execution of a main section or an alternative, based on a tested logical condition
- ***do ... while*** – repeat a section while a condition is true. (The test is at the end, so this type of loop always runs at least once, even if the test condition is initially false.)
- ***while*** – repeat a section while a condition is true. (The test is at the start, so this type of loop may not run at all if the test is initially false.)
- ***for*** – repeat a section a fixed number of times, with the loop variable progressing through a predetermined series of values.
- ***Switch case*** – execute a single section, chosen from a series of options

As an example, the following sketch first reads digital input *X0*. It then uses an *if* statement, based on the state of the digital input, to conditionally read an analogue input, using this result to control the brightness of an LED.

```
#include <plcLib.h>

/* Programmable Logic Controller Library for the Arduino and Compatibles

   IF - Conditional control of a PWM output

   LED brightness may be varied using the potentiometer, but only when X0 is pressed.
   Previous PWM output is displayed otherwise.

   Connections:
   Input - switch connected to input X0 (Arduino pin A0)
   Input - potentiometer connected to input X1 (Arduino pin A1)
   Output - LED connected to output Y0 (Arduino pin 3)
```

```
   Software and Documentation:
   https://github.com/wditch/plcLib

*/

unsigned int myVar = 0;   // Create a user defined variable and set initial value

void setup() {
  setupPLC();             // Setup inputs and outputs
}

void loop() {
  myVar = in(X0);         // Read digital Input 0, storing result in user variable

  if (myVar == 1) {       // Vary PWM, if enabled
    inAnalog(X1);         // Read Analogue Input 1
    outPWM(Y0);           // Send to Output 0 as PWM waveform
  }
}
```

Listing 40. *Using an IF command to conditionally control a PWM output (Source: File > Examples > plcLib > StructuredText > If)*

The above sketch allows the brightness of the LED to be varied, but only when the switch is pressed. At other times, the previously set PWM value is used, in effect acting as a 'programmable' light dimmer.

A range of other examples are available from the same directory as the above example, with each one illustrating a different control structure.

Note: *The C / C++ programming language used by the Arduino (and hence in the examples) is similar, but not identical, to the Pascal-style programming language described by the appropriate standard (IEC 61131-3).*

21 Advanced Concepts

This section explains the internal operation of the plcLib software. An understanding of these concepts is not necessary for basic programming, but may be useful when developing more advanced applications, or for understanding why things aren't quite working as you expected.

21.1 How the Software Works

A PLC operates by repeatedly reading inputs, performing calculations, and then sending the results to the outputs. This process is known as the *scan cycle*. A typical ladder diagram based application is 'scanned' one rung at a time, from left to right, starting at the top rung and working progressively downwards. This process repeats continuously.

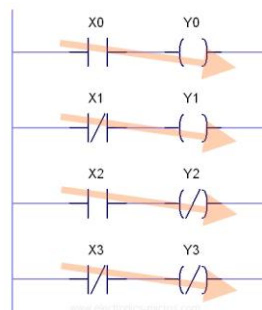


Figure 54. Each row is scanned in a repeating process as part of the scan cycle.

Each rung of the ladder may be thought of as a *parallel process*, which receives its own share of the processor time as the scan cycle repeatedly executes. Hence PLC-based applications demonstrate simple parallel processing capabilities, but without the need to resort to advanced programming techniques.

For basic operation, the PLC software uses a single variable called *scanValue* to hold its running calculation result as each branch is solved. Consider the following code snippet to see how this works for single bit values:

```
void loop() {
  in(X0);       // Read Input 0
  out(Y0);      // Send to Output 0
}
```

The single bit input command **in(X0)** reads digital input pin *X0* and stores its result in the *scanValue* variable as 1 or 0. A subsequent bit output command **out(Y0)** simply reads the *scanValue* variable and sends this value to digital output pin *Y0*. This process repeats as each rung of the ladder diagram is calculated, with *scanValue* repeatedly initialised, updated and then discarded as the ladder logic program executes.

The process is similar for an analogue input, which is read from an *analogue to digital converter* as a 10-bit value in the range 0-1023 using the **inAnalog()** command – as illustrated by the following code snippet.

```
void loop() {
  inAnalog(X0);     // Read Analogue Input 0
  outPWM(Y0);       // Send to Output 0 as PWM waveform
}
```

The same *scanValue* variable is used to hold this analogue value, which is not a problem as the variable is actually an unsigned integer variable type. The plcLib software automatically handles the scaling of any 'analogue' outputs, so an **outPWM()** command is scaled to be in the range 0-255 (8-bit binary), while an **outServo()** command would use a value scaled in the range 0-179 (representing 180° of rotation).

The next section explains how the above mentioned *scanValue* variable may be used, and discusses possible applications of user-created variables in sketches.

21.2 Using Variables in Programs

This section considers some applications of variables in user sketches, beginning with the built-in *scanValue* variable, which keeps track of the current calculation result as the scan cycle is executed. Creation of user defined variables is then discussed, leading finally to the application of user variables to solve complex, multiple branch logic circuits.

21.2.1 Using the scanValue Variable

It was seen previously that the *scanValue* is used internally by the plcLib software as the PLC *scan cycle* is repeatedly executed. However, this variable is not directly available in user programs, as it is only defined inside the plcLib header file (it is said to be 'out of scope' inside your sketch). You can optionally make *scanValue* available by defining it as an *external variable*, as shown in the following example.

```
#include <Servo.h>
#include <plcLib.h>

/* Programmable Logic Controller Library for the Arduino and Compatibles

   Single Servo - Controlling the position of a servo using a potentiometer

   Connections:
   Input - potentiometer connected to input X0 (Arduino pin A0)
   Output - servo connected to output Y0 (Arduino pin 3)

   Software and Documentation:
   https://github.com/wditch/plcLib

*/

Servo servo1;            // Create a servo object to control a single servo

extern int scanValue;   // Link to scanValue defined in PLC library file

void setup() {
  setupPLC();            // Setup inputs and outputs
  servo1.attach(Y0);     // Attaches Servo 1 to Output 0
}

void loop() {
  inAnalog(X0);          // Read potentiometer connected to input 0
  outServo(servo1);      // Output to Servo 1 (connected to Output 0)
```

```
}

// The outServo function is defined locally
int outServo(Servo &ServoObj) {
  ServoObj.write(map(scanValue, 0, 1023, 0, 179));
  return(scanValue);
}
```

