plcLib User Guide

Simple PLC-style programming in JavaScript and C++.

Version 2.0 Beta 6 | Last updated 23/4/23 by WD (***Note:*** Editing is still in progress – page 33 onwards!)

Contents

Overview of *plcLib* and the *plcLib – live* Environment 3
- Getting Started with PlcLib - Live 4

Introducing to Ladder Logic and Major Program Features 6
- Single Bit Input and Output 7
- Performing Boolean Operations 8
- Latching Outputs 10
  - Using the Latch Command 10
  - Using the Set and Reset Commands 11
- Edge Triggered Pulses 12
  - Specifying the Default Pulse Polarity 13
  - Creating an Edge Triggered Bistable 14
- Generating Timing Diagrams 15
- Using Time Delays 16
  - Producing a Turn-on Delay 16
  - Switch Debouncing 17
  - Creating a Turn-off Delay 17
  - Creating a Fixed Duration Pulse 18
  - Producing Repeating Waveforms 19
    - Manual Pulse Creation 19
    - Using the timerCycle Command 20
- Counting and Counters 20
  - Up Counter 21
  - Down Counter 22
  - Up/Down Counter 22
  - Debugging Counter-based Applications 23
- Shifting and Rotating Binary Data 24
  - Creating and Using Shift Registers 24
  - Rotating Data 26
- Working with Analogue Signals 27
  - Controlling LED Brightness using PWM 27
  - Position Control Using Servos 28
  - Comparing Analogue Values 28
- Solving Complex Logic Circuits 30
  - Using Variables with Complex Logic Circuits 31

Stack-based Storage and Logic....31
Block Logic Operations....32

Overview of *plcLib* and the *plcLib – live* Environment

The *plcLib – live* environment (https://plclib.org/live/) consists of a JavaScript-based web IDE and simulator, plus an associated C++ library. It allows development and testing of simple PLC-style programs in a Web-based development environment. The resulting code may be downloaded to a range of low-cost microcontroller-based systems, provided they support the Arduino IDE.

It is important to state at this early stage, that the system is intended to be used for educational and academic research purposes only. The associated C++-based plcLib library is released under a permissive MIT licence, but does not come with any kind of warranty, and is used entirely at your own risk.

A traditional PLC is typically used to control the production process – i.e. to *make* products. In contrast, the plcLib library allows PLC-style functionality to be embedded within the actual end-product itself, so it *becomes part of* the product and its functionality. The result is not a PLC, but rather a *smart product* which has access to the typical range of features of a PLC, including inputs, outputs, latches, time delays, counters, shift registers, pre-defined functions, finite state machines, and of course, simplified parallel programming. The system also has features intended to simplify the integration of 3rd party software libraries. This in turn makes it easier to link with the additional hardware, peripherals and communication protocols found in modern microcontroller-based systems. A selection of potential applications of smart devices is shown in Figure 1.

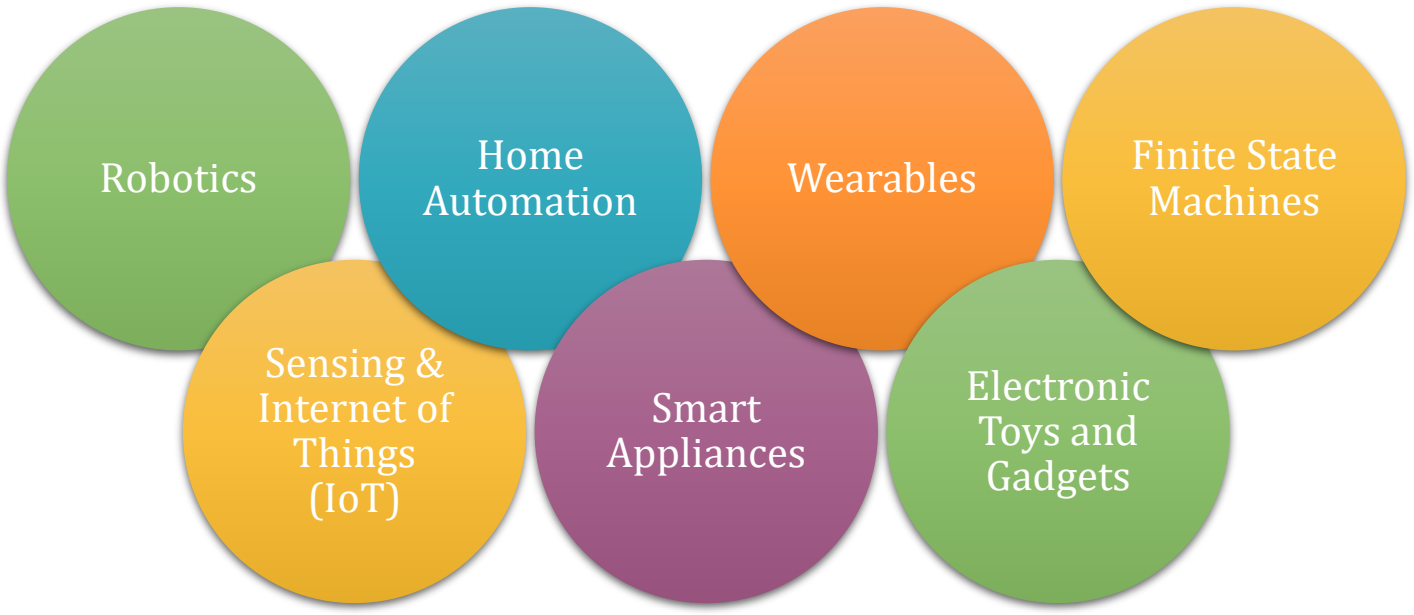

Figure 1. *Some potential application areas of smart devices.*

Getting Started with PlcLib - Live

First time users may wish to start by running the default 'sample' program from the *Simulator Code* text area. To do this, click the *Run* button in the *Simulation* area and then activate inputs to see the effect on outputs, as shown in Figure 1.

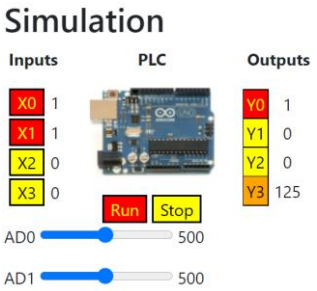

Figure 2. *Running and testing the program.*

Next, study the sample code to see what it does, referring to the documentation, as required. You should be able to identify a two input Boolean logic gate, a 1-second time delay (on-delay timer), a counter (which counts to 3), and finally a *pulse-width modulated* (PWM) output which is driven by a slider.

A wide range of example programs are available from the pull-down menu, grouped by category. Some examples are also ordered from basic to more advanced, as shown in Figure 3.

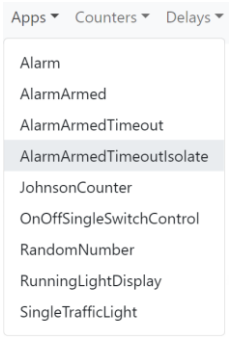

Figure 3. *Selecting an example program from the pull-down menu.*

Once correct operation is confirmed by simulation, the desired target system hardware may be selected from the *Select Target System* drop-down list, as shown in Figure 4.

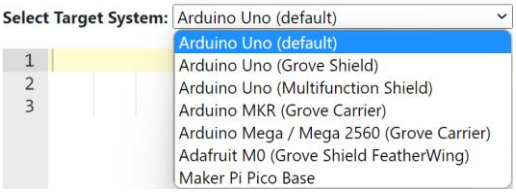

Figure 4. *Currently supported target systems.*

Click the *Generate code & copy to clipboard* button to create the C++ code and copy it to the clipboard. Next, switch to the Arduino IDE, delete the current sketch and replace it with the code stored in the clipboard (pressing *Ctrl+A* followed by *Ctrl+V* is a quick way to do this). Finally, compile and download the sketch to the target hardware - having previously installed the C++ version of the plcLib library to the Arduino IDE, of course.

It is also possible to load and save your own files, in addition to using the pre-created examples. However, there are some slight restrictions, which relate to the use of a browser-based system.

Opening a local file is a two-stage process. Firstly, select the file to be loaded by clicking the *Browse...* button and then using the dialogue box to select the file. Secondly, Click the *Load selected local file* option from the *Actions* pull-down menu. This is illustrated by Figure 5.

