plcLib Reference Manual

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

Version 2.0 Beta 6 | Last updated 23/4/23 by WD

Contents

Command Reference

The following sections give the available commands and syntax of plcLib commands, organised into related functional groups. Readers may also study associated examples from the Web IDE, which illustrate their use.

Many of the examples in this guide make use of timing diagrams to illustrate their operation. These may be produced with the aid of the `LogVars` command, as described in the *Timing Diagrams* section.

Input and Output

Output and input commands write and read information from/to the microcontroller, respectively. Input commands return a value to the calling command and also update the global *scanValue* variable. This in turn enables the value to be processed by subsequent commands. Digital output commands send a 0/1 value, loaded from *scanValue*, to the specified destination. Hence a bare minimum digital I/O example consists of a digital input command (e.g. `din`) immediately followed by a digital output (e.g. `dout`), as shown in Listing 1.

```
// Bare Minimum - Single bit digital input and output

function setup() {
}

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

Listing 1. *Single bit digital input and output (Source: IO > BareMinimum).*

A sample timing diagram is shown in Figure 1.

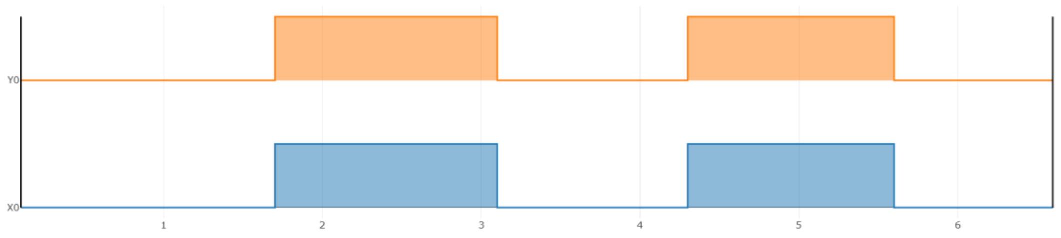


Figure 1. A sample timing diagram for Listing 1, created using the `LogVars` command.

The analogue input command (`ain`) reads the associated analogue to digital converter (ADC). A value in the range 0-1023 is produced by the default 10-bit ADC and stored in the *scanValue* variable. This value may, for example, be compared against a second analogue value or variable by using an analogue comparison command. An analogue value (range 0-1023) may also be automatically scaled and sent to a PWM-enabled output (range 0-255) by using the `pout` command, as shown in Listing 2.

```
// 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 2. *Analogue control of a PWM output (Source: IO > PWM).*

Command	Function	Returns
`din(pin[1]|entity[2]);`	Single bit digital input	*scanValue*[3] = 0|1
`dinNot(pin|entity);`	Single bit digital input (inverted)	*scanValue* = 1|0 (inverted)
`dout(pin|entity);`	Single bit digital output	*scanValue* = 0|1
`doutNot(pin|entity);`	Single bit digital output (inverted)	*scanValue* = 1|0 (inverted)
`ain(pin|entity);`	Analogue input	*scanValue* = 0-1023 (10-bit)
`pout(pin|entity);`	PWM output (0-255)	*scanValue* = 0-1023

Table 1. *Input and output commands (C++, JavaScript).*

Related Examples:

- *IO > BareMinimum*
- *IO > DigitalInputOutput*
- *IO > PWM*

Given that the *scanValue* variable can hold both digital or analogue values (for example 0/1 or 0-1023), it is important that you use sensible combinations of commands in 'input-process-output' program branches (AKA *ladder logic* 'rungs'). For example, an analogue input command (*ain*) followed immediately by a digital output (*dout*) would not make sense, unless an intermediate command was used, such as an analogue comparison (e.g. *compareGT*).

Advanced users may extend the default I/O capabilities by adding 3rd party libraries, either directly to C++ code, or through the use of *escape sequences* in JavaScript. This could for example enable the value stored in the *scanValue* variable to be scaled by the Arduino *map* command and used to control an attached *servo*, as shown in the *3rdParty > Servo* example.

Analogue Comparison

Analogue comparison commands compare a previously inputted analogue value with a second supplied value. The returned value is 1 if the test is true and 0 if it is false. An example is given in Listing 3.

```
// 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 3. *Comparing two analogue input values (Source: Analogue > GreaterThan).*

[1] A pin number refers to a physical pin on the microcontroller.
[2] An entity is a named identifier which may refer to either a variable (such as an Auxiliary), or an object property (such as the *.value()* property of an object).
[3] The *scanValue* variable holds the result of the previous command (if any) and may be updated with the result of the current command. It holds the most recent result and may also maintain a running total (AKA an 'accumulator') for certain mathematical or logical operations. As a rule of thumb, input commands update *scanValue*, based on their result, while output commands expect *scanValue* to have been modified by the result of the previous command from the same program branch. A series of '*implicit if*' statements may be created by placing two or more statements in sequence, in the same branch, where the current statement will conditionally execute, based on the scanValue result from the previous command. The *scanValue* variable is reused by the next program branch, which will typically start afresh with an input command.

Command	Function	Returns
`compareGT(pin|entity);`	Test if previous value (*scanValue*) > supplied value	*scanValue* = 0|1
`compareLT(pin|entity);`	Test if previous value (*scanValue*) < supplied value	*scanValue* = 0|1

Table 2. *Analogue comparison commands (C++, JavaScript).*

Related Examples:

- *Analogue* > `GreaterThan`
- *Analogue* > `GreaterThanThreshold`
- *Analogue* > `LessThan`
- *Analogue* > `LessThanThreshold`
- *Analogue* > `MaxMin`

Single Bit Logical

Single bit logical commands perform the specified Boolean operation between the previous binary value (stored in *scanValue*) and the binary value supplied with the command. The *scanValue* variable is updated with the result.

Command	Function	Returns
`andBit(pin|entity);`	Boolean AND of *scanValue* and supplied value	*scanValue* = 0 | 1
`andNotBit(pin|entity);`	Boolean AND of *scanValue* and supplied value (inverted)[4]	*scanValue* = 0 | 1
`orBit(pin|entity);`	Boolean OR of *scanValue* and supplied value	*scanValue* = 0 | 1
`orNotBit(pin|entity);`	Boolean OR of scanValue and supplied value (inverted)	*scanValue* = 0 | 1
`xorBit(pin|entity);`	Boolean XOR of *scanValue* and supplied value	*scanValue* = 0 | 1
`xorNotBit(pin|entity);`	Boolean XOR of *scanValue* and supplied value (inverted)	*scanValue* = 0 | 1

Table 3. *Single Bit Logical Commands (C++, JavaScript).*

The logical operation may be repeated multiple times if more than two inputs are required, in which case *scanValue* will act as an accumulator. A simple example is given in Listing 4.

```
// 3-input AND using repeated single bit AND commands

function setup() {
}

function loop(){
  din(X0);      // Read digital input X0
  andBit(X1);   // AND with X1
  andBit(X2);   // AND with X2
  dout(Y0);     // Send result to Y0
}
```

Listing 4. *Performing a 3-bit Boolean by using repeated single bit AND commands (Source: Logic > RepeatedAnd).*

The example timing diagram of Figure 2 confirms the output *Y0* (shown in red) is enabled only when inputs *X0* , *X1*, and *X2* are simultaneously active.

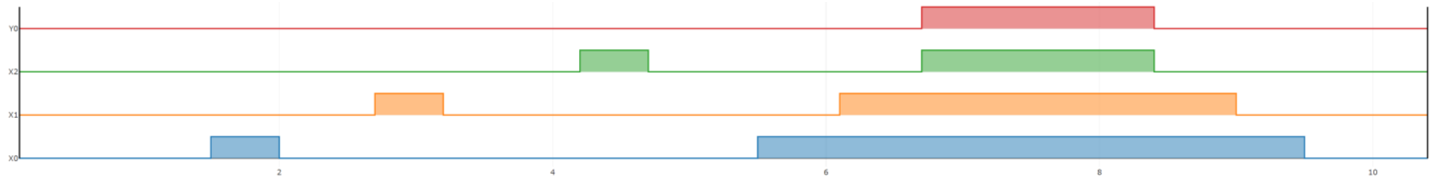


Figure 2. *A sample timing diagram for the logical AND operation of Listing 4 (created using the `LogVars` command).*

[4] Inverting the supplied value in a Boolean logic operation is equivalent to an inversion operation on the input pin (often shown with a circle on the input pin of the logic gate symbol).

Related Examples:

- *Logic* > `AndOrXorNot`
- *Logic* > `InvertedInputLogic`
- *Logic* > `NandNorXnor`
- *Logic* > *RepeatedAnd*

Logical Function Blocks

Logical function block commands perform combinational logic operations. Commands accept 2-4 inputs, with the exception of the *`notFB`* command which performs a logical inversion of a single input. The result is stored in the specified *FunctionBlock* variable (C++) or *FunctionBlock* object (JavaScript), and the *scanValue* variable is updated.

The *FunctionBlock* object/variable must first be created, as shown in Tables 4 (JavaScript) and 5 (C++). (The JavaScript syntax is automatically converted to the C++ equivalent by the code generation feature of the Web IDE.)

Command	Function	Returns
`fbName`[5] `= new FunctionBlock`[6]`();`	Create FunctionBlock.	Object fbName created

Table 4. *Creating a new FunctionBlock object (JavaScript).*

Command	Function	Returns
`FunctionBlock fbName;`	Create FunctionBlock variable.	Variable fbName created

Table 5. *Creating a new FunctionBlock variable (C++).*

Available logical function block commands are given in Table 6.

Command	Function	Returns
`andFB(FbName, pin1|entity1, pin2|entity2, …);`	Boolean AND of *scanValue* and supplied values (2-4 allowed)	FbName = 0 | 1 (*scanValue* is also updated)
`nandFB(FbName, pin1|entity1, pin2|entity2, …);`	Boolean NAND of *scanValue* and supplied values (2-4 allowed)	FbName = 0 | 1 (*scanValue* is also updated)
`orFB(FbName, pin1|entity1, pin2|entity2, …);`	Boolean OR of *scanValue* and supplied values (2-4 allowed)	FbName = 0 | 1 (*scanValue* is also updated)
`norFB(FbName, pin1|entity1, pin2|entity2, …);`	Boolean NOR of *scanValue* and supplied values (2-4 allowed)	FbName = 0 | 1 (*scanValue* is also updated)
`xorFB(FbName, pin1|entity1, pin2|entity2, …);`	Boolean XOR of *scanValue* and supplied values (2-4 allowed)	FbName = 0 | 1 (*scanValue* is also updated)
`xnorFB(FbName, pin1|entity1, pin2|entity2, …);`	Boolean XNOR of *scanValue* and supplied values (2-4 allowed)	FbName = 0 | 1 (*scanValue* is also updated)
`notFB(FbName, pin|entity);`	Boolean Not of *scanValue* and supplied value	FbName = 0 | 1 (*scanValue* is also updated)

Table 6. *Logical function block commands (C++, JavaScript).*

An example sketch is given in Listing 5.

[5] The label *FbName* should be replaced with your function block name.
[6] Function blocks are created as *FunctionBlock* objects ('new' keyword) in JavaScript, but as *FunctionBlock* variables in C++.