Listing 41. *Making the scanValue variable available to multiple libraries (Source: File > Examples > plcLib > InputOutput > ServoSingle)*

As an alternative to the above approach, plcLib instructions may optionally return the current *scanValue* value to a previously defined user variable (which is only accurate at that specific point in the scan cycle). The content of a user defined variable may also be used as an 'input' to a plcLib command, in place of direct input or output from a named pin. The following example illustrates some of the possibilities.

```
#include <plcLib.h>

/* Programmable Logic Controller Library for the Arduino and Compatibles

   Latch Command with Variables - Latch command using variables to hold temporary values

   Connections:
   Input - Set - switch connected to input X0 (Arduino pin A0)
   Input - Reset - Switch connected to input X1 (Arduino pin A1)
   Output - Q Output - LED connected to output Y0 (Arduino pin 3)

   Software and Documentation:
   https://github.com/wditch/plcLib

*/

// Temporary Variables:
unsigned int AUX0;        // Latch output variable
unsigned int AUX1;        // Latch reset variable

void setup() {
  setupPLC();             // Setup inputs and outputs
}

void loop() {
  AUX1 = in(X1);          // AUX1 (Reset) controlled by input X1

  in(X0);                 // Read switch connected to Input 0 (Set input - Input 0)
  latch(AUX0, AUX1);     // Latch command, Output = AUX0, Reset = AUX1 (Input 1)

  in(AUX0);               // Send Q output to output Y0
  out(Y0);
}
```

Listing 42. *Returning the scanValue to a user defined variable (Source: File > Examples > plcLib > Variables > LachCommandVariables)*

Things to notice above are the saving of the *scanValue* variable in a user defined variable (**AUX1 = in(X1);**), and also the use of variable names in place of pin names in other plcLib commands.

21.3 Working with Custom Variables

When referring to names such as *X0*, *X1*, *Y0*, *AUX0*, *AUX1*, in the previous listing. you might wonder how does the software 'know' whether you are referring to a pin name or the content of a user defined variable? The answer is that all pin names (*X0*, *X1* and *Y0*) are defined internally by the plcLib software

as *signed integer* variables, while the user defined variables in the above program (*AUX0* and *AUX1*) are of an *unsigned* variable type. The plcLib software actually has at least two versions of each command, and the compiler picks the appropriate version based on the supplied variable type (this is called *function overloading*).

Just remember to specify your user defined integer-type variables as either *unsigned integer* (16-bit) or *unsigned long* (32-bit) to avoid confusion between user variables and pin names.

21.4 Using Variables with Complex Logic Circuits

A possible application of user variables is in solving complex Boolean logic circuits, as shown in the following example sketch.

```
#include <plcLib.h>

/* Programmable Logic Controller Library for the Arduino and Compatibles

   Complex Logic - Solving multiple branch Boolean Logic circuits by using
                   variable(s) to hold temporary calculation results

   Connections:
   Input - switch connected to input X0 (Arduino pin A0)
   Input - switch connected to input X1 (Arduino pin A1)
   Input - switch connected to input X2 (Arduino pin A2)
   Input - switch connected to input X3 (Arduino pin A3)
   Output - LED connected to output Y0 (Arduino pin 3)

   Equivalent Ladder Logic Circuit:
   |                                |
   |         X0    X1               |
   |     |---| |---|/|---|          |
   |     |               |    Y0    |
   |-----|               |---(  )---|
   |     |   X2    X3    |          |
   |     |---| |---| |---|          |
   |                                |
   |                                |

   Software and Documentation:
   https://github.com/wditch/plcLib

*/

unsigned int AUX0;  // AUX0 variable holds top rung temporary result

void setup() {
  setupPLC();       // Setup inputs and outputs
}

void loop() {
                    // Solve first branch
  in(X0);           // Read Input 0
  andNotBit(X1);    // AND with inverted Input 1
  out(AUX0);        // Use auxiliary variable AUX0 to store first branch result

                    // Solve second branch
  in(X2);           // Read Input 2
  andBit(X3);       // AND with Input 3
  orBit(AUX0);      // OR with result from first branch (AUX0)
  out(Y0);          // Send result to Output 0
}
```

Listing 43. *Solving complex logic with user variables (Source: File > Examples > plcLib > Variables > ComplexLogic)*

The above example first solves the upper branch of the logic circuit and then stores (or 'outputs') this as a temporary result to a user defined variable – **out(AUX0);**. (This line could have been omitted completely by rewriting the preceding line as **AUX0 = andNotBit(X1);**.) The second branch is then solved and the result from the first branch ORed with the previous temporary result – **orBit(AUX0);**.

An alternative to the 'user variable' approach discussed above is the application of *block logic* commands, which is discussed in the next section.

22 Stack-based Storage and Logic

Version 1.0 of the plcLib software adds the ability to create and use a software-based *stack* for temporary storage and retrieval of single-bit values. This capability may be combined with *block logic* commands, to simplify the solution of complex networks based on *Boolean algebra* – but without the need to create individual user variables for temporary storage, as discussed previously.

A *stack* is a special area of memory which may be used for temporary data storage and retrieval. Information is stored by being *pushed* onto the stack and is later retrieved by *popping* from the stack. The most recently stored information is always the first to be removed, so the stack acts as a *last-in, first-out* store.

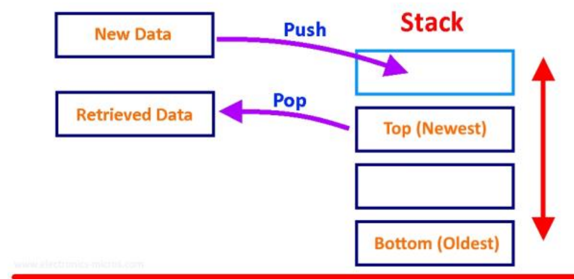


Figure 55. Using a stack as a *last-in first-out* data store.

A useful analogy to aid understanding of stack-based data storage and retrieval is a pile of plates, where each plate represents a single piece of information. Storing data is equivalent to adding a new plate to the top of the pile, which causes the 'stack' of plates to grow higher. Conversely, information is retrieved by removing a plate. The most recently added plate will always be at the top, and the oldest at the bottom.