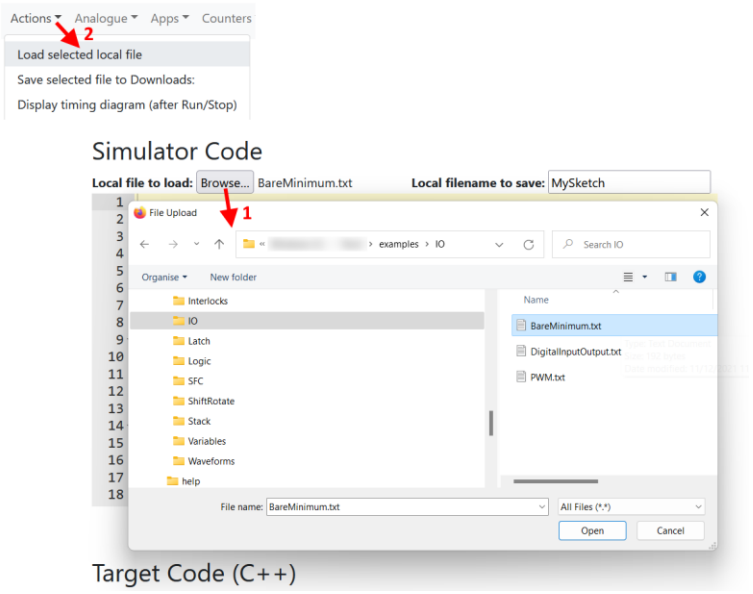

Figure 5. *Selecting and loading a local file.*

You can also save the contents of the Simulator code window as a local file, although you are restricted to saving to the *Downloads* folder, for browser security reasons. Once again, this is a two-stage process. Firstly, enter the desired filename in the *Local filename to save* text box. Secondly, select the *Save selected file to Downloads* option from the *Actions* pull-down menu. This is illustrated by Figure 6.

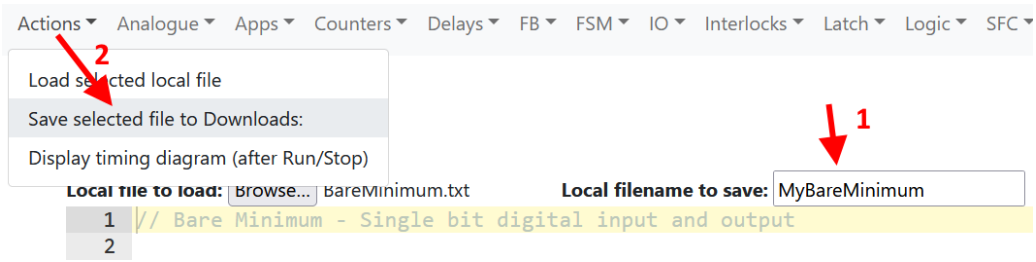

Figure 6. *Saving to the Downloads folder as a local file.*

Remaining sections of the User Guide will discuss the various features and capabilities of the plcLib library, starting with basic concepts and becoming progressively more advanced. Note that following code examples, or sketches, show the JavaScript syntax, unless otherwise stated. These examples may be loaded into the *Simulator Code* text area by accessing the given path from the plcLib live webpage (for example - Source: *Logic > NandNorXnor*). Equivalent Arduino C++ syntax becomes visible from the *Target Code* text area, after clicking the *Generate code & copy to clipboard* button. Syntax differences between the two languages are relatively minor, but include the inclusion of libraries in C++, plus variations function definition, object creation and debugging commands.

Introducing to Ladder Logic and Major Program Features

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. Figure 7 below illustrates the process.

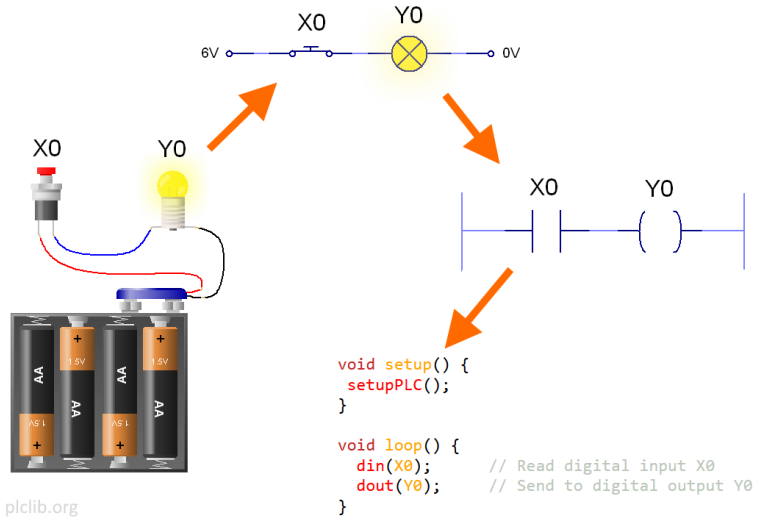

Figure 7. *Converting an electrical circuit into a ladder diagram and then a program (C++ syntax shown above).*

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 in Figure 8 below.

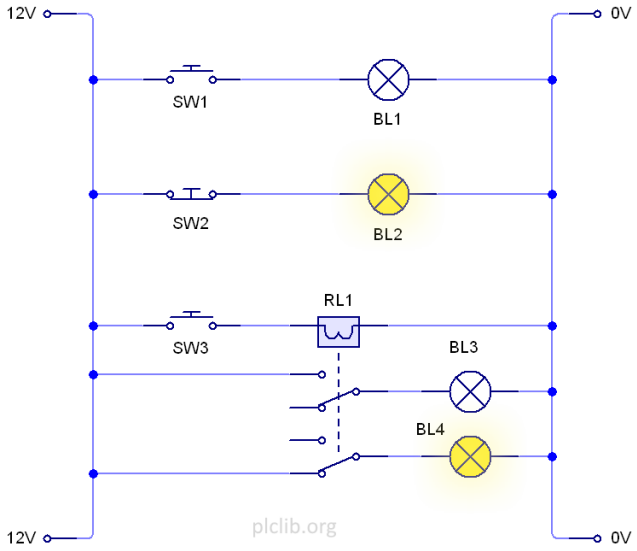

Figure 8. *A simple Relay Logic circuit.*

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 makes use of the plcLib library must first be entered as a text-based sketch. With practice, the process of converting a ladder diagram into an Arduino sketch becomes relatively straightforward.

The Web-based simulator is written in JavaScript, while the Arduino library itself uses C++, which is the underlying programming language of the Arduino family. There are subtle differences in syntax between the two, but conversion between them is handled automatically by the Web IDE.

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 of Figure 9 below shows the input and output of values in normal and inverted forms.

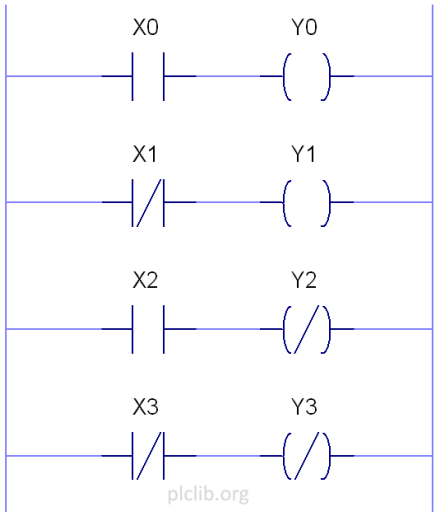

Figure 9. *A ladder diagram showing methods of reading inputs and controlling outputs.*

The ladder diagram may be easily converted to a text-based sketch, as shown in Listing 1 below.

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

function setup() {
}

function loop() {
  din(X0);      // Read input X0
  dout(Y0);     // Send to output Y0

  dinNot(X1);   // Read input X1 (inverted)
  dout(Y1);     // Send to output Y1

  din(X2);      // Read input X2
  doutNot(Y2);  // Send to output Y2 (inverted)

  dinNot(X3);   // Read input X3 (inverted) and send to output Y3 (inverted)
  doutNot(Y3);  // (The double negative cancels out)
}
```

Listing 1. *Digital Input / Output (source: IO > 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, which is 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.

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. This is illustrated by Figure 10 below.

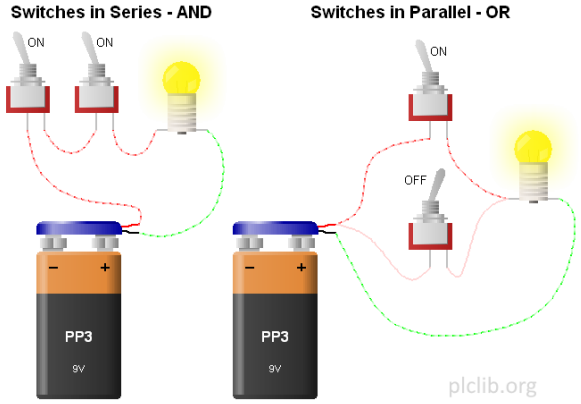

Figure 10. *Switches in series and parallel perform logical AND and OR functions.*

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 in Figure 11 below.

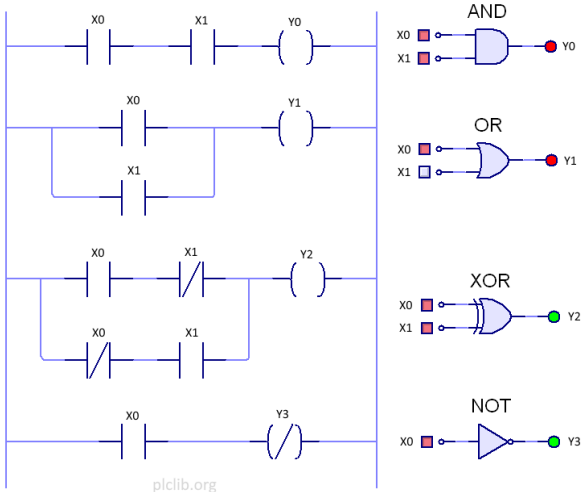

Figure 11. *Boolean logic ladder logic functions and their equivalent logic functions.*

This ladder diagram arrangement may be easily coded, as shown in Listing 2.