```
// 4 input OR gate Function Block

OR0 = new FunctionBlock();

function setup() {
}

function loop() {
  orFB(OR0, X0, X1, X2, X3);    // 2-4 inputs allowed

  din(OR0);      // Read OR0.value
  dout(Y0);      // Send result to Y0
}
```

Listing 5. *Using a logical function block to perform a 4-input Boolean OR (Source adapted from: Logic > OrNor).*

Figure 3 shows a sample timing diagram based on Listing 5. This demonstrates that the output of an inclusive-OR function block is high (true) if one or more of the inputs is high.

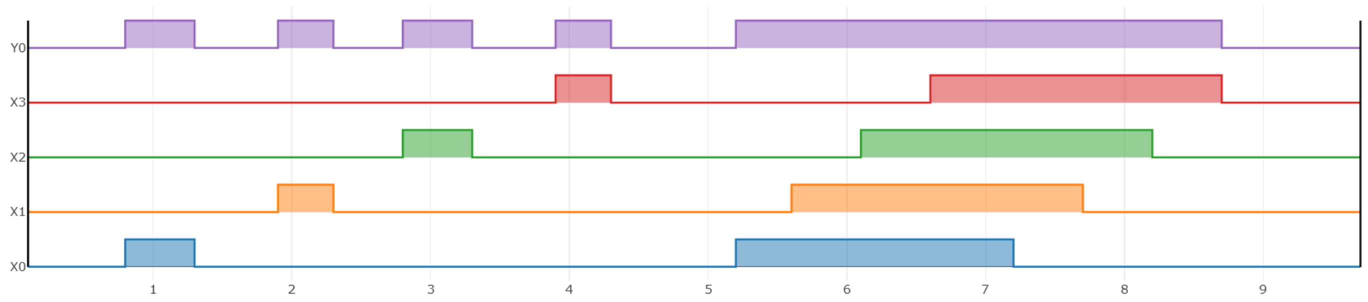


Figure 3. *A timing diagram based on the inclusive-OR function block of Listing 5 (produced using the `LogVars` command).*

Related Examples:

- *FB* > `AndNand`
- *FB* > `OrNor`
- *FB* > `XorXnor`
- *FB* > `NetworkAnd`
- FB > NetworkNotAndOr

Latches

The ***setL*** and ***resetL*** commands force the specified output to change to 1 or 0, respectively when the result from a previous command (such as *din*) causes *scanValue* to change to a 1. The output will remain at the specified value (is 'latched'), even after the input is removed. A matched pair of ***setL*** and ***resetL*** commands is needed to create a set-reset latch, as shown in 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 setL and resetL commands to create a set-reset latch (Source: Latch > SetResetCommands).*

A sample timing diagram is given in Figure 4, based on Listing 6.

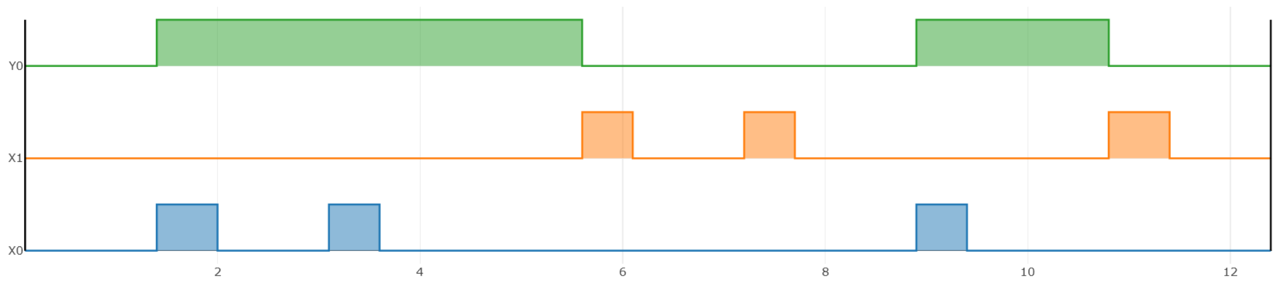


Figure 4. *Using setL and resetL commands to enable or disable an output.*

The *`Latch`* command effectively combines the *`setL`* and *`resetL`* commands, to create a full set-reset latch. The specified output changes to 1 when the result from a previous command makes *scanValue* change to a 1 or remain there. The output will remain at this value, even after the input is removed. A momentary 1 on the *reset* input causes the output value to change to 0 (or remain there), as shown in Listing 7.

```
// 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)
}
```

Listing 7. *Using the latch command to create a set-reset latch (Source: Latch > LatchCommand).*

A key characteristic of a latch is how it behaves when the *Set* and *Reset* inputs are applied at the same time. The *`Latch`* command is designed to be 'reset dominant', so *Set = Reset = 1* causes the output to be disabled. In the case of linked *`setL`* and *`resetL`* commands with outputs connected directly to an output pin, then each command will directly update the output, in sequence. This will generate a pulse waveform, when *Set* and *Reset* are simultaneously applied – which is typically not desired! This behaviour may be overcome by outputting firstly to an Auxiliary variable and then subsequently updating the output pin, as shown in the *`Latch >`* *`SetResetCommandsAuxiliary`* example. It is possible to design latches which are either set- or reset-dominant by using this approach.

A summary of latch related commands is given in Table 7.

Command	Function	Returns
`setL(outPin|outEntity);`	Output value changes to 1 if the input (read from *scanValue*) is momentarily 1. No change to the output if the input = 0.	Output value = 0|1 (*scanValue* is also updated)
`resetL(outPin|outEntity);`	Output value changed to 0 if the input (read from *scanValue*) is momentarily 1. No change to the output if the input = 0.	Output value = 0|1 (*scanValue* is also updated)
`latch(outPin|outEntity, resetPin|resetEntity);`	Output value changed to 1 if *scanValue* is 1, changed to 0 if Reset value is 1, no change otherwise.	Output value = 0|1 (*scanValue* is also updated)

Table 7. *Latch commands (C++, JavaScript).*

Related Examples:

- *`Latch`* `> LatchCommand`
- *`Latch`* `> SetResetCommands`
- *`Latch`* `> SetResetCommandsAuxiliary`

- *Latch* > SetResetEdgeTriggered

Timers

Timer commands are used to create a delayed response to a change in an input signal. For example, an 'on-delay timer' delays turning an output on, while an 'off-delay timer' delays turning an output off. The first step is to create the timer, as explained in Tables 8 and 9, for JavaScript and C++ respectively.

Command	Function	Returns
`TmrName[7] = new Timer[8]();`	Create Timer and initialise elapsed time value to zero.	Object *TmrName* created and initialised.

Table 8. *Creating a new Timer object (JavaScript).*

The JavaScript syntax of Table 8 is automatically converted to that of Table 9 by the code generation feature of the Web IDE.

Command	Function	Returns
`Timer TmrName;`	Create Timer variable and initialise elapsed time to zero.	Variable *TmrName* created

Table 9. *Creating a new Timer variable (C++).*

Available Timer commands are given in Table 10.

Command	Function	Returns
`timerOn(TmrName, period);`	Output value = 1 if previous input (*scanValue*) is continuously active for more than *period* milliseconds. Output value = 0 otherwise.	Output value = 0|1 (*scanValue* is also updated)
`timerOff(TmrName, period);`	Output value = 1 after the input is enabled, then stays high for *period* milliseconds after the input (*scanValue*) is removed.	Output value = 0|1 (*scanValue* is also updated)
`timerPulse(TmrName, period);`	Activates an output for a fixed period after a momentary input is applied.	Output value = 0|1 (*scanValue* is also updated)

Table 10. *Timer commands (C++, JavaScript).*

The output of an on-delay timer goes high if the input is continuously active for a time greater than the *period* in milliseconds, and then goes low immediately if the input is disabled. A simple example is given in Listing 8, while the corresponding timing diagram is shown in Figure 5.

```
// Turn-on Delay

TIMER0 = new Timer();

function setup() {
}

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

Listing 8. *Using the timerOn command to create an on-delay timer (Source adapted from: Delays > TimerOn).*

[7] The label *TmrName* should be replaced with your Timer name.
[8] Timers are created as objects ('new' keyword) in JavaScript, but as Timer variables in C++. The associated timer command uses the Timer object/variable to hold the elapsed time in milliseconds since the timer was enabled.

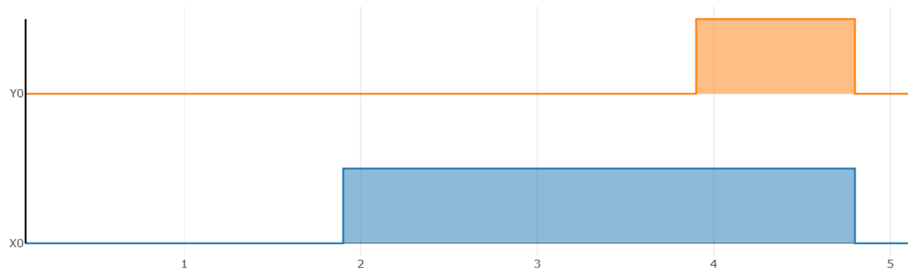


Figure 5. *An example timing diagram for the on-delay timer of Listing 8.*

On-delay timers with periods in tens of milliseconds are commonly used for switch debounce purposes. In this scenario, the brief physical bouncing of the switch contacts is ignored, which in turn greatly reduces the chance of a single switch press causing multiple triggering of a connected system, such as a counter. See the *Delays > SwitchDebounce* example for more details.