The first step when writing a stack-based sketch is to use the **Stack** command to create a stack object. (For example, the command **Stack stack1;** creates a stack called *stack1*, capable of holding up to 32 single-bit numbers.) Values may be added or removed from the stack by using the **push()** or **pop()** methods of the previously created stack object. The following sketch demonstrates the use of the stack to store and subsequently retrieve a series of single bit values.

```
#include <plcLib.h>

/* Programmable Logic Controller Library for the Arduino and Compatibles

   Push and Pop values from a single-bit software stack

   Connections:
   Input - switch connected to input X0 (Arduino pin A0)
   Input - switch connected to input X1 (Arduino pin A1)
   Input - switch connected to input X2 (Arduino pin A2)
   Input - switch connected to input X3 (Arduino pin A3)
   Output - LED connected to output Y0 (Arduino pin 3)
   Output - LED connected to output Y1 (Arduino pin 5)
   Output - LED connected to output Y2 (Arduino pin 6)
   Output - LED connected to output Y3 (Arduino pin 9)

   Software and Documentation:
   https://github.com/wditch/plcLib

*/
```

```
Stack stack1;     // Create a single-bit stack with 32 levels

void setup() {
  setupPLC();     // Setup inputs and outputs
}

void loop() {

                  // Push 4 values onto the stack
                  // 1) X0, 2) X1, 3) X2, 4) X3

  in(X0);         // Read Input 0
  stack1.push();  // Push X0 onto the stack

  in(X1);         // Read Input 1
  stack1.push();  // Push X1 onto the stack

  in(X2);         // Read Input 2
  stack1.push();  // Push X2 onto the stack

  in(X3);         // Read Input 3
  stack1.push();  // Push X3 onto the stack

                  // Remove 4 values from the stack and
                  // send to outputs in reverse order
                  // 1) X3->Y0, 2) X2->Y1, 3) X1->Y2, 4) X0->Y3
                  // (a stack is a last-in first-out or LIFO store)

  stack1.pop();   // Remove X3 value from the stack
  out(Y0);        // Send to Output 0

  stack1.pop();   // Remove X2 value from the stack
  out(Y1);        // Send to Output 1

  stack1.pop();   // Remove X1 value from the stack
  out(Y2);        // Send to Output 2

  stack1.pop();   // Remove X0 value from the stack
  out(Y3);        // Send to Output 3
}
```

Listing 44. *Pushing and Popping values from a single-bit software stack (Source: File > Examples > plcLib > Stack > PushPop)*

The ability to store temporary calculation results on the stack may be used to simplify the solution of complex logic networks. Options are available to combine parallel or series branches by using block-based logical AND and OR operations, as discussed in the next section.

22.1 Block Logic Operations

A logic network consisting of two parallel branches may be solved by first calculating the upper branch, then saving this intermediate result on the stack. The second branch may then be solved and combined with the earlier result, using the **orBlock()** method of the stack object (which also removes the previous result from the stack).

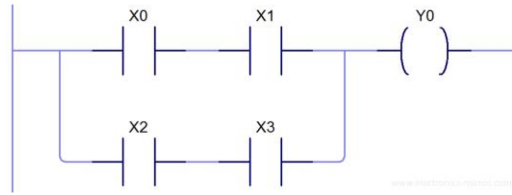


Figure 56. A Block-OR operation may be used to combine a parallel branch with a calculation result previously stored on the stack.

The following sketch demonstrates the approach.

```
#include <plcLib.h>

/* Programmable Logic Controller Library for the Arduino and Compatibles

   Logical OR of two parallel switch branches using Block OR instruction

                            X0          X1
                       ----|  |--------|  |----
                       |                      |     Y0
                  -----                       -----(  )-----
                       |    X2          X3    |
                       ----|  |--------|  |----

   Connections:
   Input - switch connected to input X0 (Arduino pin A0)
   Input - switch connected to input X1 (Arduino pin A1)
   Input - switch connected to input X2 (Arduino pin A2)
   Input - switch connected to input X3 (Arduino pin A3)
   Output - LED connected to output Y0 (Arduino pin 3)


   Software and Documentation:
   https://github.com/wditch/plcLib

*/

Stack stack1;         // Create a single-bit stack with 32 levels

void setup() {
  setupPLC();         // Setup inputs and outputs
}

void loop() {

                      // Calculate First Branch
  in(X0);             // Read switch connected to Input 0
  andBit(X1);         // Logical AND with Input 1
  stack1.push();      // Push temporary result onto the stack

                      // Calculate second branch
  in(X2);             // Read switch connected to Input 2
  andBit(X3);         // Logical AND with Input 3

  stack1.orBlock();   // Merge branches using Block OR
  out(Y0);            // Send result to Output 0

}
```

Listing 45. *Performing a logical OR of two parallel switch branches using Block OR instruction (Source: File > Examples > plcLib > Stack > OrBlock)*

A similar technique may be applied with series connections of switch groups, which may be combined using an AND Block.

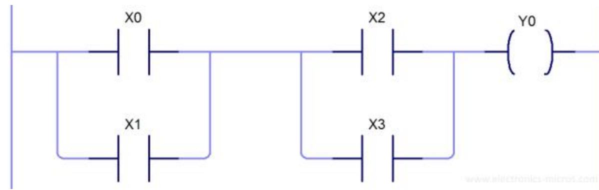


Figure 57. A *Block-AND* may be used to solve a complex network consisting of series elements.

The following example first calculates the result of the switch group at the left, which is stored as an intermediate result on the stack. The right hand block is then solved and combined with the earlier result by using the **andBlock()** method of the stack object.

```
#include <plcLib.h>

/* Programmable Logic Controller Library for the Arduino and Compatibles

   Logical AND of two series switch groups using Block AND instruction

                    X0                      X2
                ----|  |----            ----|  |----
                |          |            |          |      Y0
            -----          -------------           -----(  )-----
                |    X1    |            |    X3    |
                ----|  |----            ----|  |----

   Connections:
   Input - switch connected to input X0 (Arduino pin A0)
   Input - switch connected to input X1 (Arduino pin A1)
   Input - switch connected to input X2 (Arduino pin A2)
   Input - switch connected to input X3 (Arduino pin A3)
   Output - LED connected to output Y0 (Arduino pin 3)


   Software and Documentation:
   https://github.com/wditch/plcLib

*/

Stack stack1;         // Create a single-bit stack with 32 levels

void setup() {
  setupPLC();         // Setup inputs and outputs
}

void loop() {

                      // Calculate First Branch
  in(X0);             // Read switch connected to Input 0
  orBit(X1);          // Logical OR with Input 1
  stack1.push();      // Push temporary result onto the stack

                      // Calculate second branch
  in(X2);             // Read switch connected to Input 2
  orBit(X3);          // Logical OR with Input 3

  stack1.andBlock(); // Merge series branches using Block AND
  out(Y0);            // Send result to Output 0

}
```

Listing 46. *Logical AND of two series switch groups using Block AND instruction (Source: File > Examples > plcLib > Stack > AndBlock)*

The next section discusses methods of creating custom input/output allocations, should the standard configuration be unsuitable.