```
// AND, OR, XOR and Not - Boolean Logic

function setup() {
}

function loop(){
  din(X0);      // Read digital input X0
  andBit(X1);   // AND with X1
  dout(Y0);     // Send result to Y0

  din(X0);      // Read digital input X0
  orBit(X1);    // OR with X1
  dout(Y1);     // Send result to Y1

  din(X0);      // Read digital input X0
  xorBit(X1);   // XOR with X1
  dout(Y2);     // Send result to Y2
```

```
  din(X0);     // Read digital input X0
  doutNot(Y3); // Send inverted result to Y3
}
```

Listing 2. *AND, OR, XOR and Not functions (Source: Logic > AndOrXorNot).*

If active-low outputs are required then NAND, NOR and XNOR equivalent functions may be created in ladder logic, as shown in Figure 12 below.

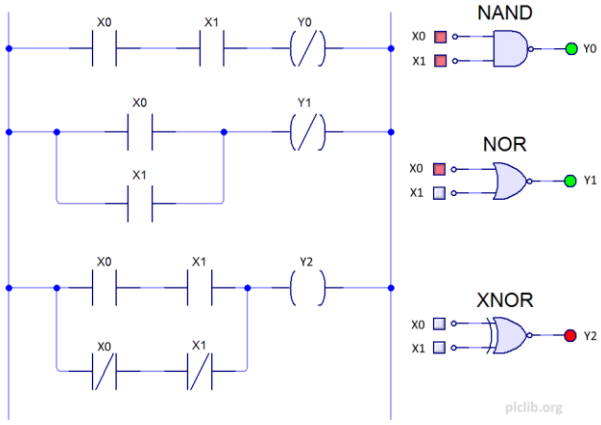

Figure 12. *NAND, NOR and XNOR ladder logic circuits and their equivalent logic functions.*

Coding of these functions is achieved by replacing the *dout* instructions of the previous example with the negative logic *doutNot* equivalent, as shown in Listing 3, below.

```
// NAND, NOR, XNOR

function setup() {
}

function loop() {
               // NAND
  din(X0);     // Read Input X0
  andBit(X1);  // AND with Input X1
  doutNot(Y0); // Send result to Output Y0 (inverted)

               // NOR
  din(X0);     // Read Input X0
  orBit(X1);   // OR with Input X1
  doutNot(Y1); // Send result to Output Y1 (inverted)

               // XNOR
  din(X0);     // Read Input X0
  xorBit(X1);  // XOR with Input X1
  doutNot(Y2); // Send result to Output Y2 (inverted)
}
```

Listing 3. *NAND, NOR, and XNOR functions (Source: Logic > NandNorXnor).*

Logical operations may also be performed with one or more of the inputs inverted, as seen in Figure 13.

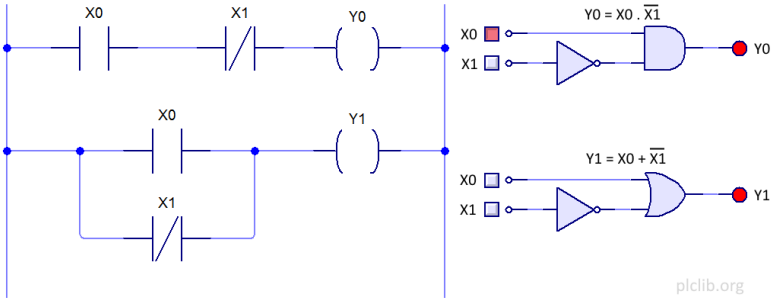

Figure 13. *Performing Boolean operations involving inverted input signals.*

An equivalent JavaScript sketch is shown in Listing 4 below.

```
// Inverted Input Logic

function setup() {
}

function loop() {
  din(X0);          // Read digital input X0
  andNotBit(X1);    // AND with Input X1 (inverted)
  dout(Y0);         // Send result to output Y0

  din(X0);          // Read digital input X0
  orNotBit(X1);     // OR with Input X1 (inverted)
  dout(Y1);         // Send result to output Y1
}
```

Listing 4. *Inverted input logic (Source: 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.

Latching Outputs

A momentary input may be *latched*, causing it to remain active (or *set*) until it needs to be cancelled (or *reset*).

The plcLib library allows latches to be created either using a dedicated *latch* command, or using separate `setL` (set latch) and `resetL` (Reset latch) commands, as discussed in the next two sections.

Using the Latch Command

The `Latch` command offers a self-latching arrangement, and requires a minimum of two lines of code. An equivalent block diagram representation is shown in Figure 14 below, the upper variant having normal and inverted outputs, and the lower with only a single normal output.

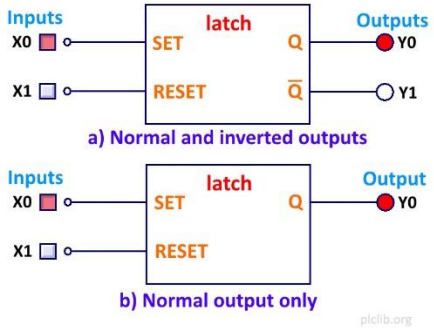

Figure 14. *Set reset latch symbols, shown both with and without an inverted output.*

The sketch of Listing 5 demonstrates a latch with both normal and inverted outputs, although the latter two lines used to generate the inverted output are optional.

```
// Latch Command

function setup() {
}

function loop()
{
  din(X0);              // Read switch connected to digital input X0 (Set input)
  latch(Y0, X1);        // Latch, Q = Output Y0, Reset = Input X1

  din(Y0);              // Read Q digital output Y0 and generate NotQ on Output Y1
  doutNot(Y1);          // (These two lines are optional)

                        // Note: If different +ve/-ve logic settings are used for
```

```
                        // inputs/outputs, then for C++ only, you'll need to: -
                        // * Replace 'din(Y0);' with 'dinNot(Y0);'
}
```

Listing 5. *Latch Command (Source: 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.

Care should also be exercised to check proper operation of the optional inverted output (in C++ only), where differing positive and negative logic settings have been used for inputs and outputs, given the command *`din(Y0)`* is actually reading the state of an output!

Using the Set and Reset Commands

An alternative method of creating a latched output is to use separate *setL* and *resetL* commands, as shown in the sketch of Listing 6.

```
// Using separate setL and resetL Commands

function setup() {
}

function loop()
{
  din(X0);              // Read switch connected to digital input X0 (Set input)
  setL(Y0);             // Set latched output Y0 to 1 if X0 = 1,
                        // leave Y0 unaltered otherwise

  din(X1);              // Read switch connected to digital input X1 (Reset input)
  resetL(Y0);           // Reset latched output Y0 to 0 if X1 = 1,
                        // leave Y0 unaltered otherwise
}
```

Listing 6*. Using separate setL and resetL commands (Source: Latch > SetResetCommands).*

A letter may be added to the standard ladder logic output symbol indicating whether the output is a 'set' or 'reset' type, as shown in the equivalent ladder diagram of Figure 15.

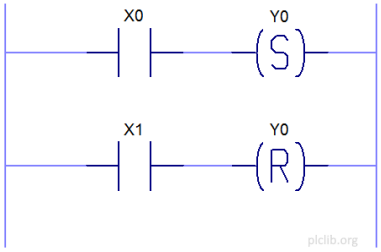

Figure 15. *Ladder diagram representation of a latch based on Set and Reset outputs.*

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

A potential issue with the sketch of Listing 6 relates to the direct updating of outputs by the library. If inputs *X0* and *X1* are simultaneously applied, then the output is first set and then reset by each pass of the scan cycle. The associated output will then seem to 'oscillate'. This behaviour may be overcome by writing to an intermediate variable (called an *auxiliary*), and then to the output. The last value written to the auxiliary is the one which effectively 'dominates'. Hence, it is possible to design latch circuits which are either set- or reset-dominant, simply by altering the order of operation. This approach is demonstrated in Listing 7.