An off-delay timer is immediately enabled by its input, but delays turning off until it has been continuously disabled for *period* milliseconds. An off-delay timer example is given in Listing 9, while a corresponding timing diagram is given in Figure 6.

```
// 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 9. *Using the timerOff command to create an off-delay timer (Source: Delays > TimerOff).*

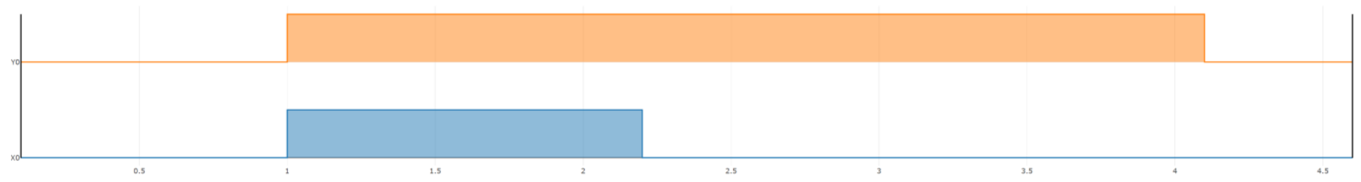


Figure 6. *An example timing diagram for the off-delay timer of Listing 9.*

It is also possible to generate a fixed width pulse by using the *timerPulse* command, whose output is activated for a fixed period after being momentarily enabled. The method of use is otherwise identical to that of the on-delay timer, discussed earlier.

Related Examples:

- *Delays* > TimerOn
- *Delays* > TimerOff
- *Delays* > DelayedPulse
- *Delays* > SwitchDebounce
- *Delays* > FixedPulse

Waveforms

The *timerCycle* command produces a repeating pulse waveform, when enabled by the previous command (via the *scanValue* variable), otherwise the command returns 0. The resulting pulse waveform is typically sent to an output pin by a following *dout* command.

The first step is to create the *Waveform* object, as shown by Table 11, for JavaScript and Table 12 for C++.

Command	**Function**	**Returns**
`WvName`[9] `= new Waveform`[10]`();`	Create Waveform object and initialise elapsed time values to zero.	Object *WvName* created

Table 11. *Creating a new Waveform object (JavaScript).*

The syntax of Table 11 is automatically converted to that of Table 12 by the code generation feature of the Web-based IDE.

Command	**Function**	**Returns**
`Waveform WvName;`	Create Waveform object and initialise elapsed time values to zero.	Object *WvName* created

Table 12. *Creating a new Waveform object (C++).*

Table 13 gives the syntax of the *`timerCycle`* command.

Command	**Function**	**Returns**
`timerCycle(WvName, lowTime, highTime);`	Creates a repeating pulse waveform, when enabled by the previous command.	Output value = 0|1 (*scanValue* is also updated)

Table 13. *Using `timerCycle` to create repeating pulse waveforms.*

The example of Listing 10 creates repeating pulse waveform of period 1 second (low for 0.9 seconds, then high for 0.1 seconds), which is enabled by input *X0*, and with the output sent to output *Y0*.

```
// 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 10. *Using the `timerCycle` command to create a repeating pulse waveform (Source: Waveforms > PulsedOutput).*

An example timing diagram for Listing 8 is given in Figure 7.

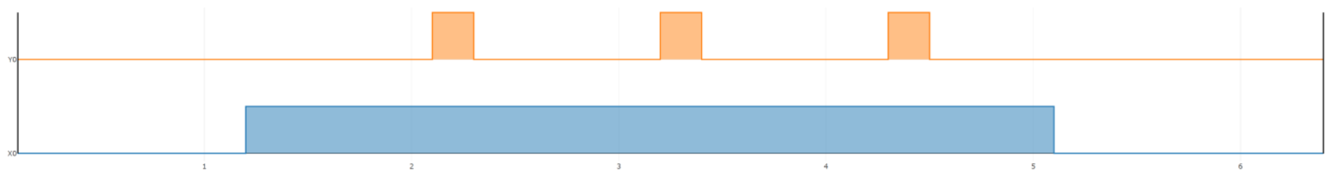


Figure 7. *A sample timing diagram for the timerPulse example of Listing 10.*

Related Examples:

- *`Waveforms`* `> PulsedOutput`
- *`Waveforms`* `> PulsedOutputManual`

[9] The label *WvName* should be replaced with your Waveform name.
[10] Waveforms are created as objects in JavaScript and C++. The associated *`timerCycle`* command uses the *Waveform* object to hold the elapsed time for the low and high portions of the repeating pulse, measured in milliseconds.

- *Waveforms* > `PulsedOutputVariables`

Pulses

A *Pulse* object may be used to create brief (single scan cycle) pulses, triggered by the rising or falling edges of an input signal. This is sometimes called a 'one shot'. The resulting single scan cycle pulse may then be used to trigger following actions, which are intended to occur once only.

As with Waveforms and Timers, the first step is to create the *Pulse* object, as shown in Tables 14 and 15, for JavaScript and C++ respectively.

Command	Function	Returns
`PlsName[11] = new Pulse[12]([0|1]);`	Create new Pulse object.	Object *PlsName* created. The default polarity is positive or rising, but this may be explicitly defined as rising (0) or falling (1).

Table 14. *Creating a new Pulse object (JavaScript).*

The syntax of Table 14 is automatically converted to that of Table 15 by the code generation feature of the Web-based IDE.

Command	Function	Returns
`Pulse PlsName[(0|1)];`	Create new Pulse object.	Object *PlsName* created. The default polarity is positive or rising, but this may be explicitly defined as rising (0) or falling (1).

Table 15. *Creating a new Pulse object (C++).*

The *Pulse* object provides several object-oriented methods, as shown in Table 16.

Parameter / Method	Action	Returns
`PlsName.inClock();`	Connects the previous result (read from *scanValue*) to the input clock of the Pulse.	
`PlsName`	Returns true (1) to the calling command if the default edge type is detected, as specified at pulse creation.	Returns 1 to the calling command if default edge detected, 0 otherwise.
`PlsName.rising();`	Enables following commands in the same branch if the rising edge of the associated input has been detected in the current scan cycle.	Output value = 1 if rising edge detected, 0 otherwise (*scanValue* is also updated).
`PlsName.falling();`	Enables following commands in the same branch if the falling edge of the associated input has been detected in the current scan cycle.	Output value = 1 if falling edge detected, 0 otherwise (*scanValue* is also updated).
`PlsName.expired();`	Clears the rising and falling edge detection flags for the remainder of the current scan-cycle (useful with mutually exclusive sets of actions).	The *`.rising()`* and *`.falling()`* methods return 0 for the remainder of the current scan cycle.

Table 16. *Pulse object command parameters and methods.*

[11] The label *PlsName* should be replaced with your Pulse name.
[12] Pulses are created as objects in both JavaScript and C++.

The example of Listing 11 illustrates the generation of one-shot (single scan cycle) pulses on outputs *Y0* and *Y1*, based on the rising and falling edge, respectively, of input *X0*.

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

myPulse = new Pulse();  // Create 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 11. *Using a Pulse object to detect rising and falling edges of an input (Source: Waveforms > PulseOneShot).*

Notice that an escape sequence (//$) is used towards the end of the above example, to automatically insert a 0.2 second delay into the C++ version, hence making the pulses visible. This line is ignored by the JavaScript version, but should be removed once debugging of C++ code is complete.

A sample timing diagram is given in Figure 8.

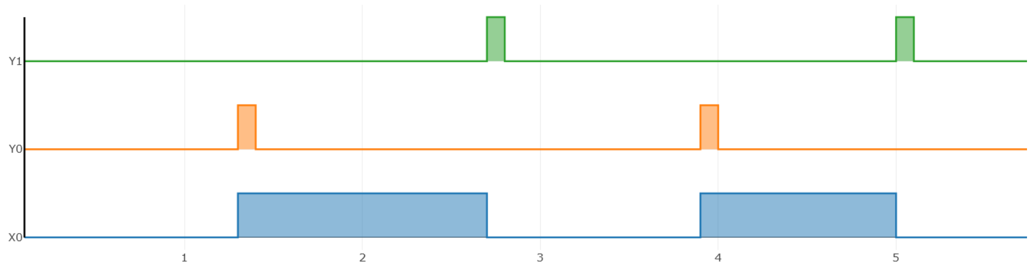


Figure 8. *Using a Pulse to detect rising and falling edges of an input signal.*

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. The example of Listing 12 creates two *Pulse* objects, with *P0* defaulting to 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* is 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 12. *Solving complex logic with user variables (Source: Variables > ComplexLogic).*

Related Examples:

- *Waveforms* > `PulseOneShot`
- *Waveforms* > `PulseOneShotDefault`
- *Latch* > `SetResetEdgeTriggered`

See also examples in the *Apps* and *SFC* folders.

Counters

Counters are used to count the number of events that have occurred. At its simplest, the counter activates an output to indicate that the count is complete, once the specified number of events have been recorded. The first step is to create the *Counter* object, as given in Tables 17 and 18.

Command	Function	Returns
`CtrName`[13] `= new Counter`[14]`(maxValue[, direction]);`	Create new *Counter* object. Defaults to an up counter if *direction* is 0 or not supplied. Acts as a down counter if *direction* = 1.	Object *CtrName* created. Starting *value* = 0 for an up counter. Starting *value* = *maxValue* for a down counter.

Table 17. *Creating a new Counter object (JavaScript).*

The syntax of Table 17 is automatically converted to that of Table 18 by the code generation feature of the Web-based IDE.

Command	Function	Returns
`Counter CtrName(maxValue[, direction]);`	Create new *Counter* object. Defaults to an up counter if *direction* is 0 or not supplied. Acts as a down counter if *direction* = 1.	Object *CtrName* created. Starting *value* = 0 for an up counter. Starting *value* = *maxValue* for a down counter.

Table 18. *Creating a new Counter object (C++).*

Available counter methods are given in Table 19.

Method	Action	Returns
`CtrName.preset();`	Sets the *value* property of the counter to the maximum, if enabled by the previous command. Hence the *.upperQ()* method will return 1 or true.	*value* property = *maxValue*.
`CtrName.clear();`	Sets the *value* property of the counter to the minimum (0), if enabled by the previous	*value* property = 0.

[13] The label *CtrName* should be replaced with your Counter name.
[14] Counters are created as objects in both JavaScript and C++.

	command.. Hence the *.lowerQ()* method will return 1 or true.	
`CtrName.setValue(myCount);`	Sets the *value* property of the counter to *myCount* (so specifies the start count of the counter).	*value* property = *myCount*.
`CtrName.upperQ();`	Tests whether the counter has reached its maximum value.	Returns 1 if *value* = *maxValue* (*scanValue* is also updated).
`CtrName.lowerQ();`	Tests whether the counter has reached its minimum value (0).	Returns 1 if *value* = 0 (*scanValue* is also updated).
`CtrName.value();`	Returns the current counter *value*.	Returns the current *value* property (useful for debugging).
`CtrName.countUp();`	Counts up by one on the rising edge of the previous input (read from *scanValue*).	
`CtrName.countDown();`	Counts down by one on the rising edge of the previous input (read from *scanValue*).	

Table 19. *Counter methods.*

In the case of an up-counter, the initial count *value* is set to 0 when the counter is created, while the final value is set to the specified maximum value. The counter then counts up by using the *.countUp()* method, which is turn driven by the preceding input signal. The *.upperQ()* method becomes true once the internal count value reaches the specified maximum value. This process is illustrated by Listing 12.