23 Defining Custom IO Allocations

In some cases, the default input output configuration may not be suitable. A *custom I/O allocation* may be appropriate if additional inputs or outputs are needed, custom hardware is being used, or if a different pin naming convention is preferred.

23.1 Preconfigured I/O Allocations

Version 1.2 of the PLC library provides an extended range of custom I/O files to suit a variety of hardware. The following configuration files are available from the **File > Examples > CustomIO** menu section.

- Arduino Uno and Mega with default pin configurations (**Uno**, **Mega**)
- Controllino PLCs (**ControllinoMaxiPLC**, **ControllinoMegaPLC**, **ControllinoMiniPLC**)
- Industrial Shields *Ardbox* PLCs (**ArdboxAnalogPLC**, **ArdboxPNPPLC**, **ArdboxRelayPLC**, **ArdboxTCHPLC**)
- Industrial Shields *M-Duino* PLCs (**MDuino19RelayPLC**, **MDuino21PLC**, **MDuino38RelayPLC**, **MDuino42PLC**, **MDuino57RelayPLC**, **MDuino58PLC**)
- Seedstudio Grove shields for Uno and Mega (**GroveUno**, **GroveMega**)
- TinkerKit shields for Uno and Mega (**TinkerkitUno**, **TinkerkitMega**)
- Velleman I/O Shield (**VellemanIOShield**)
- Use of custom pin names and user defined variable names (**CustomIO**)

If the supplied configurations are unsuitable, then you may need to create your own from scratch, as discussed next.

23.2 Case Study: Creating a Custom IO Allocation

This section demonstrates the creation of a custom IO mapping for the *Velleman Input/Output Shield for Arduino*.

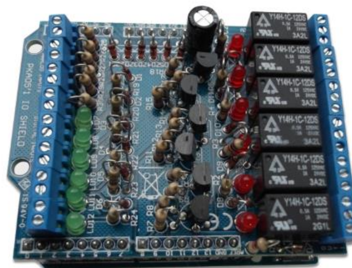


Figure 58. The Velleman Input/Output Shield for Arduino offers a useful range of inputs an outputs.

The module provides a total of eighteen IO connections, arranged as six analogue inputs, six digital inputs, and six relay-based digital outputs. Further details of connection and internal wiring details are available in the manual and datasheet.

The first task is to choose a suitable naming convention for inputs and outputs, and identify the associated pin connections.

- Analogue inputs can continue to use existing names *A0–A5*.
- Digital inputs will be allocated names *D0–D5*, linked to Arduino pins 2, 3, 4, 5, 6 and 7.
- Relay outputs will be called *R0–R5*, using pins 8, 9, 10, 11, 12 and 13.

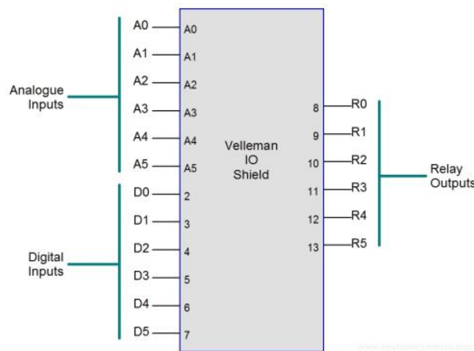


Figure 59. IO mapping for the Velleman Arduino IO shield.

The following sketch begins by disabling the definition of default pin names (X0, X1, ..., Y0, Y1, ...) using the command **#define noPinDefs**, and then specifies the required pin names as a series of integer constants.

```
#define noPinDefs       // Disable default pin definitions (X0, X1, ..., Y0, Y1, ...)
#include <plcLib.h>     // Load the PLC library

/* Programmable Logic Controller Library for the Arduino and Compatibles

   Velleman IO Shield Input / Output mapping
   Product information: http://www.velleman.co.uk/

   Connections:
   6 digital inputs D0-D5 (Arduino pins 2, 3, 4, 5, 6 and 7)
   6 analogue inputs A0-A5 (connected to the same Arduino pins)
   6 relay outputs R0-R5 (Arduino pins 8, 9, 10,11, 12 and 13)


   Note: outPWM() and outServo() commands should not be used with this
         board due to the use of mechanical switch-based relay outputs.

   Software and Documentation:
   https://github.com/wditch/plcLib

*/

  // Define digital input pins
  const int D0 = 2;
  const int D1 = 3;
  const int D2 = 4;
  const int D3 = 5;
  const int D4 = 6;
  const int D5 = 7;
```

```
  // Analogue pins will use existing names A0 - A5

  // Define Relay outputs as R0 - R5
  const int R0 = 8;
  const int R1 = 9;
  const int R2 = 10;
  const int R3 = 11;
  const int R4 = 12;
  const int R5 = 13;

void setup() {
  customIO();          // Setup inputs and outputs for Velleman IO Shield
}                      // (See IO tab for details)

void loop() {         // Sample code follows - replace as required
  in(D0);              // Read Digital Input 0
  out(R0);             // Send to Relay Output 0 (Relay 1 of 6)
}
```

Listing 47. *Creating a Velleman IO Shield Input / Output mapping (Source: File > Examples > plcLib > CustomIO > VellemanIOShield)*

The **customIO()** function is used (in place of the standard *setupPLC()* function) to define data directions for all inputs and outputs as shown in the following program extract (available by clicking the ***IO tab*** of the sketch).

```
void customIO (){

  // Input pin directions
  pinMode(D0, INPUT);
  pinMode(D1, INPUT);
  pinMode(D2, INPUT);
  pinMode(D3, INPUT);
  pinMode(D4, INPUT);
  pinMode(D5, INPUT);

  pinMode(A0, INPUT);
  pinMode(A1, INPUT);
  pinMode(A2, INPUT);
  pinMode(A3, INPUT);
  pinMode(A4, INPUT);
  pinMode(A5, INPUT);

  // Relay Output pin directions
  pinMode(R0, OUTPUT);
  pinMode(R1, OUTPUT);
  pinMode(R2, OUTPUT);
  pinMode(R3, OUTPUT);
  pinMode(R4, OUTPUT);
  pinMode(R5, OUTPUT);
}
```

The following points should be noted from the above example.