```
// Using setL and resetL commands with output via an Auxiliary

// Outputting via an intermediary variable avoids oscillation when
```

```
// S = R = 1, which is associated with direct output. The last command
// (resetL in the example below) is the one which 'dominates'.

myLatch = new Auxiliary();

function setup() {
}

function loop()
{
  din(X0);              // Read switch connected to digital input X0 (Set input)
  setL(myLatch);        // Set latched output to 1 if X0 = 1,
                        // leave output unaltered otherwise

  din(X1);              // Read switch connected to digital input X1 (Reset input)
  resetL(myLatch);      // Reset latched output to 0 if X1 = 1,
                        // leave output unaltered otherwise

  din(myLatch);         // Read latched variable
  dout(Y0);             // Output to Y0

}
```

Listing 7. *Using setL and resetL commands with an auxiliary variable (Source: Latch >SetResetCommandsAuxiliary).*

Note that increasing (but still subtle) differences may also be observed between the JavaScript program syntax used by the web-based editor and simulator, and the C++ syntax of the Arduino. This is illustrated by Figure 16.

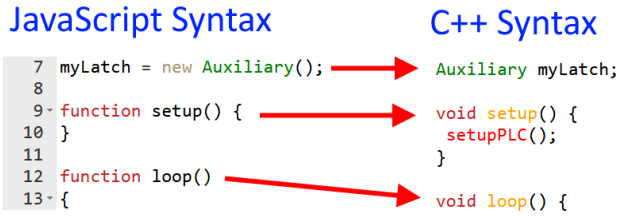

Figure 16. *Automated conversion between JavaScript and C++ syntax.*

No action is needed by the user, as the system automatically converts from one to the other during target code production. Please see the *plcLib Reference Manual* for specific details of any syntax differences.

Edge Triggered Pulses

The *pulse* command allows the generation of 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 in Figure 17.

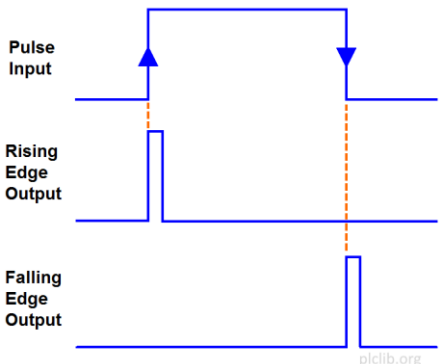

Figure 17. *Brief output pulses may be generated from the rising or falling edges of an input waveform.*

This may be represented as shown in the following ladder diagram of Figure 18, with rising or falling edges on input *X0* being used to momentarily set outputs *Y0* and *Y1*, respectively.

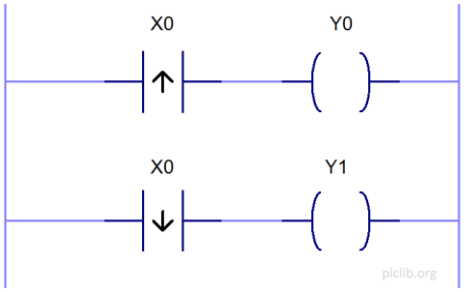

Figure 18. *Output pulses on Y0 and Y1 are generated by rising or falling edges of input X0.*

An equivalent sketch is shown in Listing 8.

```
// One Shot Pulse - Single scan cycle pulse

myPulse = new Pulse();  // Create pulse object

function setup() {
}

function loop() {
  din(X0);              // Read digital input X0
  myPulse.inClock();    // Connect to pulse object

  myPulse.rising();     // Detect rising edge
  dout(Y0);             // Send to digital output Y0

  myPulse.falling();    // Detect falling edge
  dout(Y1);             // Send to digital output Y1

  //$delay(200);         // Slow down pulse for viewing in C++
}
```

Listing 8. *Generating rising- and falling-edge pulses (Source: Waveforms > PulseOneShot).*

Examining the above listing, a pulse object called *'myPulse'* 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.

Note that output pulses last for a single scan cycle only, so a 200 millisecond time delay has been added to the above sketch via an escape sequence (//$). The line is treated as a comment by the JavaScript simulator, but the escape sequence is automatically removed before passing the remainder of the line to the C++ environment. The resulting short delay 'stretches' 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.

Specifying the Default Pulse Polarity

It is also possible to use the *Pulse* object itself as a parameter in other plcLib commands, making use of the *default* pulse polarity specified when the *Pulse* object was created. For example, the following JavaScript code extracts create pulses with the specified default pulse polarities: -

```
P0 = new Pulse();      // Creates a pulse P0 which defaults to detecting a 0 to 1 transition
P0 = new Pulse(0);     // Creates a pulse P0 which specifies a 0 to 1 transition
P0 = new Pulse(1);     // Creates a pulse P0 which specifies a 1 to 0 transition
```

Listing 9. *A code extract demonstrating alternative pulse creation syntax. (These are mutually exclusive, so only one can be selected.)*

The pulse object P0 may then be referred to as a parameter in other plcLib commands, which will then be 'triggered' during the single scan cycle when the specified edge is active. As an example of this approach, Listing 10 shows the creation of two *Pulse* objects, with *P0* having a positive edge transition and *P1* a negative equivalent.

Following lines link the positive going pulse on *X0* to digital output *Y0*, while the falling edge of Pulse *P1* connects the negative edge of *X1* to digital output *Y1*. Pulse objects *P0* and *P1* are logically combined using a Boolean OR function block (*orFB*), with the output sent to digital output *Y2*.

```
// One Shot Pulse - Single scan cycle pulse using default transitions

P0 = new Pulse(0);    // Create 1st pulse object, default = 0 to 1
P1 = new Pulse(1);    // Create 2nd pulse object, default = 1 to 0

OR0 = new FunctionBlock(); // Create OR FB variable

function setup() {
}

function loop() {
  din(X0);            // Read digital input X0
  P0.inClock();       // Connect to pulse object P0

  din(X1);            // Read digital input X1
  P1.inClock();       // Connect to pulse object P1

  din(P0);            // Detect rising edge of P0 (from default)
  dout(Y0);           // Send to digital output Y0

  din(P1);            // Detect falling edge of P1 (from default)
  dout(Y1);           // Send to digital output Y1

  orFB(OR0, P0, P1);  // Combine pulses

  din(OR0);           // Read OR0 result
  dout(Y2);           // Send combined result to Y2

  //$delay(200);         // Slow down pulse for viewing in C++
}
```

Listing 10. *Using default edge transition types with one shot pulses (Source: Waveforms > PulseOneShotDefault).*

The approach of Listing 10 is slightly simpler than that seen earlier in Listing 8, particularly where a single type of edge transition is required. However, the *rising* and *falling* methods of the *Pulse* object remain available, even where the simpler approach has been used.

Creating an Edge Triggered Bistable

A simple set reset latch may be created using several different methods, as demonstrated in the earlier *Latching Outputs* section. We previously saw that direct output via *setL* and *resetL* commands could cause oscillation if *Set* and *Reset* were simultaneously applied, and that output via a buffer variable could overcome this issue. Given the potential for faulty input switches, it is also useful to consider what would happen to a latch-based circuit if both the *Set* and *Reset* inputs were simultaneously activated, due to one of the external input switches becoming '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 of Figure 19.

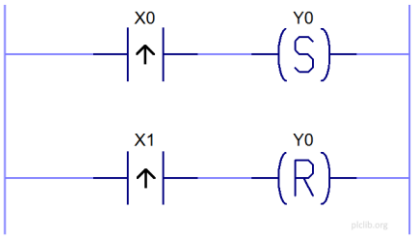

Figure 19. *Using edge triggered inputs to set or reset an output.*

An equivalent sketch is shown below, in Listing 11.

```
// Using setL and resetL commands with edge triggered inputs

setPulse = new Pulse();     // Create a pulse object to set the latch
resetPulse = new Pulse();   // Create a pulse object to reset the latch

function setup() {
}

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

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

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

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

Listing 11. *Set or reset an output using edge-triggered inputs (Source: Latch > SetResetEdgeTriggered).*

Generating Timing Diagrams

The web IDE is also capable of generating *timing diagrams*, similar to the one previously seen in Figure 17. This is achieved with the aid of the `LogVars` command. An example timing diagram is shown in Figure 20, which is adapted from the earlier example of Listing 8.