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

ctr = new Counter(5);       // Final count = 5, 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);

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

Listing 12. *An up counter which counts from 0 to 5 (Source adapted from: Counters > CountUp).*

The example of Listing 12 also has *clear* and *preset* inputs, which are connected to inputs *X1* and *X2* respectively. These make use of the similarly named *.clear()* and *.preset()* methods. These methods set the internal counter *value* to be either 0 or the maximum (if enabled by the previous command), hence also updating outputs *Y0* and *Y1* via the *.lowerQ()* and *.upperQ()* methods, respectively.

A sample timing diagram is shown in Figure 9, based on the up counter of Listing 12.

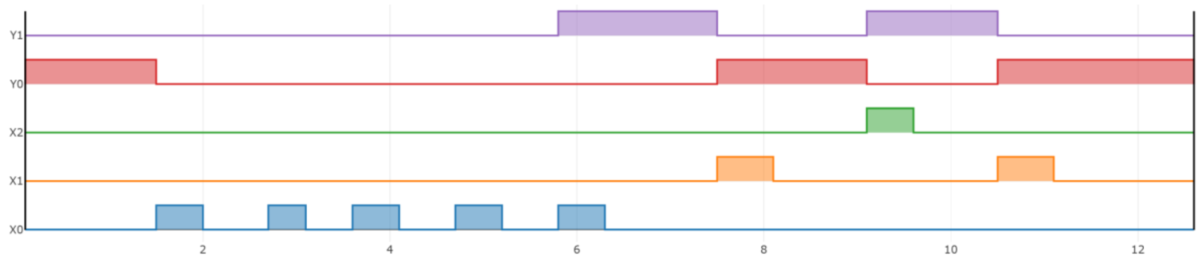


Figure 9. *A sample timing diagram based on the up counter of Listing 12.*

Given that counters fundamentally exist in two states – *finished* or *not finished* – in terms of the counter output, it is sometimes useful to view the current internal count *value*, for debugging purposes. This may be achieved by calling the *.value()* method of the counter, from within a debugging command such as *console.log()* (JavaScript) or *Serial.println()* (C++), as shown towards the end of Listing 12, above.

A common issue with counters, particularly those driven by input switches, is switch bounce. This may be overcome by the addition of a switch debounce delay, as given in the following examples.

Related Examples:

- *Counters* > CountUp
- *Counters* > CountDown
- *Counters* > CountUpDown
- *Counters* > CountUpDownCustomStart
- *Counters* > DualCounters
- *Counters* > CountUpDownLCD

Shift Registers

Shift registers allow a binary bit pattern to be first loaded into a register, which may then to be moved to the left or right. In a shift register the outgoing bit is discarded. However, the outgoing bit may optionally be fed back to the start, which then allows the original bit pattern to be rotated to the left or right. The register has a width of 16-bits, with available bit positions in the range 0-15. The first step is to create the shift register object, as given in Tables 20 and 21.

Command	Function	Returns
`SfrName`[15] `= new Shift`[16]`(value);`	Create new shift register object with the specified initial 16-bit value.	Object *SfrName* created.

Table 20. *Creating a new shift register object (JavaScript).*

The syntax of Table 20 is automatically converted to that of Table 21 by the code generation feature of the Web-based IDE.

Command	Function	Returns
`Shift SfrName();`	Create new shift register object with the specified initial 16-bit value.	Object *SfrName* created.

Table 21. *Creating a new shift register object (C++).*

Available shift register methods are given in Table 22.

Method	Action	Returns
`SfrName.bitValue(position);`	Reads the binary value of the bit at the specified bit position (0-15).	Returns 1/0 (*scanValue* is also updated).

[15] The label *SfrName* should be replaced with your Shift Register name.
[16] Shift registers are created as objects in both JavaScript and C++.

`SfrName.value();`	Reads the value of the shift register as a 16-bit number (0-65,535).	Returns the value of the shift register (*scanValue* is also updated).
`SfrName.reset();`	Clears the shift register to 0.	
`SfrName.inputBit();`	Configures the bit value to be shifted-in to the shift register, based on the result of the previous command (read from *scanValue*).	
`SfrName.shiftLeft();`	Shifts the register one position to the left, replacing the rightmost bit with the value configured using the `.inputBit` method.	
`SfrName.shiftRight();`	Shifts the register one position to the right, replacing the leftmost bit with the value configured using the `.inputBit` method.	

Table 22. *Shift register methods.*

The example of Listing 13 creates a shift register with an initial value of 0x1111 (0001 0001 0001 0001 binary), which is then shifted to the left on each rising edge of input *X1*.

```
// 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 13. *A simple shift register example which shifts data to the left (Source: ShiftRotate >ShiftLeft).*

A typical timing diagram is given in Figure 10, based on the shift register of Listing 13.

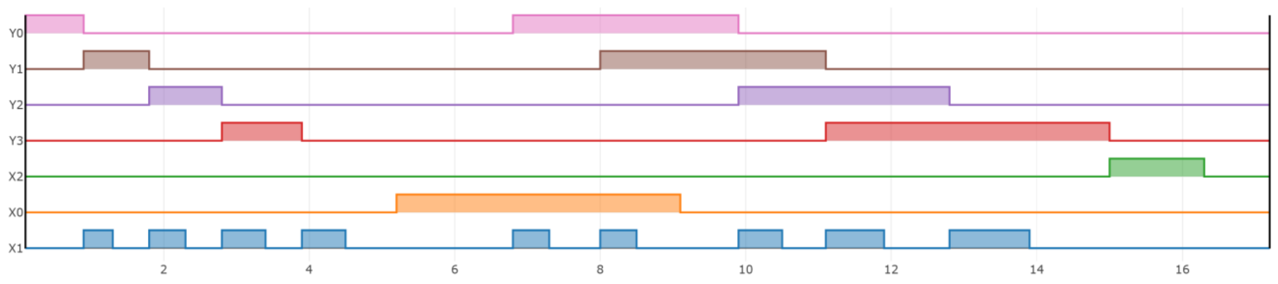


Figure 10. *An example timing diagram based on the shift register of Listing 13.*

A common issue with shift registers, particularly those driven by input switches, is switch bounce. This may be overcome by the addition of a switch debounce delay, as given in the following examples.

Related Examples:

- *ShiftRotate* > RotateLeft
- *ShiftRotate* > RotateRight
- *ShiftRotate* > ShiftLeft
- *ShiftRotate* > ShiftLeftDebug
- *ShiftRotate* > ShiftRight
- *ShiftRotate* > ShiftRightCascaded

Variables

General purpose *user variables* or *auxiliaries*[17] may be used for temporary data storage. This is in addition to the data values associated with physical inputs, outputs and the data objects (e.g. counters, timers and shift registers). Commands may write to or read from these user variables in the same way as to physical outputs or inputs, respectively. Applications of Auxiliaries include: -

- buffering inputs and outputs to avoid the possibility of the same input or output being read from or written to more than once in a single scan cycle.
- Using variables to hold intermediate results of complex calculations. (See also *Stacks*, in the next section.)

Each variable is stored internally as a 32-bit unsigned integer, enabling it to hold either Boolean true/false (1/0) values, or positive integer numbers, depending on the application[18].

User variables or Auxiliaries may be created in both JavaScript and C++, as given in Tables 23 and 24, respectively.

Command	Function	Returns
`varName`[19] `= new Auxiliary`[20]`([value]);`	Create new user variable.	Variable *varName* is created and optionally allocated an initial value.

Table 23. *Creating a new user variable (JavaScript).*

The syntax of Table 23 is automatically converted to that of Table 24 by the code generation feature of the Web-based IDE.

[17] Other names include *auxiliary relays*, *internal relays* or *memory bits*.
[18] Care is needed, to ensure that any associated command sequences are compatible, given that the library does not perform any form of type checking.
[19] The label *varName* should be replaced with your variable name.
[20] Auxiliaries are created as simple objects in JavaScript and as 32-bit integers in C++.

Command	Function	Returns
`Auxiliary varName [= value];`	Create new user variable.	Variable *varName* is created and optionally allocated an initial value.

Table 24. *Creating a new user variable (C++).*

Listing 14 demonstrates the use of a user variable, or auxiliary, as temporary storage in the solution of a multiple branch combinational logic circuit.

```
// 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 14. *Solving complex logic with user variables (Source: Variables > ComplexLogic).*

Stacks

The stack feature implements one or more *last-in first-out* (LIFO) storage areas, with each 'storage level' holding a single bit. Stacks are useful for storing intermediate results of binary calculations, including the solution of complex combinational logic circuits. Each stack is 1-bit wide, with a depth of 32-bits. Stacks are created as objects in both JavaScript and C++, as given in Tables 25 and 26, respectively.

Command	Function	Returns
`StkName`[21] `= new Stack`[22]`();`	Create new Stack object.	Object *StkName* created.

Table 25. *Creating a new Stack object (JavaScript).*

The syntax of Table 23 is automatically converted to that of Table 24 by the code generation feature of the Web-based IDE.

Command	Function	Returns
`Stack StkName;`	Create new Stack object.	Object *StkName* created.

Table 26. *Creating a new Stack object (C++).*

Available stack-related methods are given in Table 27.

Method	Action	Returns
`StkName.push();`	Places the previous binary result (from *scanValue*) onto the stack.	
`StkName.pop();`	Remove the top value from the stack, updating *scanValue* with the result.	*scanValue* = 0/1.
`StkName.orBlock();`	Remove the top value from the stack, performing a logical OR operation with the	*scanValue* = 0/1.

[21] The label *StkName* should be replaced with your Stack name.

[22] Stacks are created as objects in both JavaScript and C++.

	result of the previous command (from *scanValue*). The *scanValue* variable is updated with the result.	
`StkName.andBlock();`	Remove the top value from the stack, performing a logical AND operation with the result of the previous command (from *scanValue*). The *scanValue* variable is updated with the result.	*scanValue* = 0/1.
`StkName.value();`	Returns the current value of the stack as a 32-bit unsigned number (useful for debugging).	Returns the stack as a 32-bit binary number (*scanValue* is also updated).