- Names for the digital inputs and relay outputs must be defined *globally*, by defining them outside of the *setup() { ... }* and *loop() { ... }* sections, hence making them available for use anywhere within the sketch.
- Pin names and allocations will not change so are defined as *constants* rather than *variables*.
- The plcLib software requires pin names to be defined of *signed integer* type (i.e. 'int' rather than 'unsigned int'), for reasons discussed earlier, in the Using Variables in Programs section

24 Strengths and Limitations of the Software

The plcLib software is freely available, and allows simple PLC-style programs to be developed on low cost Arduino compatible hardware. As such it offers an affordable entry point for those wishing to develop control-oriented software applications, or to use the Arduino for related educational purposes.

Commands are written as extensions to the C/C++ programming language, and hence make use of a C/C++ compatible command syntax. Programs may be designed using ladder diagram, function block diagram, sequential function chart, or structured text, but must be entered into the Arduino IDE in a text-only format, commonly known as instruction list. However, this process becomes easier with practice, and is aided by the availability of examples.

While the command syntax is not identical to any particular manufacturer, it should be straightforward to transfer existing knowledge between systems, or to compare operation of a given feature against the appropriate standard (IEC 61131).

The following points may be useful when deciding whether or not to use a plcLib-based solution in a particular situation.

- The ladder logic approach is particularly effective when implementing systems which can easily be represented using a ladder diagram, block diagram, or sequence-based system. These systems often involve performing a significant number of tasks in parallel, which is a particular strength. For other systems most easily represented using a flowchart, or similar, then a traditional programming approach may be more effective. In some cases it may be possible to use a combination of these approaches, perhaps making use of *structured text*.

- The *scan cycle* will inevitably slow down as more parallel tasks are added. Care should always be taken to ensure that the system response time is adequate for the system being controlled.

- The system may in some circumstance require several passes of the scan cycle to complete complex calculations, and intermediate results or 'glitches' may briefly occur at this time. In addition, outputs are updated at *each step* in the scan cycle (not just at the end). You should test programs carefully to ensure that any intermediate results will not affect the correct operation of the system.

- In general, you should avoid using the **delay()** command in ladder logic programs, as this halts the scan cycle for the duration of the time delay (software debugging is an exception, where the aim may be to deliberately slow down the scan cycle). If time-based operation is required then consider using a timer command such as **timerOn()**, **timerOff()**, **timerPulse()** or **timerCycle()**, as these use an interrupt driven approach which does not affect the scan cycle.

25 Command Reference

This page lists all commands supported by the plcLib software.

25.1 General Configuration

Command	**Description**	**Example**	**Section**
setupPLC();	Configures basic PLC settings.	setupPLC();	Configuring the Hardware

Notes

1. **setupPLC()** is an optional command which configures data directions for the default set of inputs and outputs, together with initial output values, as discussed in the *Configuring the Hardware* section. This command is typically omitted if a custom input/output allocation is being used, as discussed in the *Defining Custom IO Allocations* section.
2. The **#define noPinDefs** option, which is available with Version 1.2 or later, prevents creation and configuration of standard inputs and outputs (X0, X1, X2, ..., Y0, Y1, Y2, ...). If used, this setting must appear before the inclusion of the PLC library.

25.2 Single Bit Digital Input / Output

Command	**Description**	**Example**	**Section**
in(input);	Reads a digital input.	in(X0);	Single Bit Input / Output
out(output);	Outputs to a digital output.	out(Y0);	Single Bit Input / Output
inNot(input);	Reads an inverted digital input.	inNot(X0);	Single Bit Input / Output
outNot(output);	Outputs an inverted signal to a digital output.	outNot(Y0);	Single Bit Input / Output

25.3 Combinational Logic

Command	Description	Example	Section

andBit(input);	Logical AND with a digital input.	in(X0);
		andBit(X1);
		out(Y0);
			Performing Boolean Operations

orBit(input);	Logical OR with a digital input.	in(X0);
		orBit(X1);
		out(Y0);
			Performing Boolean Operations

xorBit(input);	Logical XOR with a digital input.	in(X0);
		xorBit(X1);
		out(Y0);
			Performing Boolean Operations

andNotBit(input);	Logical AND with an inverted digital input.	in(X0);
		andNotBit(X1);
		out(Y0);
			Performing Boolean Operations

orNotBit(input);	Logical OR with an inverted digital input.	in(X0);
		orNotBit(X1);
		out(Y0);
			Performing Boolean Operations

Notes:

1. To create a NAND or NOR function, start with the appropriate AND / OR example above, replacing the **out()** command with **outNot()**.
2. AND- and OR-based examples may be extended to have more than two inputs, if required.
3. Methods of solving complex (multiple branch) logic circuits are discussed in the *Using Variables in Programs* and *Stack-based Storage and Logic* sections.

25.4 Analogue Signal Input / Output

Command	Description	Example	Section
inAnalog(input);	Reads an analogue input.	inAnalog(X0);	Working with Analogue Signals
outPWM(output);	Outputs a PWM waveform.	outPWM(Y0);	Working with Analogue Signals

outServo(output);	Outputs to a servo.	outServo(Y0);	Working with Analogue Signals

Notes

1. Analogue input values are in the range 0–1023 (based on a standard 10-bit A–D converter). Output values are automatically scaled to match the output type (0–255 for PWM or 0–180° for a servo.)
2. The outServo() command requires the *Arduino Servo library*. The associated function must be locally defined, as explained in the Working with Analogue Signals section.

25.5 Comparing Analogue Signals

Command	Description	Example	Section
compareGT(input);	Compare an analogue input with a second analogue value, returning 1 if the former is larger.	inAnalog(X0); compareGT(X1); out(Y0);	Comparing Analogue Values
compareLT(input);	Compare an analogue input with a second analogue value, returning 1 if the former is smaller.	inAnalog(X0); compareLT(X1); out(Y0);	Comparing Analogue Values

Notes

1. To compare an analogue input with a fixed threshold, replace the input parameter of the compare command with a user specified variable of type 'unsigned int', and assigned a value in the range 0–1023.