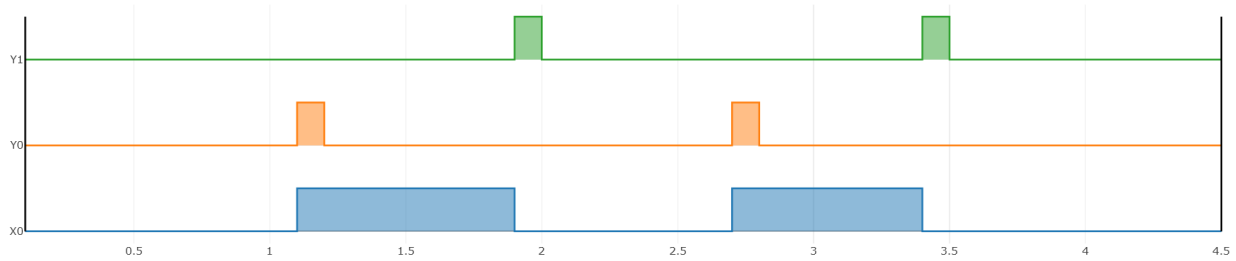

Figure 20. *A sample timing diagram showing rising- and falling-edge pulses.*

The first step is to add the variables to be monitored, via the `LogVars` command, as shown in Listing 12.

```
// One Shot Pulse - Single scan cycle pulse with timing diagram

myPulse = new Pulse();  // Create object

function setup() {
  logVars("X0", "Y0", "Y1");
}

function loop() {
  din(X0);              // Read digital input X0
  myPulse.inClock();    // Connect to pulse object

  myPulse.rising();     // Detect rising edge
  dout(Y0);             // Send to digital output Y0

  myPulse.falling();    // Detect falling edge
  dout(Y1);             // Send to digital output Y1

  logVars(X0, Y0, Y1);
  //$delay(200);         // Slow down pulse for viewing in C++
}
```

Listing 12. *Adding variable logging via the logVars command (Source: Waveforms > PulseOneShotLogVars).*

Notice that the `LogVars` command appears twice - firstly in the setup function, where it sets up the variable names (in quotes), then secondly in the main program loop, where the actual logging takes place.

To generate the Timing diagram, the recommended sequence is: -

1. Press *Stop* to clear any previous data.
2. Press *Run*
3. Quickly activate the correct input sequence, as required to generate the signal timing.
4. Press *Stop*
5. Select *Actions > Display timing diagram (after Run / Stop)*

Please see the Reference Manual for more details regarding the creation of timing diagrams.

Using Time Delays

The plcLib library can create time delays which are triggered by the activation or deactivation of an input signal (`timerOn` and `timerOff` commands respectively), and can also produce an output pulse of fixed duration (`timerPulse` command). Each timer operates independently of any others, which is an example of parallel processing, and is one of the most powerful features of PLC-style programming.

All commands are linked to a named *Timer* object, which is used to keeps track of the internal operation of the timer (whether it is active or inactive, the elapsed time since it was last triggered for example).

Note: Avoid using the Arduino's own `delay` command wherever possible, as this halts the scan cycle for the duration of the delay, and only supports a single delay being active at any time.

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, as shown in the timing diagram of Figure 21.

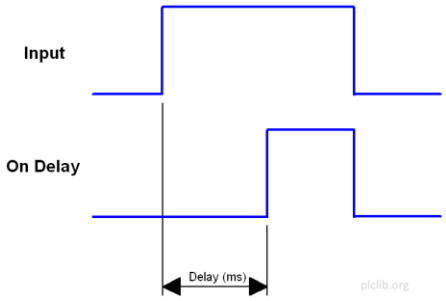

Figure 21. *An on-delay timer provides delayed activation of an output.*

A timer-based system may be be conveniently represented using a block diagram symbol as shown in the example of Figure 22.

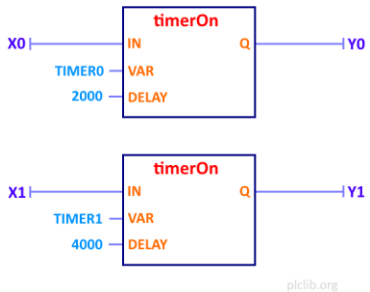

Figure 22. *A pair of on-delay timers produce separate delays, controlled by two different inputs.*

The two on-delay timers provide independent activation time delays of 2 seconds and 4 seconds on Outputs *Y0* and *Y1*, respectively. Output *Y0* becomes active after a signal connected to Input *X0* has been continuously active for 2

seconds, while Output *Y1* requires Input *X1* to have been continuously active for a period of 4 seconds. The equivalent JavaScript sketch is shown below.

```
// Turn-on Delay

TIMER0 = new Timer();
TIMER1 = new Timer();

function setup() {
}

function loop() {
  din(X0);                  // Read Input 0
  timerOn(TIMER0, 2000);    // 2 second delay
  dout(Y0);                 // Output to Output 0

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

Listing 13. *A pair of turn-on delays (Source: Delays > TimerOn).*

Notice that each timer makes internal use of a *Timer* object, which must be declared at the start of the program.

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 example of Listing 14.

```
// Switch Debounce

TIMER0 = new Timer();

function setup() {
}

function loop()
{
  din(X0);                 // Read Input 0
  timerOn(TIMER0, 10);     // 10 ms delay
  dout(Y0);                // Output to Output 0
}
```

Listing 14. *Switch debounce (Source: Delays > SwitchDebounce).*

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, as shown in the timing diagram of Figure 23.

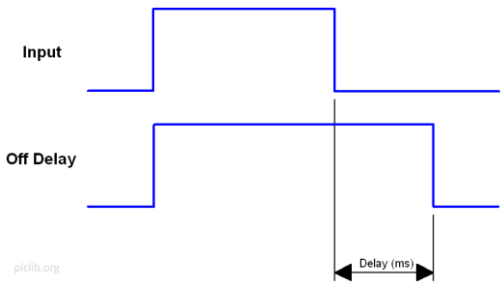

Figure 23. *An off-delay causes an output to remain active for a fixed duration after the input is removed.*

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 block diagram symbol of Figure 24 shows an off-delay timer with a 2 second turn-off delay.

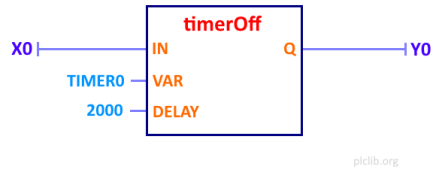

Figure 24. *An off-delay timer with a 2 second delay.*

The equivalent sketch is given in Listing 15, below.

```
// Turn-off Delay

TIMER0 = new Timer();

function setup() {
}

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

Listing 15. *A turn-off delay (Source: Delays > TimerOff).*

Creating a Fixed Duration Pulse

The `timerPulse` command creates a fixed width output pulse when triggered by a brief input pulse. This is equivalent to an electronic monostable circuit. A sample timing diagram is given in Figure 25.

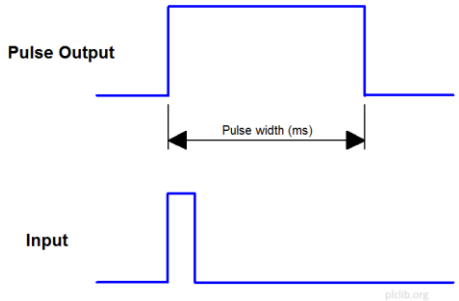

Figure 25. *A fixed-width pulse is created by the* `timerPulse` *command.*

An example sketch is shown in Listing 16 below.

```
// Fixed Pulse

TIMER0 = new Timer();

function setup() {
}

function loop() {
  din(X0);                    // Read Input 0
  timerPulse(TIMER0, 2000);   // 2 second pulse
  dout(Y0);                   // Output to Output 0

}
```

Listing 16. *Creating a fixed width pulse (Source: Delays > FixedPulse).*

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

Producing Repeating Waveforms

A repeating pulse may be defined using the duration of the low and high components of the waveform, as shown in Figure 26.

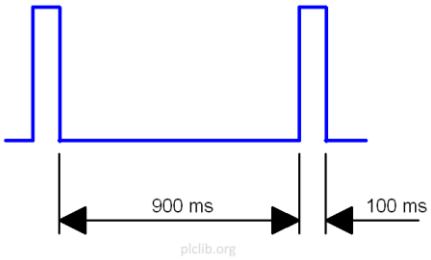

Figure 26. *A repeating pulse may be defined in terms of the durations of the low and high sections.*

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.