Table 27. *Available stack-related methods.*

The example of Listing 15 is based on two pairs of parallel switches (*X0* in parallel with *X1*, plus *X2* in parallel with *X3*, both of which are equivalent to OR). These are then connected in series (equivalent to AND), with the intermediate result from the first OR command temporarily stored on a stack.

```
// 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 15. *Using a Stack to solve a complex combinational logic network (Source: Latch > AND Block).*

The timing diagram of Figure 11 shows a typical output based on Listing 14.

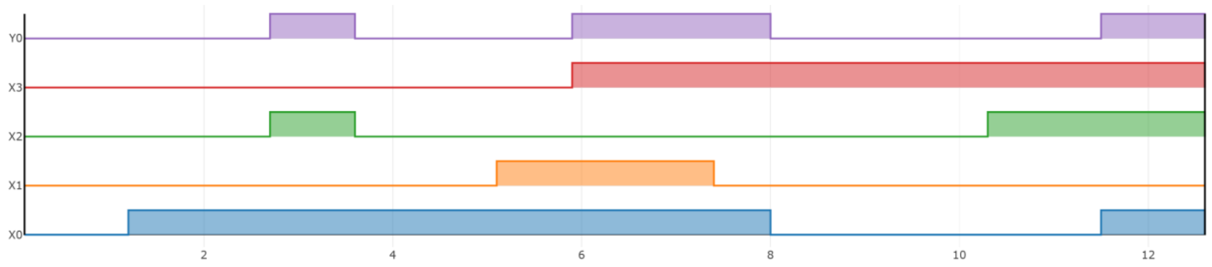


Figure 11. *A sample output from the complex combinational logic system of Listing 14.*

Notice from Figure 11, that the output *Y0* is active if one or more of *X0*/*X1* are active, while one or more of *X2*/X3 is simultaneously active.

Related Examples:

- *Stack* > AndBlock
- *Stack* > OrBlock
- *Stack* > PushPop

States

The combination of *states* (discussed here), plus *transitions* and *events* (discussed in the next section) may be used to create a wide variety of *finite state machine* (FSM) style systems. Supported types include: -

- PLC-style Sequential Function Charts (SFCs)
- Moore-style Finite State Machines (FSMs)
- Mealy-style FSMs
- Hierarchical FSMs

The *State* command is used to create a group of states, each of which may be enabled or disabled at start-up.

Note that *States* are often referred to as '*Steps*' in PLC-style SFCs, although these terms are broadly equivalent, as far as plcLib functionality is concerned. A key feature of steps is that more than one step can be active at any time, as in a parallel branch for example. In traditional FSMs, only one state can be active at any time. However, *parallel* or *orthogonal* (independent) operation is permitted, in which each individual FSM would have one active state.

A further consideration is the sequential nature of microcontroller operations, in which each discrete task takes a miniscule period of time. Thus, two events which appear to happen simultaneously, in fact take place one after another, even if the difference between them is measured in microseconds. In simple terms, a plcLib-based system is not truly synchronous, but approximates towards this 'ideal'. Consider the SFC-style parallel branch of Figure 12 as an example.

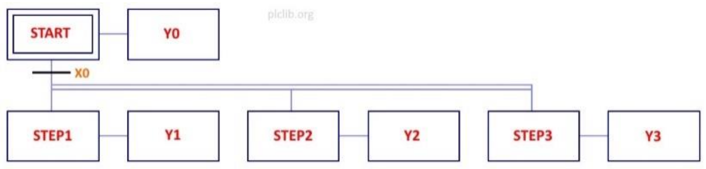


Figure 12. An SFC style parallel branch.

The intended operation is that the system is initially in the *START* step (or state). Trigger event *X0*, which might be a rising edge, causes a simultaneous transition to *STEP1*, *STEP2* and *STEP3*, which also disables the *START* step. In practice the microcontroller performs each task sequentially, so four separate sub-operations are involved. Furthermore, the actions of updating outputs *Y0* to *Y3* are performed in sequence, although at high speed, hence involving a further four sub-tasks. This might not matter in a traditional PLC, in which outputs may be updated synchronously, once calculations are complete, at the end of the scan cycle. However, the plcLib library uses *direct output*, while the Arduino-based system on which it runs uses bit-oriented *digitalWrite* commands to update outputs. Hence, a limitation of plcLib-based FSMs is the potential for brief glitches to occur, as the microcontroller performs each sub-task of a more complex operation, in sequence. This may not matter in systems where inputs and outputs change at relatively low speeds, compared to the operating speed of the microcontroller, and the presence of transitory signals is not a concern. However, potential users should carefully evaluate the suitability of the plcLib system, before deciding to use it.

States are created as objects in both JavaScript and C++, as given in Tables 28 and 29, respectively.

Command	**Function**	**Returns**
`StName`[23] `= new State`[24]`(1/0);`	Create new *State* object *StName*, which is either enabled (1) or disabled(0) at startup.	Object *StName* created and either enabled or disabled.

Table 28. *Creating a new State object (JavaScript).*

[23] The label *StName* should be replaced with your State name.
[24] States are created as objects in both JavaScript and C++.

The syntax of Table 26 is automatically converted to that of Table 27 by the code generation feature of the Web-based IDE.

Command	Function	Returns
`State StName(1/0);`	Create new State object StName, which is either enabled (1) or disabled(0) at startup.	Object *StName* created and either enabled or disabled.

Table 29. *Creating a new State object (C++).*

Available state-related methods are given in Table 30.

Method	Action	Returns
`StName.active();`	Reads the enabled / disabled value of the state.	Returns 0 (state is disabled) or 1 (state is enabled). The *scanValue* variable is also updated.
`StName.enable();`	Enables the state.	State *StName* is enabled and the *`.entry()`* method returns true for the remainder of the scan cycle.
`StName.disable();`	Disables the state	State *StName* is disabled and the *`.exit()`* method returns true for the remainder of the scan cycle.
`StName.entry();`	Returns 1 if the state has been enabled in the current scan cycle, hence providing an entry event for the state.	Returns 1 if state *StName* has been entered during the current scan cycle (*scanValue* is also updated).
`StName.exit();`	Returns 1 if the state has been disabled in the current scan cycle, hence providing an exit event for the state.	Returns 1 if state *StName* has been exited during the current scan cycle (*scanValue* is also updated).
`StName.back();`	Reverts to the previous state, if any. It is equivalent to performing the previous *`trans`* command in reverse.	Enables the previous state (if any) and disables the current state. Entry and exit events are triggered.

Table 30. *Available State-related methods.*

An example is given in Listing 16. This implements a *Sequential Function Chart* (SFC) style system, having a switch based parallel branch, followed by a converge.

```
// Parallel Switch Branch with Converge

                         // Define state/step names & values
START = new State(1);    // Start-up step (START = 1)
STEP1 = new State(0);    // Other steps disabled (= 0)
STEP2 = new State(0);
STEP3 = new State(0);

                         // Define Pulse object(s) for edge triggering
P0 = new Pulse();        // Edge detect on X0
P1 = new Pulse();        // Edge detect on X1

function setup() {
}

function loop() {

                         // Read input(s)
```

```
  din(X0);
  P0.inClock();          // Connect X0 to P0 Pulse

  din(X1);
  P1.inClock();          // Connect X1 to P1 Pulse


                         // Do transitions

  P0.rising();           // Detect rising edge of X0 (or use 'din(P0);')
  trans(START, STEP1);   // Transition START > STEP1
  STEP2.enable();        // Also enable STEP2

  P1.rising();           // Detect rising edge of X1 (or use 'din(P1);')
  trans(STEP1, STEP3);   // Transition STEP1 > STEP3
  STEP2.disable();       // Also disable STEP2


                         // Display current step
  din(START);
  dout(Y0);              // Send to output Y0

  din(STEP1);
  dout(Y1);              // Send to output Y1

  din(STEP2);
  dout(Y2);              // Send to output Y2

  din(STEP3);
  dout(Y3);              // Send to output Y3
}
```

Listing 16. *A SFC style system, which implements a switch-based parallel branch and converge (Source: SFC > ParallelSwitchBranchConverge).*

A sample timing diagram is given in Figure 13, based on Listing 15.

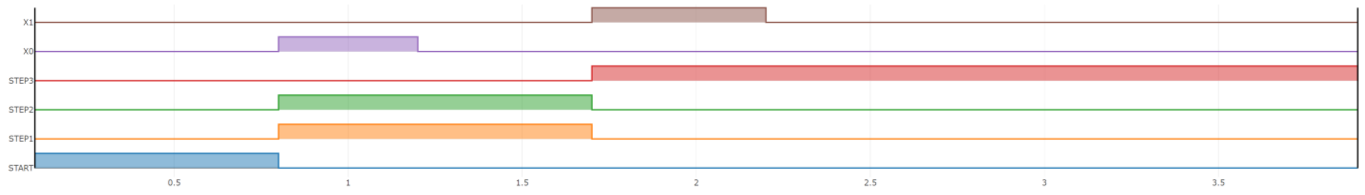


Figure 13. *A sample timing diagram for the switch-based parallel branch and converge of Listing 15.*

The system starts in the *START* step . Pressing *X0* causes *STEP1* and *STEP2* to be simultaneously enabled (a parallel branch). Finally pressing *X1*, cancels *STEP1* and *STEP2*, while enabling *STEP3* (a parallel converge).

Related Examples:

A range of SFC-style examples are available, as shown below.

- *SFC* > `ParallelSwitchBranch`
- *SFC* > `ParallelSwitchBranchConverge`
- *SFC* > `ParallelSwitchBranchDiscrete`
- *SFC* > `RepeatingSwitchSequence`
- *SFC* > `SelectiveSwitchBranch`
- *SFC* > `SelectiveSwitchBranchConverge`
- *SFC* > `SimpleSwitchSequence`
- *SFC* > `SimpleTimedSequence`

See also the FSM folder and several examples from the Apps folder.

Transitions and Events

The `trans` command causes a conditional transition from the source state to the destination state. For the transition to take place, the source state must be currently enabled and the `trans` command must itself have been enabled by the result of the preceding command sequence, via the *scanValue* variable. Table 31 gives the syntax of the `trans` command.

Command	Function	Returns
`trans(State1`[25]`, State2);`	Conditionally enables State2 and disables State1. For the transition to take place, State1 must be enabled and the trans command must itself be enabled by the previous result, via *scanValue*.	Returns 1 if the transition takes place and returns 0 otherwise (*scanValue* is also updated).

Table 31. *The* `trans` *command.*

The `trans` command is level-based, by default, being enabled if the preceding command causes the *scanValue* variable to be set and the source state is active. For example, the followed extract would enable *STATE2*, and disable *STATE1*, if a switch connected to input *X0* is either momentarily pressed, or held in the on position, while *STATE1* is simultaneously active.

```
din(X0);
trans(STATE1, STATE2);
```

Listing 17. *A level-based transition will take place if X0 is enabled and STATE1 is active.*

It is also possible to create edge-based transitions by combining the trans command with *Pulse* objects and their rising/falling edge events. An example is given in Listing 18.