25.6 Latches

Command	Description	Example	Section
latch(latch_output, reset_input);	Latches the previous digital input value.	in(X0); latch(Y0, X1);	Latching Outputs
latchKey(set_key, reset_key, output);	Latches a digital output based on keypad entry.	latchKey('1', '2', Y0);	Inputting from a Keypad
set(latch_output);	Latches (sets) a digital output if the previous value is true.	in(X0);	Latching Outputs

		set(Y0);	
reset(latch_output);	Clears (unsets) a latched output if the previous value is true.	in(X0); reset(Y0);	Latching Outputs

Notes

1. The latchKey() command requires the *Keypad library*. The associated function must be locally defined, as explained in the Inputting from a Keypad section.

25.7 Timers

Command	Description	Example	Section

timerOn(timer_variable, delay_ms);	Produces a delayed output after an input is enabled.	in(X0);
		timerOn(TIMER0, 2000);
		out(Y0);
			Using Time Delays

timerOff(timer_variable, delay_ms);	Delays turning an output off after an input is removed.	in(X0);
		timerOff(TIMER0, 2000);
		out(Y0);
			Using Time Delays

timerPulse(timer_variable, pulse_ms);	Produces a fixed width pulse triggered by a brief[3] input.	in(X0);
		timerPulse(TIMER0, 2000);
		out(Y0);
			Using Time Delays

timerCycle(low_variable, low_ms,
high_variable, high_ms);
	Creates a repeating pulse waveform, if enabled.	in(X0);
		timerCycle(AUX0, 900, AUX1, 100);
		out(Y0);
			Producing Repeating Waveforms

Notes

1. Elapsed timer variables should be defined of type *unsigned long* (e.g. **unsigned long TIMER0 = 0;**).

2. The timerPulse() command is available with Version 1.0 or later of the plcLib software.
3. The timerPulse() command is modified in Version 1.1 to be edge triggered, rather than level triggered. This prevents the timer from being automatically re-triggered if the trigger input is still high when the fixed width pulse finishes.

25.8 Edge Triggered Pulses

Command	Description	Example	Section
Pulse pulse_name;	Creates a pulse object.	Pulse pulse1;	Edge Triggered Pulses
pulse_name.inClock();	Connects an input signal to the pulse object.	in(X0); pulse1.inClock();	Edge Triggered Pulses
pulse_name.rising();	Read rising edge of pulse waveform[1].	pulse1.rising(); out(Y0);	Edge Triggered Pulses
pulse_name.falling();	Read falling edge of pulse waveform[1].	pulse1.falling(); out(Y1);	Edge Triggered Pulses

Notes

1. Pulses generated by rising or falling edges are active for a single scan cycle only.

25.9 Counters

Command	Description	Example	Section
counter_name(preset_value [, direction]);	Creates and configures a counter object[1].	Counter ctr1(5); Counter ctr2(10,1);	Counting and Counters
counter_name.countUp();	Counts up, if *count* is less than *preset value*[2, 3].	in(X0); ctr1.countUp();	Counting and Counters
counter_name.countDown();	Counts down, if *count* is greater than zero[2, 3].	in(X0); ctr1.countDown();	Counting and Counters
counter_name.preset();	Sets the internal count to the *preset value*, activating the *upperQ* output[3].	in(X1); ctr1.preset();	Counting and Counters
counter_name.clear();	Sets the internal count to zero, activating the *lowerQ* output[3].	in(X2); ctr1.clear();	Counting and Counters
counter_name.upperQ();	Equal to 1 if the internal count is equal to the *preset value*, 0 otherwise.	ctr1.upperQ(); out(Y0);	Counting and Counters
counter_name.lowerQ();	Equal to 1 if the internal count is equal to zero, 0 otherwise.	ctr1.lowerQ(); out(Y0);	Counting and Counters

counter_name.count();	Returns the internal count value.	Serial.println(ctr1.count());	Counting and Counters
counter_name.presetValue();	Returns the *preset value*.	Serial.println(ctr1.presetValue());	Counting and Counters

Notes

1. A newly created counter object is configured with a *preset value* (upper count limit) specified by the first parameter. If a single parameter is supplied, or the optional second parameter is 0, then the initial count value is set to 0, which is suitable for an up counter. A non-zero second parameter value causes the internal count value to be set to the preset value, which is effectively the start position for a down counter.
2. A *debounced* switch input, provided by an on-delay timer, may be used to prevent multiple triggering when connecting a switch to a counter input.
3. The *countUp*, *countDown*, *preset*, and *clear* counter methods are conditionally enabled by the result of the previous instruction, which will often be produced by reading a switch.
4. Counter commands are available with Version 0.8 or later of the plcLib software.

25.10 Shift Registers

Command	**Description**	**Example**	**Section**
Shift register_name([start_value]);	Creates and configures a shift register object[1].	Shift shift1(); Shift shift2(0x8888);	Shifting and Rotating Binary Data
register_name.inputBit();	Sets the serial input bit based on the previous input.	in(X0); shift1.inputBit();	Shifting and Rotating Binary Data
register_name.shiftLeft();	Shifts data one place to the left, on	in(X1); shift1.shiftLeft();	Shifting and Rotating

	the rising edge of the previous input.		Binary Data
register_name.shiftRight();	Shifts data one place to the right, on the rising edge of the previous input.	in(X2); shift1.shiftRight();	Shifting and Rotating Binary Data
register_name.reset();	Resets the internal shift register value to zero if the previous input is equal to 1.	in(X3); shift1.reset();	Shifting and Rotating Binary Data
register_name.bitValue(bit_position);	Returns the value of the shift register bit at the specified position.	shift1.bitValue(0); out(Y0);	Shifting and Rotating Binary Data
register_name.value();	Returns the value of the shift register as an unsigned integer[2].	Serial.println(shift1.value()); delay(200);	Shifting and Rotating Binary Data

Notes

1. Shift registers have a fixed data width of 16 bits. A newly created shift register object has a default value of 0x0000 if no parameter is specified, or a start value specified by the first parameter in the range 0X0000–0XFFFF.
2. The ability to read the shift register value is useful when debugging shift register applications.
3. Shift register commands are available with Version 0.9 or later of the plcLib software.