Manual Pulse Creation

A pair of cross-connected on-delay timers may be used to create a repeating pulse, as shown in the example of Listing 17.

```
// Manual Pulse Waveform

// Variables:

TIMER0 = new Timer();     // Low timer
TIMER1 = new Timer();     // High timer

function setup() {

}

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

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

Listing 17. *A repeating pulse using a pair of linked on-delay timers (Source: 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.

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 block diagram symbol is shown in Figure 27 below.

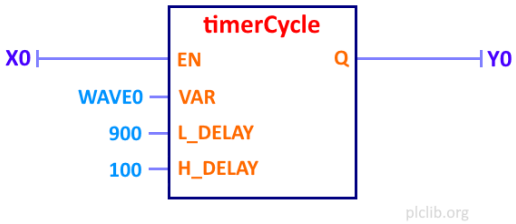

Figure 27. *A cycle timer produces a repeating pulse waveform, when enabled.*

The command uses a named *Waveform* object which holds the elapsed time for the low and high portions of the waveform. The associated low and high pulse widths are defined as parameters of the timerCycle command, as shown in Listing 18.

```
// Pulsed output

// Variables:

WAVE0 = new Waveform();          // Waveform for timerCycle

function setup() {
}

function loop() {
  din (X0);                      // Read Enable input X0 (1 = enable)
  timerCycle(WAVE0, 900, 100);   // Repeating pulse, low 0.9 s, high 0.1 s
                                 // (hence period = 1 second)
  dout(Y0);                      // Send pulse waveform to output Y0
}
```

Listing 18. *Creating a repeating pulse using the timerCycle command (Source: Waveforms > PulsedOutput).*

A typical application of the `timerCycle` command is the creation of a repeating 'alarm armed' flashing pulse, as demonstrated in the *Apps > AlarmArmed* example sketch.

Counting and Counters

A counter is implemented as an *object* and may be configured to count *up*, count *down*, or to operate as a combination of both. Conceptually, a counter has four inputs, two outputs and two internal values, as shown in Figure 28.

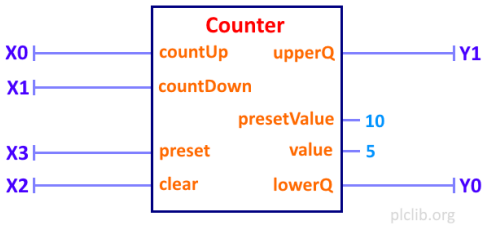

Figure 28. *A graphical model of a 'generic' counter.*

In practice, only a subset of these may be required, depending on the type of counter required. The available counter *methods* are: -

- `countUp` – counts up (increments) on each rising edge.
- `countDown` – counts down (decrements) on each rising edge.

- `preset` – forces the counter to its highest value (*presetValue*) and activates the *upperQ* output.
- `clear` – forces the counter to its lowest value (*0*) and activates the *lowerQ* output.
- `upperQ` – active when the counter has reached its highest value (*presetValue*).
- `lowerQ` – active when the counter has reached its lowest value (*0*).
- `presetValue` – the highest value the counter is allowed to reach, which is configured when the counter object is created.
- `value` – the current internal count, which is allowed to vary between 0 and *presetValue*.

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 JavaScript example creates an up counter which counts from 0 to 10, driven by switch presses applied to input *X0*.

```
// Up Counter - Counts 10 pulses on Input X0 with switch debounce

ctr = new Counter(10);      // Final count = 10, starting at zero
TIMER0 = new Timer();       // Switch debounce timer

function setup() {
}

function loop() {
  din(X0);                  // Read digital input X0
  timerOn(TIMER0, 10);      // 10 ms switch debounce delay
  ctr.countUp();            // Count up

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

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

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

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

Listing 19. *An up counter with a debounced switch input (Source: Counters > CountUp).*

The first step is to create a counter object *ctr*, at the same time setting the maximum count 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, forcing the *lowerQ* output to be activated. Pulses applied to the *X0* input are linked to the *countUp* input, causing the internal *value* property (running count) to be incremented by each 0 to 1 transition. After 10 pulses, the *value* becomes equal to the upper limit or *presetValue*, causing the *upperQ* output to be enabled.

A commonly encountered problem with counter circuits is *switch bounce*, where the switch contacts 'oscillate' 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 value (*presetValue*). Hence pressing the switches connected to inputs *X1* or *X2* will cause the *lowerQ* or *upperQ* outputs to be immediately activated, respectively.

Down Counter

The creation of a *down counter* is quite similar, as shown Listing 20.

```
// Down Counter - Counts 5 pulses on Input X0 with switch debounce

ctr = new Counter(5,1);     // Counts down, starting at 5
TIMER0 = new Timer();       // Switch debounce timer

function setup() {
}

function loop() {
  din(X0);                  // Read digital input X0
  timerOn(TIMER0, 10);      // 10 ms switch debounce delay
  ctr.countDown();          // Count down

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

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

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

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

Listing 20. *A Down counter (Source: Counters > CountDown).*

The first difference is in the syntax associated with creation of the Counter object *ctr*. As well as setting the *presetValue* to 5, the optional second parameter has a value of 1, causing the initial count *value* to be set equal to the upper limit (presetValue), 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*. Once again, a 10 millisecond on-delay timer has been linked in series with the *X0* switch connected to the *countDown* input.

Up/Down Counter

It is straightforward to combine these two approaches, to create a device capable of counting up and down, as shown in Listing 21.

```
// Up-Down Counter

ctr = new Counter(10);      // Counts up or down in range 0-10, starting at zero
TIMER0 = new Timer();       // Switch debounce timer
TIMER1 = new Timer();       // Switch debounce timer

function setup() {
}

function loop() {
  din(X0);                  // Read digital input X0
  timerOn(TIMER0, 10);      // 10 ms switch debounce delay
  ctr.countUp();            // Count up

  din(X1);                  // Read digital input X1
  timerOn(TIMER1, 10);      // 10 ms switch debounce delay
  ctr.countDown();          // Count down

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

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

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

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

```
  dout(Y1);
}
```

Listing 21. *An up-down counter (Source: 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.

The above up/down counter will start at zero (hence *value* property = 0), due to the syntax of the command which creates the counter object. It is also possible to explicitly set the starting value of the counter, by calling the `setValue` method of the counter, as shown in the code extract of Listing 22.

```
function setup() {
  ctr.setValue(5);          // Start counter at 5
}
```

Listing 22. *Setting the starting value of the counter (Source based on: Counters > CountUpDownCustomStart).*

Debugging Counter-based Applications

The output of a counter is activated once the defined number of pulses have been detected, causing the counter to reach either the *presetValue* (when counting up), or zero (if counting down). In effect, the counter exists in two states, being either 'finished' or 'not finished'.

More detailed information may be needed, particularly if switch bounce is suspected! However, the internal count value is not normally directly visible to the user, unless additional debugging code is added. This can be achieved by displaying the *value* property of the counter. The *`console.log`* command may be used in JavaScript, the output of which will be visible in the Console area of the web browser, with debugging options enabled. Similarly, the *`Serial.println`* command may be used in within the Arduino IDE, combined with the *Serial Monitor* window.

Optional debugging code has already been added towards the end of all examples in the *Counters* pull-down menu, which may be enabled by removing the comment characters '//' from the `console.log` command, as shown in the code extract of Listing 23.

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

  // console.log(ctr.value()); // Optionally display current count
  //$delay(200);             // 0.2 second loop delay in C++ for debugging purposes
}
```

Listing 23. *Enabling counter debugging in the JavaScript IDE (Source based on Counters > CountUp).*

The last two code lines may be deleted once debugging is complete, within the browser-based simulator. Alternatively, leave them as shown above, should it be required to continue debugging within the Arduino IDE. Clicking the *Generate code & copy to clipboard* button causes several changes to be automatically made to the equivalent Arduino code. First, the serial interface is enabled, with a default baud rate of 9600 baud. Second the JavaScript *`console.log`* command is replaced with its *`Serial.println`* Arduino equivalent. Finally, the '*`//$`*' escape sequence is removed, hence leaving a useful time delay, which slows the printing rate to approximately 5 messages per second. The resulting Arduino debugging code is shown in the extract of Listing 24.

```
void setup() {
   setupPLC();
  Serial.begin(9600);
}
```

...

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

  // Serial.println(ctr.value()); // Optionally display current count
  delay(200);             // 0.2 second loop delay in C++ for debugging purposes
}
```

Listing 24. *Equivalent counter debugging commands for the Arduino IDE.*

An additional counter debugging option is to connect an LCD display and use it to display the running count. Please see the *Extras* > *CountUpDownLCD* example for more details. (This is designed to work with a Grove LCD display, connected via an I2C interface.)

Shifting and Rotating Binary Data

At its simplest, a *shift register* allows binary data to be shifted to the left or right, by one bit position at a time.