```
// Switch Sequence based on FSM with optional Timing Diagram

s1 = new State(1);      // 1 = active at start
s2 = new State(0);
s3 = new State(0);

P0 = new Pulse();       // Pulse used for edge detection on X0

function setup() {
  logVars("X0", "X1", "X2", "s1", "s2", "s3"); // Variables for timing diagram
}

function loop()
{
  din(X0);
  P0.inClock();         // Link X0 input to P0 Pulse object

  P0.rising();          // Read rising edge of X0 - or use 'din(P0);'
  trans(s1, s2);        // Transition from s1 and s2 using X0 rising edge

  P0.falling();         // Read falling edge of X0
  trans(s2, s1);        // Transition from s2 and s1 using X0 falling edge

  din(X1);
  trans(s2, s3);        // Move from s2 to s3 using X1 (level-based)

  din(X2);
  trans(s3, s2);        // Move from s3 to s2 using X2 (level-based)

  din(s1);              // Display active state on Y0 - Y2
  dout(Y0);

  din(s2);
  dout(Y1);
```

[25] *State1* and *State2* should be replaced by your source and destination state names.

```
  din(s3);
  dout(Y2);

  logVars(X0, X1, X2, s1, s2, s3); // log values to timing diagram
}
```

Listing 18. *A switch-based FSM, using edge-based transitions (Source: FSM > SwitchSequenceWithTimingDiagram).*

A sample timing diagram is given in Figure 14, which was created using the `LogVars` commands of Listing 17. (See the next section for more details.)

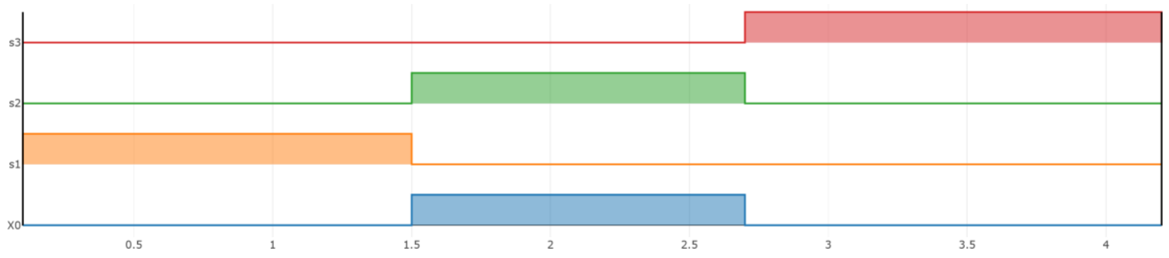


Figure 14. *A sample timing diagram based on the FSM example of Listing 17.*

The above timing diagram confirms that the system starts in state *s1*, based on the initial state definitions. The rising edge of *X0* is used to trigger the transition from *s1* to *s2*, while the falling edge of *X0* causes the final transition from *s2* to *s3*.

In general, edge-based transitions tend to be effective for events which occur at a precise instant, such as button presses. Level-based transitions are useful with signals which have alternative values, such as a temperature threshold sensor, for example. Careful system design is needed in either case, to ensure that FSMs behave as expected under all possible circumstances. A common issue with level-based systems is the rapid and unexpected triggering of multiple transitions. The likelihood of this problem is reduced in event-based systems, but not completely eliminated. Note that the `pulse.expired()` method may be used to clear a rising or falling edge event for the remainder of the scan cycle, hence preventing unwanted duplicate triggering of subsequent commands.

As well as events being used to trigger transitions between states, additional events are also triggered when a state is either enabled or disabled. As their names suggest, an `entry` event is triggered when a state becomes active, while an `exit` event occurs when a state is disabled, each of which lasts for a single scan cycle. These events are typically used to perform 'tidy up' operations, or to display debugging information, as shown in the JavaScript extract of Listing 19.

```
    // Print debugging information to Console
    // (Enable Developer Tools in browser or
    // open Serial Monitor in Arduino IDE.)

    if(DISABLED.entry()) {
        console.log("Entering DISABLED state");
    }

    if(DISABLED.exit()) {
        console.log("Leaving DISABLED state");
    }

    if(HEATING.entry()) {
        console.log("Entering HEATING state");
    }

    if(HEATING.exit()) {
        console.log("Leaving HEATING state");
    }

    if(IDLE.entry()) {
        console.log("Entering IDLE state");
    }

    if(IDLE.exit()) {
        console.log("Leaving IDLE state");
    }
```

Listing 19. *Using entry and exit events to display state-based debugging information (Source based on: FSM > OvenControlMooreDebug).*

Related Examples:

A range of FSM examples is available in the FSM folder, demonstrating the creation of *Moore*, *Mealy* and *hierarchical* style FSMs.

- *FSM* > `OvenControlHierarchical`
- *FSM* > `OvenControlHierarchicalDebug`
- *FSM* > `OvenControlMealy`
- *FSM* > `OvenControlMealyDebug`
- *FSM* > `OvenControlMoore`
- *FSM* > `OvenControlMooreDebug`
- *FSM* > `Alarm`
- *FSM* > `AlarmArmed`
- *FSM* > `AlarmArmedTimeout`
- *FSM* > `AlarmArmedTimeoutIsolate`
- *FSM* > `SwitchSequenceWithTimingDiagram`
- *FSM* > `RobotWalkPauseResume`

See the *SFC* folder listed earlier, for PLC-style examples, and also the *Apps* folder, for state-based applications.

Timing Diagrams (JavaScript only)

The *LogVars* command may be used to create timing diagrams, based on the current inputs, outputs or auxiliary variables. This feature is available in JavaScript only. The *LogVars* command must occur twice in the listing, with slightly differing syntax, as shown in Table 30.

Command	Function	Returns
`logVars("entity1", ("entity2", …);`	First occurrence in the *setup* function enables data logging for the named entities (with double quotes).	
`logVars(entity1, (entity2, …);`	Second occurrence in the *loop* function records data for each of the specified entities (without double quotes).	

Table 32. *The LogVars command.*

As an example, the extract of Listing 20 illustrates how the *LogVars* command was added to the earlier example, allowing generation of the timing diagram previously seen in Figure 13.

```
function setup() {
  logVars("X0", "X1", "X2", "s1", "s2", "s3"); // Variables for timing diagram
}
```
…
```
  din(s3);
  dout(Y2);

  logVars(X0, X1, X2, s1, s2, s3); // log values to timing diagram
}
```
Listing 20. *Using the logVars command to gather data for subsequent display of a timing diagram (Source adapted from: FSM > SwitchSequenceWithTimingDiagram).*

In order to generate the timing diagram, the recommended sequence is.

1. Press *Stop* to clear any previous logging 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 note that any timing diagrams generated will be an approximation to the actual timing of signals. Hence, these should not be relied on as being accurate, for measurement purposes, or for determination of the precise sequence of events. This is due to the repeating nature of the PLC scan cycle, combined with the finite sampling rate of the *LogVars* command.

Default Settings and Configuration Options

The following sections contain information which will be useful if you need to understand, or change the configuration of the JavaScript-based IDE, or the associated C++ library.

JavaScript IDE and Simulator

The JavaScript-based *Integrated Development Environment* (IDE) and simulator is available from https://plclib.org/live/.

It may in future be possible to download and install your own copy – onto a locally running webserver (*localhost*) for example. This option may be of interest if you intend to customise or extend the library, or if you require your own exclusive copy. However, detailed instructions related to the installation and configuration of your own copy are beyond the scope of this guide.

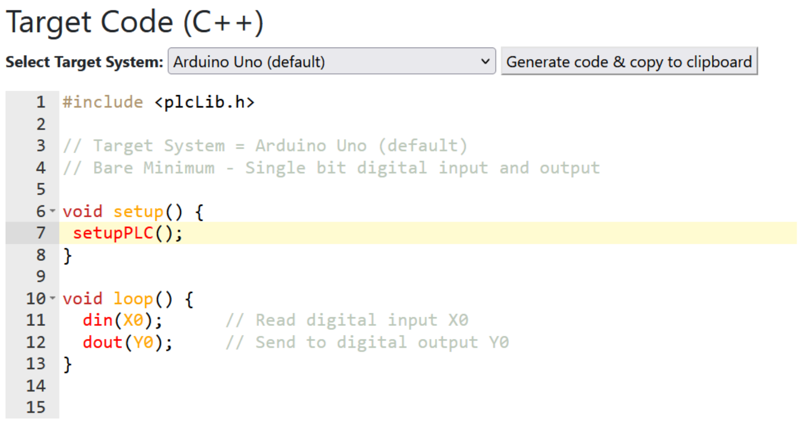
Installation of C++ Library

The C++ library should be installed onto the Arduino IDE prior to attempting to compile and download code to an actual target system. Please follow instructions associated with your Arduino IDE to install the library.

Default Pin Allocations

Default pin allocations and associated configuration settings are automatically generated for supported hardware, once this has been selected via the *Select Target System* drop down list and then the *Generate code & copy to clipboard* button is pressed. Any 'custom' pin configuration information will then appear in the *Target Code* editor window.

In the case of the first option on the list, which is *Arduino Uno (default)*, no custom code is generated. Instead, the library uses pre-defined settings in the *setupPLC* function, as found in the 'plcLib.cpp' file. By default, this sets pins *X0-X3* to be inputs, sets pins *Y0-Y3* to be outputs and configures the initial value of pins *Y0-Y3* to be low, where *low* is assumed to correspond to 'off'. Mappings between pin names and pin numbers are found in the 'plcLib.h' file. For example, the line starting `const int X0 = 7;` allocates input *X0* to pin 7 on the Arduino Uno, while the line `pinMode(X0, INPUT);` from plcLib.cpp configures *X0* as an input. The *setupPLC* function is itself invoked by being placed inside the setup function in the *Target Code (C++)* editor window, as shown in Figure 15.

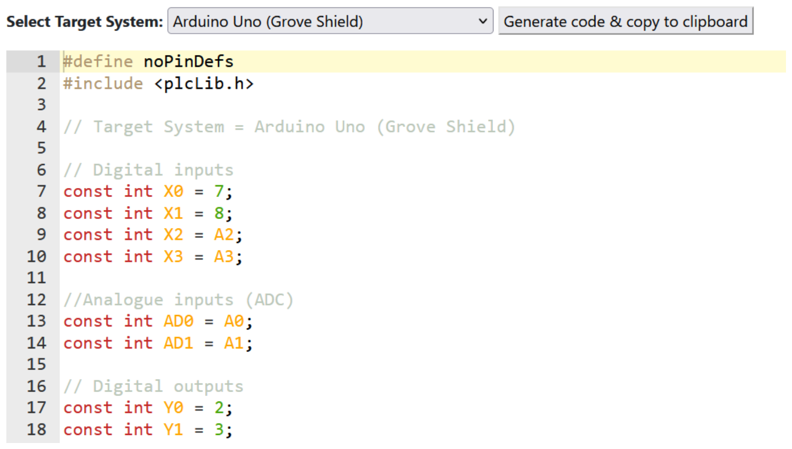

Target Code (C++)
Select Target System: Arduino Uno (default)
Generate code & copy to clipboard