25.11 Stack and Block Logic

Command	Description	Example	Section
Stack stack_name;	Creates a single-bit, 32-level stack.	Stack stack1;	Stack-based Storage and Logic
stack_name.push();	Pushes the scanValue, expressed as a single-bit number, onto the stack.	stack1.push();	Stack-based Storage and Logic
stack_name.pop();	Updates the scanValue with a single-bit value removed from the stack.	stack1.pop();	Stack-based Storage and Logic
stack_name.andBlock();	ANDs the current scanValue with a single-bit value removed from the stack.	stack1.andBlock();	Stack-based Storage and Logic
stack_name.orBlock();	ORs the current scanValue with a single-bit value removed from the stack.	stack1.orBlock();	Stack-based Storage and Logic

Notes

1. Stack-based commands are available with Version 1.0 or later of the plcLib software.

26 Frequently Asked Questions

Q1. I've installed the software. What next?.

A1. If you have successfully installed the plcLib library then a set of example programs should be available by selecting ***File > Examples > plcLib > ...*** from the pull-down menu of the Arduino IDE. Examples are arranged into folders which are related to the different sections of the User Guide. (If these are not found, then see the Installing the Software section of the User Guide.)

Q2. Is there a graphical front-end?.

A2. A graphical front end is not currently supplied with the library, but a ladder logic GUI (Graphical User Interface) has been developed by Costantino Pipero. Source code is available through GitHub. Sample screenshots are shown below.

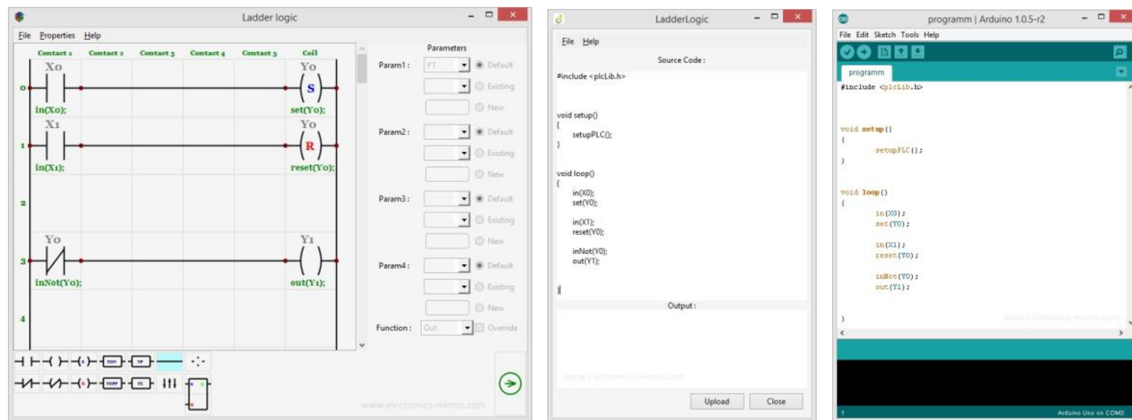


Figure 60. GUI-based modelling of a latch circuit with normal and inverted outputs.

Q3. Is the software free?.

A3. Yes, the software is released under the terms of the *GNU General Public Licence*, as described in the comments section at the top of the *plcLib.h* and *plcLib.cpp* source files.

Q4. What are the future development plans?.

A4. This is likely to be the final version of the current plcLib 1.x software branch, although Version 1.4 will remain available for the foreseeable future. It is hoped that an improved library and associated software will be developed in the future, with some or all of the following features.

- A fully object-oriented command syntax.
- Versions for other microcontroller-based products.
- Closer compatibility with appropriate standards, including IEC 61131-3.
- Graphical front end with cross-platform automated code generation.

Q5. My program doesn't work. Can you help?.
A5. Unfortunately, direct support is not available, although a User Guide and range of example sketches has been provided. The first step in fault finding is to make sure you understand how the software works, so read the manual carefully and study the example files.

A useful approach is to logically separate or 'decouple' the testing of hardware and software. One way to do this is to test your hardware connections using the simplest possible test application (something like the plcLib version of the *BareMinimum* sketch works well). Once you know the hardware is working, then proceed to testing and debugging the software.

As a rule, it is better to start with a simple working system, and progressively add to it, testing as you go, rather than 'dive straight in' with a complex combination of software and hardware.

27 Revision History

Brief details of all published versions is given below.

- Version 1.4: Published 1st October 2017. Added stand-alone user documentation to the github repository. Confirmed software compatibility with the Arduino on-line editor.
- Version 1.3: Published 1st April 2016. Added a remote serial monitor feature (an experimental feature intended to allow a remote software application to query Arduino internal values).
- Version 1.2: Published 21st Dec, 2015. Added improved ability to define custom pin names and optionally disable the default pin configuration through the "#define noPinDefs" setting. Provided a range of custom input/output configurations in the File > Examples > CustomIO menu area. Supported hardware now includes Industrial Shields Ardbox and M-Duino PLCs (10 variants), Controllino PLCs (3 variants), Seeedstudio Grove shields (Uno and Mega), Arduino (Uno and Mega), Tinkerkit shields (Uno and Mega) and the Velleman IO shield. Modified operation of TimerOn, TimerPulse, TimerOff, and TimerCycle commands to operate correctly during millis() command rollover, which occurs every 49.7 days after power-on or reset. (Test sketches for each command are given in the File > Examples >TimeDelays > TimerRolloverTest menu section.) Updated user documentation.
- Version 1.1: Published 16th May, 2015. Added single shot (edge triggered) pulse inputs triggered by low to high and high to low transitions of an input waveform. Modified the timerPulse command to wait for the input to go low before being re-triggered. (This now agrees with the standard behaviour in IEC standard 61131-3.). Updated the library format to be compatible with Arduino IDE 1.5+ (while remaining compatible with earlier versions).
- Version 1.0: Published 26th December, 2014. Added stack-oriented commands (push, pop, andBlock, orBlock). Added a pulse generator command (timerPulse). Added support for *unsigned long* user variables in addition to *unsigned int*.
- Version 0.9: Published 30th November, 2014. Added shift registers.
- Version 0.8: Published 20th September, 2014. Added counters.
- Version 0.7: Published 31st August, 2014. Added set() and reset() latch commands. Added analogue comparison commands. Added sequential function charts. Renamed timerPulse() command as timerCycle() to better reflect its operation as a cycle timer. Updated user documentation and added several new sections.
- Version 0.6: Published 7th July, 2013. Added keypad support.
- Version 0.51: Published 23rd June, 2013. Minor corrections (mostly typos).
- Version 0.5: Published 15th June, 2013. Initial release.