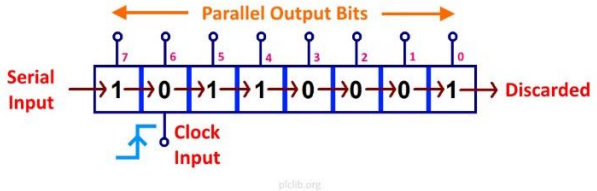

Figure 29. *A simple shift register moves data one place to the right on each rising edge of the Clock.*

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. However, depending on the particular shift register application, the presence or absence of a bit at a particular position is likely to be more important than the overall numerical value of the register.

Note that connecting the output at the right of Figure 29 back to the serial input at the left, would cause the initial content of the register to be recirculated indefinitely. This changes a *shift left* or *shift right* operation into a *rotate left* or *rotate right*, respectively, as will be seen shortly.

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 in plcLib. The initial content of the register may optionally be set at creation. For example, the JavaScript command `shift1 = new Shift(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. (The equivalent Arduino/C++ syntax is `Shift shift1(0x8888);` which is automatically converted by the code generation feature of the web IDE.)

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 block diagram symbol.

Figure 30. *A block diagram representation of the shift register, plus its inputs and outputs.*

The actual coding of the shift register is shown in Listing 25.

```
// Shift register: Shift data to the right

shift1 = new Shift(0x8888);// Create a shift register with initial value 0x8888
TIMER0 = new Timer();      // Define variable used for switch debounce

function setup() {
}

function loop() {

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

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

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

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

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

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

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

Listing 25. *Using a shift register to shift data to the right (Source: 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 switch *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 in the sketch of Listing 26.

```
// Shift register: Shift data to the left

shift1 = new Shift(0x1111);// Create a shift register with initial value 0x1111
TIMER0 = new Timer();      // Define variable used for switch debounce

function setup() {
}

function loop() {

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

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

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

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

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

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

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

```
}
```

Listing 26. *Using a shift register to shift data to the left (Source: 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).

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.

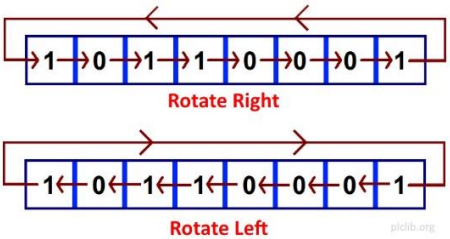

Figure 31. Adding a feedback connection causes data to be rotated right or left in a continuous loop.

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, Listing 27 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.

```
// Shift register: Rotate data to the Right

shift1 = new Shift(0x8000);// Create a shift register with initial value 0x8000
                           // Leftmost 4 bits are rotated to the right
                           // (and bits also continue shifting to the right)

TIMER0 = new Timer();      // Define variable used for switch debounce

function setup() {
}

function loop() {

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

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

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

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

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

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

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

Listing 27. *Rotating data to the right (Source: ShiftRotate > RotateRight).*

An equivalent sketch to rotate a 4-bit data word to the left uses bit 3 (the 4th bit from the right) as the feedback connection, as shown in the example of Listing 28.

```
// Shift register: Rotate data to the Right

shift1 = new Shift(0x8000);// Create a shift register with initial value 0x8000
                           // Leftmost 4 bits are rotated to the right
                           // (and bits also continue shifting to the right)

TIMER0 = new Timer();      // Define variable used for switch debounce

function setup() {
}

function loop() {

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

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

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

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

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

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

  shift1.bitValue(12);      // Send bit 12 value to output Y0
  dout(Y0);

  // console.log(shift1.value());  // Debug shift register
}
```

Listing 28. *Rotating data to the left (Source: ShiftRotate > RotateLeft).*

Working with Analogue Signals

The *ain* 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. Any required scaling of inputs and outputs is performed automatically by the plcLib software. Analogue values may also be compared against threshold values, which in turn allows conditional operation.

The scanValue global variable holds the inputted value, which is in the range 0-1023 for the default 10-bit analogue input resolution. This is automatically scaled in the range 0-255 or 0-180 when used to control a PWM or servo output, respectively.

Controlling LED Brightness using PWM

Linking the *ain* command to the *pout* command, allows a PWM output to be controlled from an analogue input. As an example, the sketch of Listing 29 reads a potentiometer connected to input *AD0* and produces a repeating pulse waveform with a variable duty cycle on output *Y0*.

```
// PWM - Analogue control of a PWM output

function setup() {
}

function loop() {
  ain(AD0);        // Read analogue input AD0
  pout(Y0);        // Send to output Y0 as PWM waveform
}
```

Listing 29. *Analogue control of a PWM output (Source: IO > PWM).*

Operation of the *pout* command is displayed by the JavaScript simulator, which shows the PWM output in orange and with a scaled output in the range 0-255, as shown in Figure 32.

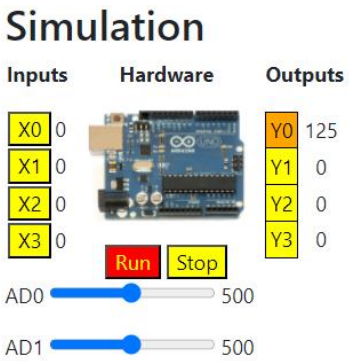

Figure 32. *The output pin linked to pout command is shown in orange and with a scaled output value.*

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. A simple example is given in Listing 30.

```
// Potentiometer and Servo

//$#include <Servo.h>        // Load servo library

//$Servo myServo;            // Create servo object

function setup() {
  //$  myServo.attach(Y0);     // Attach servo to pin Y0
}

function loop() {
  ain(AD0);                  // Read potentiometer connected to Analogue input AD0
  //$scanValue = map(scanValue, 0, 1023, 0, 180);   // Scale ADC value to use with servo (0 - 180 degrees)
  //$myServo.write(scanValue);                      // Write to servo

  // delay(2);                  // Optional 2-40 ms delay
}
```

Listing 30. *Controlling a single Servo (Source: Extras > PotentiometerServo).*

Any Arduino-specific command lines in the above example are preceded by the escape sequence '//$', which causes those lines to be treated as comments by the JavaScript interpreter. These lines are passed unaltered – minus the escape sequence – to the Arduino environment. This means of course that the example cannot be meaningfully simulated in the JavaScript simulator.

Notice in particular, the: -

- inclusion of the servo library,
- creation of a servo object *myServo*,
- attaching the servo object to pin *Y0*,
- reading the analogue input, which is in the range 0-1023,
- using the Arduino *map* command to scale the analogue value to be in the range 0-180 degrees,
- writing this angular value to the servo object
- calling a short time delay, to give the servo time to respond, before the process repeats.

Comparing Analogue Values

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

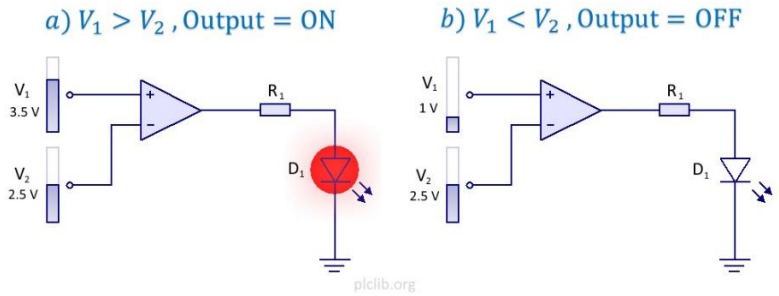

Figure 33. *A comparator 'tests' the relative magnitude of two analogue inputs.*

The circuit symbol for an electronic comparator is triangular, being based on an *operational amplifier*, with two analogue inputs at the left and a single digital output at the left. A high output voltage 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* and *inverting* inputs, respectively). Otherwise, a low output is produced.

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 example of Listing 31 tests whether the analogue voltage on analogue input *AD0* is greater than that on *AD1*, setting output *Y0* if this is true.

```
// Greater than

function setup() {
}

function loop()
{
  ain(AD0);       // Read analogue input AD0
  compareGT(AD1); // AD0 > AD1 ?
  dout(Y0);       // Y0 = 1 if AD0 > AD1, Y0 = 0 otherwise
}
```

Listing 31. *Greater than test between two analogue input pins (Source: Analogue > GreaterThan).*

It is also possible to compare an analogue input against a fixed reference, as shown in the example of Listing 32.