```
#include <plcLib.h>

// Target System = Arduino Uno (default)
// Bare Minimum - Single bit digital input and output

void setup() {
 setupPLC();
}

void loop() {
  din(X0);       // Read digital input X0
  dout(Y0);      // Send to digital output Y0
}
```

Figure 15. *Default pin configurations settings, which include use of the setupPLC function.*

Custom Pin Configurations

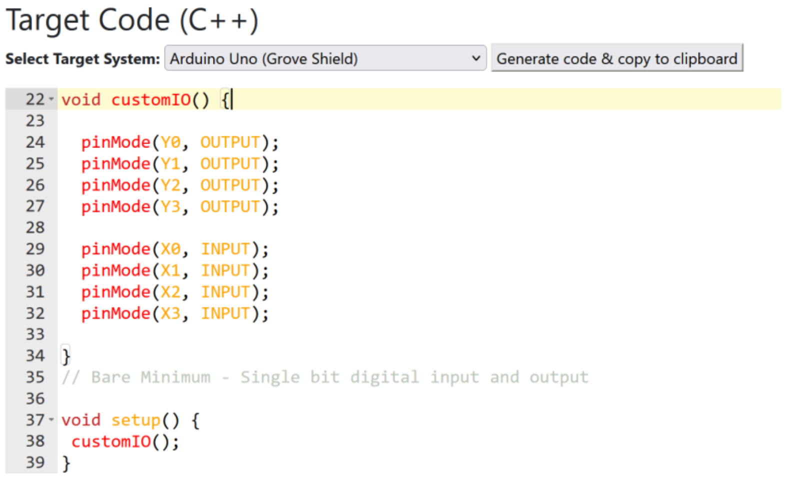
Selecting an option other than *Arduino Uno (default)* in the *Select Target System* drop-down list will cause custom I/O allocations to be generated when the *Generate code & copy to clipboard* button is pressed. The `#define noPinDefs` command is automatically created on the first line of the resulting code, within the *Target Code (C++)* editor window, as shown in Figure 16.


Select Target System: Arduino Uno (Grove Shield)
Generate code & copy to clipboard

```
#define noPinDefs
#include <plcLib.h>

// Target System = Arduino Uno (Grove Shield)

// Digital inputs
const int X0 = 7;
const int X1 = 8;
const int X2 = A2;
const int X3 = A3;

//Analogue inputs (ADC)
const int AD0 = A0;
const int AD1 = A1;

// Digital outputs
const int Y0 = 2;
const int Y1 = 3;
```

Figure 16. *Custom pin configuration, which makes use of the noPinDefs setting.*

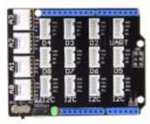
The *noPinDefs* directive, causes the C++ compiler to ignore previously defined pin allocations, associated with the *setupPLC* function. These are then replaced by explicit pin allocation declarations, as shown in Figure 15 above. This is then followed by a *customIO* function definition which is then called from within the *setup* function, hence configuring pin directions, as shown in Figure 17.

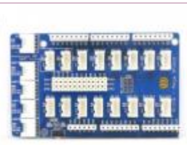

Target Code (C++)
Select Target System: Arduino Uno (Grove Shield)
Generate code & copy to clipboard

```
void customIO() {

  pinMode(Y0, OUTPUT);
  pinMode(Y1, OUTPUT);
  pinMode(Y2, OUTPUT);
  pinMode(Y3, OUTPUT);

  pinMode(X0, INPUT);
  pinMode(X1, INPUT);
  pinMode(X2, INPUT);
  pinMode(X3, INPUT);

}
// Bare Minimum - Single bit digital input and output

void setup() {
 customIO();
}
```

Figure 17. *Configuring pin directions from within the customIO function.*

Default pin mappings are given in Table 33, for currently supported hardware.

Selected Hardware	***Digital Inputs***	***Digital Outputs***	***Analogue Inputs***	***Notes***
Arduino Uno (default)	X0 -> 7 X1 -> 8 X2 -> A2 X3 -> A3	Y0 -> 2 Y1 -> 3 Y2 -> 4 Y3 -> 5	AD0 -> A0 AD1 -> A1	
Arduino Uno (Grove Shield) 	X0 -> 7 X1 -> 8 X2 -> A2 X3 -> A3	Y0 -> 3 Y1 -> 4 Y2 -> 5 Y3 -> 6	AD0 -> A0 AD1 -> A1	Outputs Y0, Y2 and Y3 are PWM-capable, but not Y1.
Arduino Uno (Multifunction Shield)	X0 -> A1 X1 -> A2 X2 -> A3 X3 -> 5	Y0 -> 10 Y1 -> 11 Y2 -> 12 Y3 -> 13	AD0 -> A0 AD1 -> A5	Built-in switches and LEDs are active low. (During testing, an external Grove push-button was connected to pin 5, while a Grove potentiometer was connected to pin A5.)
Arduino MKR (Grove Carrier)	X0 -> A2 X1 -> A3 X2 -> A4 X3 -> A5	Y0 -> 0 Y1 -> 1 Y2 -> 2 Y3 -> 3	AD0 -> A0 AD1 -> A1	
Arduino Mega / Mega 2560 (Grove Carrier)	X0 -> A8 X1 -> A10 X2 -> A12 X3 -> A14	Y0 -> 8 Y1 -> 6 Y2 -> 4 Y3 -> 2	AD0 -> A0 AD1 -> A2	

Adafruit M0 (Grove Shield FeatherWing)	X0 -> A2
	X1 -> A3
	X2 -> A4
	X3 -> A5
		Y0 -> 5
		Y1 -> 6
		Y2 -> 9
		Y3 -> 10
			AD0 -> A0
			AD1 -> A1
				A3 = 2nd pin on A2 connector
				A5 = 2nd pin on A4 connector
				(Grove pins differ from Feather pins)
				Y0 connected to Grove D2
				Y1 = 2nd pin on D2 connector
				Y2 connected to Grove D4
				Y3 = 2nd pin on Grove D4 connector

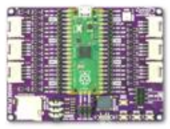
Maker Pi Pico Base 	X0 -> 20 X1 -> 21 X2 -> 22 X3 -> 1 X4 -> 0 X5 -> 3 X6 -> 2	Y0 -> 7 Y1 -> 6 Y2 -> 9 Y3 -> 8	AD0 -> 27 AD1 -> 26	X0-X2 are built-in push buttons (active low) X3 connected to Grove 1, 1st pin X4 (additional) connected to Grove 1, 2nd pin X5 (additional) connected to Grove 1, 1st pin X6 (additional) connected to Grove 2, 2nd pin Grove 3 is intended for I2C devices Y0 connected to Grove 4, 1st pin Y1 connected to Grove 4, 2nd pin Y2 connected to Grove 5, 1st pin Y3 connected to Grove 5, 2nd pin AD0 connected to Grove 6, 1st pin AD1 connected to Grove 6, 2nd pin

Table 33. *Default pin mappings for supported hardware.*

It is relatively straightforward to modify one of the above custom hardware configurations, in the event that a different hardware configuration is required to those directly supported. Possibilities include adding additional pins, changing default pin mappings, or even testing the library with new hardware platforms or microcontrollers.

Consider using dual input/output modules, where access to the second pin on a Grove connector is required. (Dual port devices from the M5Stack range were used during testing.)

Positive and Negative Logic

The C++ version of the library supports inputs and outputs which use either *positive logic* or *negative logic*, and each type may be separately configured. In positive logic, an active input or output will correspond to a high voltage level. Conversely, in negative logic, a low voltage level is associated with an active input or output. Which setting is correct will depend on the connected hardware. Figure 18 gives examples of the four possible combinations of active low inputs and outputs. (These circuits are slightly simplified for clarity.)

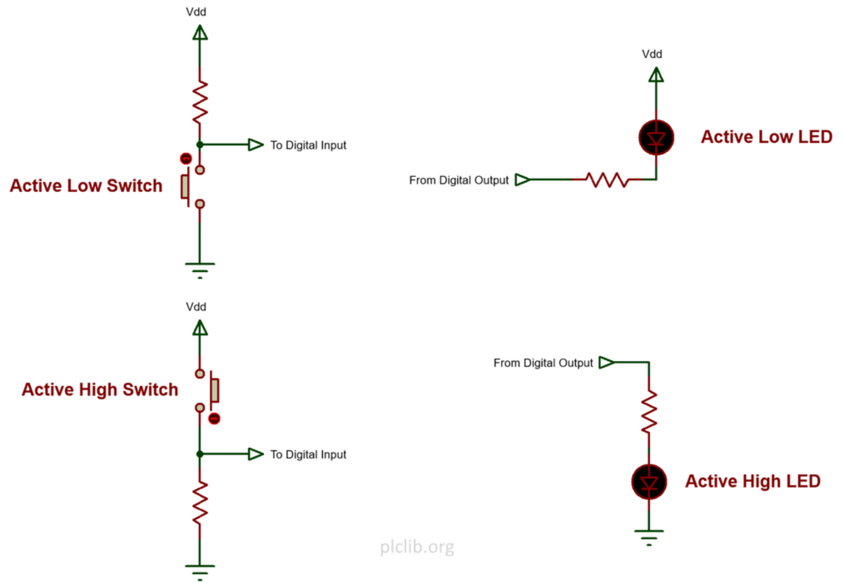


Figure 18. *Examples of positive and negative logic input and output circuits.*

The active low input at the upper left consists of a pull-up resistor *R1*, plus a push button switch *SW1*. With no switch pressed, the pull-up resistor causes the input to read 'high'. Pressing *SW1* overrides the pull-up resistor and forces the input low. Logic levels are reversed in the input circuit at the lower left, with *R3* acting as a weak pull-down resistor, which is overridden by pressing switch *SW2*.

LED *D1* at the upper right has its positive (anode) terminal connected to the positive power supply, and is illuminated when its cathode is pulled low via current limiting resistor *R2* – hence this is an active low output. Conversely LED *D2* at the lower right is illuminated by an active high digital output.

Positive or negative logic inputs and/or outputs may be separately configured to suit the connected hardware, by editing the relevant section of the *plcLib.cpp* file, as shown in Figure 19 below.