Notes:

1. Incremental versions, *0.1*, *0.2*, *0.3*, etc. add new functionality, while minor bug fixes and enhancements are indicated using an extra digit,such as *0.11*, *0.12*, *0.13.*

28 Appendix A – Serial Monitor (Experimental Feature)

28.1 Introduction

Version 1.3 of the software introduces a serial monitor feature, which allows a remote software application (or remote user) to query the internal state of Arduino inputs or outputs via a serial link.

This should be regarded as experimental feature at present. Basic testing has taken place on a limited number of Arduino platforms (mainly Uno and Mega), but software compatibility issues may still exist with other Arduino variants. The performance and range of features of the serial monitor are also yet to be fully evaluated.

28.2 Using the Serial Monitor

A concise 'query–response' control mechanism is used, which may be tested by typing commands into the Serial Monitor window. The serial monitor feature must first have been enabled, as shown in the following sketch, which is based on the 'bare minimum' sketch originally seen in Listing 1.

```
#include <plcLib.h>

/* Programmable Logic Controller Library for the Arduino and Compatibles

   Bare Minimum + Debug - Single bit digital input/output
                          with serial monitor enabled

   Connections:
   Input - switch connected to input X0 (Arduino pin A0)
   Output - LED connected to output Y0 (Arduino pin 3)

   Software and Documentation:
   https://github.com/wditch/plcLib

*/

void setup() {
  setupPLC();               // Setup inputs and outputs
  Serial.begin(9600);       // Enable serial port (needed for serial IO monitor)
}

void loop() {
  in(X0);                   // Read Input 0
  out(Y0);                  // Send to Output 0

  serialMonitor("Basic");  // Enable remote I/O monitoring via the serial port
}
```

Listing A1. *Bare Minimum with Remote Debugging enabled (Source: File > Examples > plcLib > InputOutput > BareMinimumDebug)*

As can be seen from the above listing, the first required change is to enable the serial interface at an appropriate Baud rate from within the 'setup' section. Next, the Serial Monitor must then be called from within the main program loop. The commands listed in the following table will then become available via the serial interface, once the sketch has been compiled and downloaded, and the Serial Monitor opened.

Serial (typed) Command	**Description**	**Example**
A	Reads the application name, returning “plcLib”.	**A** plcLib
a	Reads the ‘circuit-description’ string, set by the serialMonitor(“circuit-description”) command. If an empty string is supplied, the string is set to “Default”. ***Note:*** This setting is intended to allow a suitably configured remote software application to load an appropriate configuration file at startup.	**a** Default
M	Reads major version of the plcLib software. E.g. ‘1’ is returned if the current plcLib version is “1.4”.	**M** 1
m	Reads minor version of the plcLib software. E.g. ‘4’ is returned if the current plcLib version is “1.4”.	**m** 4
P	Reads the maximum number of pins (*maxPin* setting), returning ‘20’ or ‘70’ depending on the Arduino board detected.	**P** 20
S[pin_no]	Reads the status value of the specified pin. E.g. **S3** returns the status byte for pin 3 which is made up of the binary weighted values of all enabled flags, as defined below. • pinUsedFlag, bit 7 (value = 128) • reportingEnabledFlag, bit 6 (value = 64) • digitalInputFlag, bit 5 (value = 32) • analogInputFlag, bit 4 (value = 16) • digitalOutputFlag, bit 3 (value = 8) • analogOutputFlag, bit 2 (value = 4) • servoOutputFlag, bit 1 (value = 2)	**S3** 136 (=128+8)

	• pinUpdatedFlag, bit 0 (value = 1)	
V[pin_no]	Reads the current I/O value associated with the pin. E.g. **V3** returns the value of pin 3 (Y0). ***Notes:*** Reading the value of a pin clears the *pinUpdated* flag. The value of a pin may still be read, even if the *reportingEnabled* flag has been cleared.	**V3** 0
E[pin_no]	Enable status updating on a pin. E.g. the command **E3** sets the *reportingEnabled* status bit for pin 3	**E3** **S3** 200 (=128+64+8)
e[pin_no]	Disable status updating on a pin. E.g. the command **e3** clears the *reportingEnabled* status bit for pin 3 and also clears the *pinUpdated* flag.	**e3** **S3** 136 (=128+8)
U	Displays a comma separated list of pins which have been updated, or returns 'N' if no pins have been updated. ***Note:*** For a pin to be listed as updated, its *reportingEnabled* bit must first be set, and a change to the value associated with the pin must have occurred.	**E14** **E3** **U** 3,14 **V3** 0 **U** 14 **V14** 0 **U** N

28.3 Technical Operation

Notice from the previous table that pin-related commands make use of physical pin numbers, rather than logical pin names. In Listing A1 for example, a digital input is read from pin logical pin X0 (labelled as A0 on the Arduino) which is actually connected to physical pin 14. Similarly, logical output Y0 is equivalent to physical pin 3.

While the need to identify physical pin numbers may be inconvenient for a human user, recall that the serial monitor is intended to operate with a (future) remote application, which in turn would be expected to have a mapping of logical to physical pin values. This mapping could be loaded, based on the parameter value supplied to the *serialMonitor* command - e.g. *serialMonitor("Basic");*.

Internal operation of the serial monitor feature may be understood by studying the plcLib.h and plcLib.cpp files in detail. Key points are listed below.

1. The *monitorEnable* setting enables or disables the serial monitor feature (a value of 1 enables monitoring). Disabling this feature prevents generation of monitoring code, associated with several input/output commands, hence reducing the object code size. The *monitorEnable* setting is defined in the plcLib.h file.
2. The *maxPins* setting is found in the plcLib.cpp file, and is set to 70 if Arduino Mega or Due is detected and set to 20 otherwise.
3. A *pinValue* structure is created in plcLib.cpp to hold pin status bits and previous scan values for each pin used in the main sketch. The *pinStatusUpdate* function in the plcLib.cpp file updates the pinValue structure when called from commands *in*, *inNot*, *inAnalog*, *out*, *outNot*, *outPWM*, *latch*, *set*, and *reset*. The values used are: -

   - pinUsedFlag, bit 7 (value = 128)
   - reportingEnabledFlag, bit 6 (value = 64)
   - digitalInputFlag, bit 5 (value = 32)
   - analogInputFlag, bit 4 (value = 16)
   - digitalOutputFlag, bit 3 (value = 8)
   - analogOutputFlag, bit 2 (value = 4)
   - servoOutputFlag, bit 1 (value = 2)
   - pinUpdatedFlag, bit 0 (value = 1)