```
// Less Than Threshold

threshold = new Auxiliary(500);

function setup() {
}

function loop()
{
  ain(AD0);              // Read analogue input AD0
  compareLT(threshold);  // AD0 < 500?
  dout(Y0);              // Y0 = 1 if AD0 < 500, Y0 = 0 otherwise
}
```

Listing 32. *Less than test between an analogue input and a fixed threshold (Source: Analogue > LessThanThreshold).*

A slightly more complex example is shown in Figure 34.

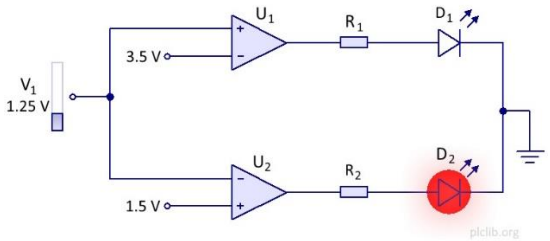

Figure 34. *A pair of comparators test if an input is above or below an allowed 'window' (Source: Analogue > MaxMin).*

This uses a pair of comparators to test whether an analogue input is between an upper and lower threshold.

The upper comparator causes its associated LED to light if the input voltage, 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 in the electronic circuit.)

An equivalent sketch is given in Listing 33, which makes use of both forms of comparator command, together with calculated threshold values, which are based on an assumed 5 V power supply and associated analogue reference voltage.

```
// Max / Min

lowLimit = new Auxiliary(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)

highLimit = new Auxiliary(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)

function setup() {
}

function loop()
{
  ain(AD0);                     // Read analogue input AD0
  compareGT(highLimit);         // AD0 > upper threshold?
  dout(Y0);                     // Y0 = 1 if AD0 > 717, Y0 = 0 otherwise

  ain(AD0);                     // Read analogue input AD0
  compareLT(lowLimit);          // AD0 < lower threshold?
  dout(Y1);                     // Y1 = 1 if AD0 < 307, Y1 = 0 otherwise
}
```

Listing 33. *Maximum / minimum comparator test using fixed threshold values (Source: Analogue > MaxMin).*

Solving Complex Logic Circuits

Larger and more complex combinational logic circuits may require storage of intermediate results as part of their solution. Consider the circuit of Figure 35 as an example, where possible approaches to its solution include use of a temporary variables, or *stack* based logic.

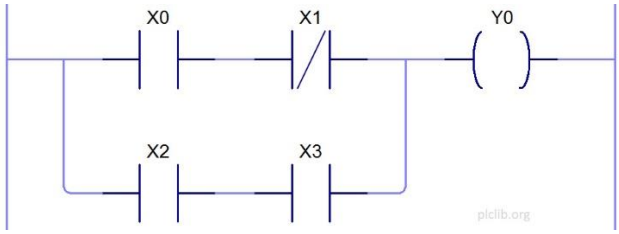

Figure 35. *A multiple branch combinational logic circuit.*

Using Variables with Complex Logic Circuits

The circuit of Figure 35 may be solved by storing the result of the upper branch in a temporary variable, as shown in the example sketch of Listing 34.

```
// Complex Logic

AUX0 = new Auxiliary();

function setup() {
}

function loop() {
                    // Solve first branch
  din(X0);          // Read input X0
  andNotBit(X1);    // AND with inverted input X1
  dout(AUX0);       // Use auxiliary variable AUX0 to store first branch result

                    // Solve second branch
  din(X2);          // Read input X2
  andBit(X3);       // AND with input X3
  orBit(AUX0);      // OR with result from first branch (AUX0)
  dout(Y0);         // Send result to output Y0
}
```

Listing 34. *Solving complex logic with user variables (Source: Variables > ComplexLogic).*

This approach may be scaled to solve combinational logic networks of arbitrary complexity. An alternative is to use a stack-based solution, as discussed in the next section.

Stack-based Storage and Logic

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* it 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* (LIFO) store, as shown in Figure 36.

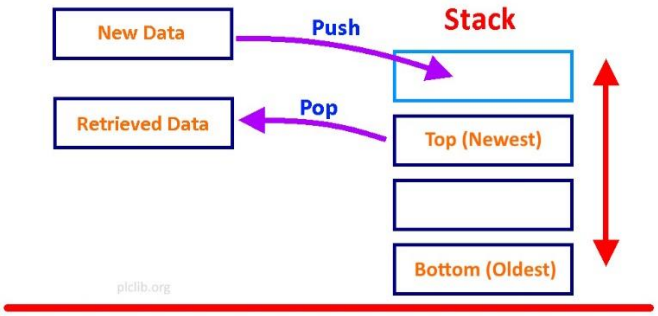

Figure 36. *Using a stack as a last-in first-out data store.*

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

The first step when writing a stack-based sketch is to create a stack object. For example, the JavaScript command `stack1 = new Stack();` creates a stack called stack1, capable of holding up to 32 single-bit values. These may be added or removed from the stack by using the *push* or *pop* methods of the previously created stack object. There is no limit to the number of stacks which may be created, other than the available memory.

The example sketch of Listing 35 demonstrates the use of the stack to store and subsequently retrieve a series of single bit values, but notice that inputs and outputs are displayed in reverse order.

```
// Push and Pop

stack1 = new Stack(); // Create a single-bit stack with 32 levels

function setup() {
}

function loop() {

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

  din(X0);            // Read input X0
  stack1.push();      // Push X0 onto the stack

  din(X1);            // Read input X1
  stack1.push();      // Push X1 onto the stack

  din(X2);            // Read input X2
  stack1.push();      // Push X2 onto the stack

  din(X3);            // Read input X3
  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
  dout(Y0);           // Send to output Y

  stack1.pop();       // Remove X2 value from the stack
  dout(Y1);           // Send to output Y1

  stack1.pop();       // Remove X1 value from the stack
  dout(Y2);           // Send to output Y2

  stack1.pop();       // Remove X0 value from the stack
  dout(Y3);           // Send to output Y3
}
```

Listing 35. *Pushing and Popping values from a single-bit software stack (Source: 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.

Block Logic Operations

A logic network such as that of Figure 37, consisting of two parallel branches, may be solved by using a three step process. The first is to calculate the upper branch, then save (or *push*) this intermediate result on the stack. The second branch may then be solved in the normal way, hence leaving its result in the *scanValue* variable. The third and final step is to combine the current and previous results, by using the `orBlock` method of the stack object, which also removes (or *pops*) the previous result from the stack.

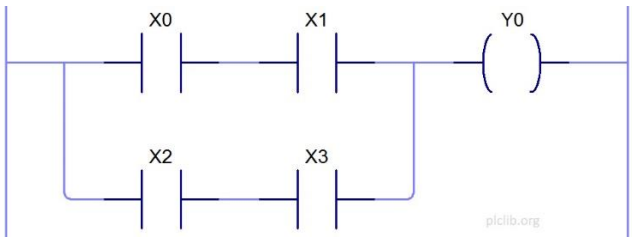

Figure 37. *Using a block-OR command to combine intermediate results in a parallel-branch logic circuit.*

The sketch of Listing 36 demonstrates the approach.

```
// OR Block

stack1 = new Stack(); // Create a single-bit stack with 32 levels

function setup() {
}

function loop() {

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

                      // Calculate second branch
  din(X2);            // Read switch connected to input X2
  andBit(X3);         // Logical AND with input X3

  stack1.orBlock();   // Merge branches using Block OR
  dout(Y0);           // Send result to output Y0
}
```

Listing 36. *Performing a logical OR of two parallel branches using Block OR instruction (Source: Stack > OrBlock).*

A similar technique may be applied with series connections of switch groups, such as that shown in Figure 38, which may be combined using an *andBlock* command.

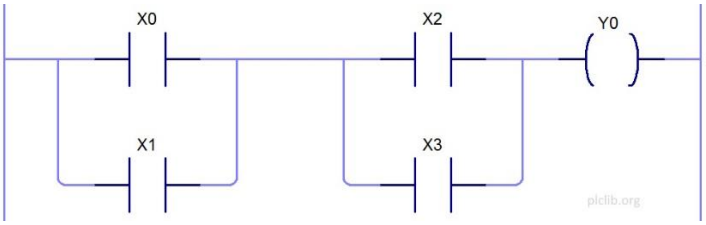

Figure 38. *A Block-AND may be used to solve a complex network consisting of series elements.*

The example of Listing 37 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.

```
// AND Block

stack1 = new Stack();  // Create a single-bit stack with 32 levels

function setup() {
}

function loop() {

                     // Calculate first branch
  din(X0);           // Read switch connected to input X0
  orBit(X1);         // Logical OR with input X1
  stack1.push();     // Push temporary result onto the stack

                     // Calculate second branch
  din(X2);           // Read switch connected to input X2
  orBit(X3);         // Logical OR with input X3

  stack1.andBlock(); // Merge series branches using Block AND
  dout(Y0);          // Send result to output Y0

}
```

Listing 37. *Logical AND of two series switch groups using a Block AND instruction (Source: Stack > AndBlock).*

[To be continued…]