```
// Uncomment the following line(s) to enable negative logic inputs and/or outputs
// #define negLogicInputs
// #define negLogicOutputs
```

Figure 19. *Enabling or disabling positive / negative logic inputs and outputs.*

Recommended polarity settings for supported hardware and default I/O devices are shown in Table 34.

Selected Hardware	*Inputs*	*Outputs*	*Notes*
Arduino Uno (default)	Positive	Positive	
Arduino Uno (Grove Shield)	Positive	Positive	
Arduino Uno (Multifunction Shield)	Negative	Negative	Built-in switches and LEDs are active low. (A fourth input switch may be externally connected via headers, as can a second analogue input potentiometer.)

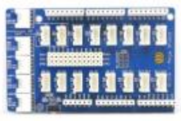
Arduino MKR (Grove Carrier)	Positive	Positive	
Arduino Mega / Mega 2560 (Grove Carrier) 	Positive	Positive	
Adafruit M0 (Grove Shield FeatherWing)	Negative	Positive	Dual port I/O modules are recommended, due to limited connector availability. (During testing, a pair of active low *M5Stack Mini Dual Button Units* was connected to the inputs, while a pair of *M5Stack 2-Channel SPST Relay Units* was connected to outputs.)
Maker Pi Pico Base	Negative	Positive	Built-in switches connected to X0-X2 are active low, so it is recommended to enable active low inputs. Some inputs and outputs use the second Grove connection. (During testing, a pair of active low *M5Stack Mini Dual Button Units* was connected to additional digital inputs (active low), a pair of *M5Stack 2-Channel SPST Relay Units* was connected to digital outputs (active high) and a Grove Joystick was connected to the dual analogue inputs.) Connector Grove3 is recommended to be used with I2C devices (e.g. keypad, LCD).

Table 34. *Recommended polarity settings for I/O devices.*

Loading a Sketch at Startup

The sample sketch is loaded in the editor window, by default. An alternative file may be loaded by specifying the sample path and filename in the URL, as shown in the following examples.

Relative path: - https://plclib.org/live/?url=examples/Latch/LatchCommand.txt

Absolute path: - https://plclib.org/live/?url=/live/examples/Latch/LatchCommand.txt

Notice in the first example a relative URL is given, starting at the current web folder (so '/live/' is assumed). However, in the second example, the sample file URL begins with a '/', so the full path from the web-root must be given.

Program Features

The following sections describe a number of program features, which have not been covered in preceding sections. This includes the internal operation method of the system (the *scan cycle*), methods of debugging sketches, and how to load or save your own sketches.

Scan Cycle Operation.

The plcLib software operates by repeatedly reading inputs, performing calculations, and then sending the results to 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 and working progressively downwards. This process repeats continuously, as shown in Figure 20.

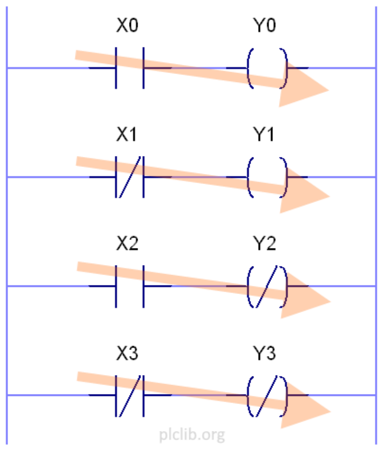


Figure 20. *The scan cycle includes the repeated process of reading inputs and updating outputs.*

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-style applications demonstrate simple parallel processing capabilities, but without the need to resort to advanced programming techniques.

For basic operation, the PLC library uses a single variable called *scanValue* to hold its running calculation result, as each branch is solved. Consider the C++ code snippet of Listing 21, to see how this works for single bit digital values.

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

Listing 21. *Reading a digital input and updating a digital output in a continuous loop (Source adapted from: IO > BareMinimum).*

The single bit input command `din(X0);` reads digital input pin *X0* and stores its result in the *scanValue* variable as 1 or 0. A subsequent bit output command `dout(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 `ain` command – as illustrated by the C++ code snippet of Listing 22.

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

Listing 22. *Reading an analogue input value from pin AD0 (A0) and sending to a PWM output (Source adapted from: IO > PWM).*

The same *scanValue* variable is used to hold this analogue value, which is not a problem as the *scanValue* variable is actually a 32-bit unsigned integer variable type. The plcLib software automatically handles the scaling of any pseudo analogue outputs, so the PWM output of the *pout* command in Listing 20 above, is scaled to be in the range 0-255 (8-bit binary). In this case, the PWM output may be used to control the apparent brightness of a connected LED, or even (via power amplification) the speed of a connected DC motor.

It is also possible to scale the current *scanValue* in any desired range. For example, a hobby servo may require an angular position in the range 0-179° (hence a span of 180°). This scaling may be achieved using the Arduino *map* command, as shown in the *Extras > Servo* example. (This example also illustrates how plcLib can interface with external libraries.)

Note: The PLC library can also perform *greater than* or *less than* comparisons based on analogue values, as was discussed earlier in the Analogue Comparison section. (See also the *Analogue* section of the pull-down menu for related examples.)

Finally, the scan cycle of the plcLib software, together with its internal operation, is somewhat simpler than that of a commercial PLC. It is useful to appreciate these differences and any potential implications on system design, performance and reliability. These characteristics are summarised in Table 35.

Feature	***plcLib***	***Typical PLC***
Number of Programme Organisation Units (POUs)	*One*	*Multiple*
Number of 'languages' directly supported	*One (C++ on target system)*	*Five (LD, IL, FBD, SFC, ST)*
Operating Speed	*Processing speed of CPU*	*Regular Event Scheduled*
Inputs	*Direct or buffered*	*Buffered*
Outputs	*Direct or buffered*	*Buffered*
3rd party libraries and directly connected additional devices	*Yes*	*No*

Table 35. *Comparing plcLib and PLC scan cycle characteristics.*

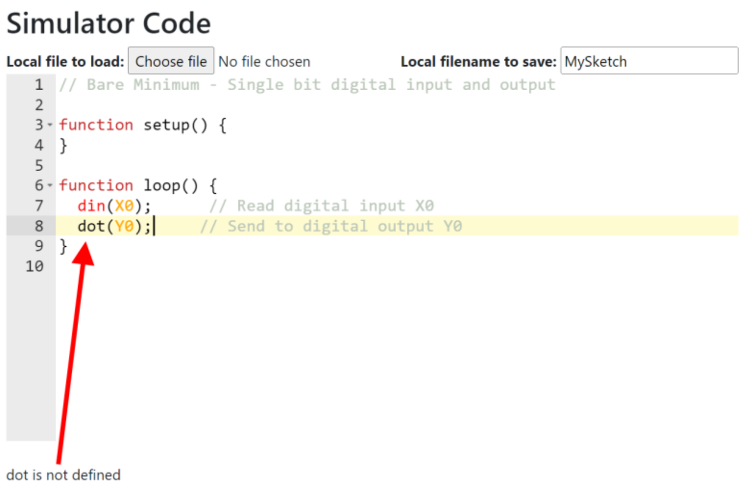
A fully featured commercial PLC might be programmable in several 'languages' (*ladder diagram*, *Instruction List*, *Function Block Diagram*, *Sequential Function Chart*, *Structured Text*) and these sub-programs may be attached to one or more 'events' which cause them to be executed on a regular basis, under the control of a scheduler. A combination of local and global variables allows proper execution of each program unit and communication between them. The plcLib library on the other hand operates as a single Arduino sketch, in a single programming language (C++) and this compiled program executes at the maximum speed permitted by the processor. Any local or global variables are also held within this single Arduino sketch.

Considering input/output arrangements, a commercial PLC will read all inputs into associated variables at the start of the scan cycle, and then perform any calculations on variable values held in the memory of the PLC. Output values are then updated at the end of the scan cycle. The plcLib library uses direct input and output, by default, and the decision whether to read and write values via buffer variables is at the discretion of the user. Which option is best may also depend on the application, which for plcLib is likely to be a 'product-related' question. In some situations, the fastest possible response may be required, which could come from a system which 'polls' inputs at maximum CPU speed, or even from an interrupt-driven system. On the other hand, an application such as a Set-Reset latch, implemented in plcLib using direct I/O, might give unwanted oscillation or ambiguous behaviour if the *Set* and *Reset* inputs are simultaneously applied, or a connected switch becomes stuck in the *on* position. (For more details, please see discussion of latches in the User Guide, and associated implementation options, including direct output via `setL` and `resetL` commands, indirect output via a buffer variable, or use of one-shot pulses.)

The plcLib library is intended to be capable of operating in conjunction with 3rd party libraries and directly or remotely connected devices, which is likely to be a requirement in a microcontroller-based product. For example, the connection of a servo or LCD display requires the installation of the related library, together with the inclusion of device-related commands. This is achieved through *escape sequences*, which cause 3rd party code to be ignored by the JavaScript editor and simulator, but executed normally by the embedded system itself. (Please see examples in the *Extras* pull-down menu, plus associated discussion in the User Guide and Reference Manual.)

Debugging Sketches

A variety of debugging options are available, both in the JavaScript-based editor and in the Arduino-IDE. The JavaScript editor has colour coded syntax highlighting which will flag any 'typos' at the earliest possible stage. The editor will also generate an error message at the bottom of the editor window, if a syntax error is detected, when *Run* is pressed.



Simulator Code

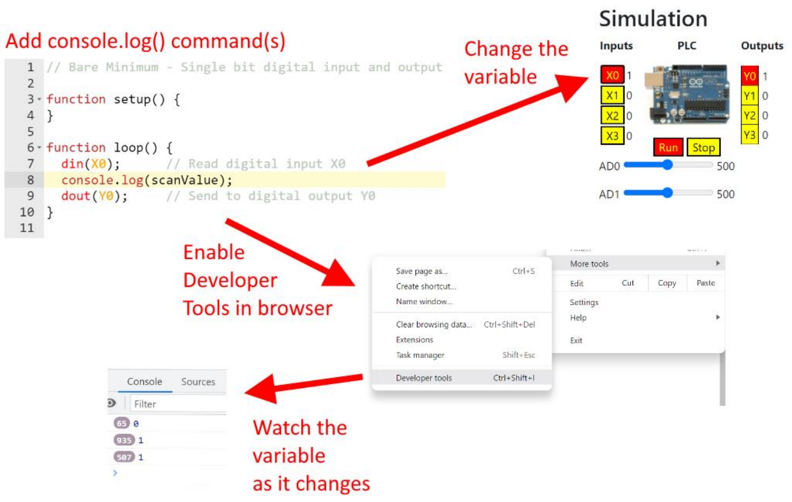
Local file to load: Choose file No file chosen   Local filename to save: MySketch

```
// Bare Minimum - Single bit digital input and output

function setup() {
}

function loop() {
  din(X0);       // Read digital input X0
  dot(Y0);      // Send to digital output Y0
}
```

dot is not defined

Figure 21. *Syntax-based highlighting and debugging in the JavaScript IDE.*

Having got past any obvious syntax errors, then next stage may be to identify logical errors in your code. A useful option is to insert one or more `console.log` commands into the sketch, given that the underlying programming language is JavaScript, as used in the *Simulator Code* window. This may be used to display the value of program variables via the *Developer Tools > Console* feature of your web browser (precise details of which will vary depending on the browser used). A simple example of the process is given in Figure 22.

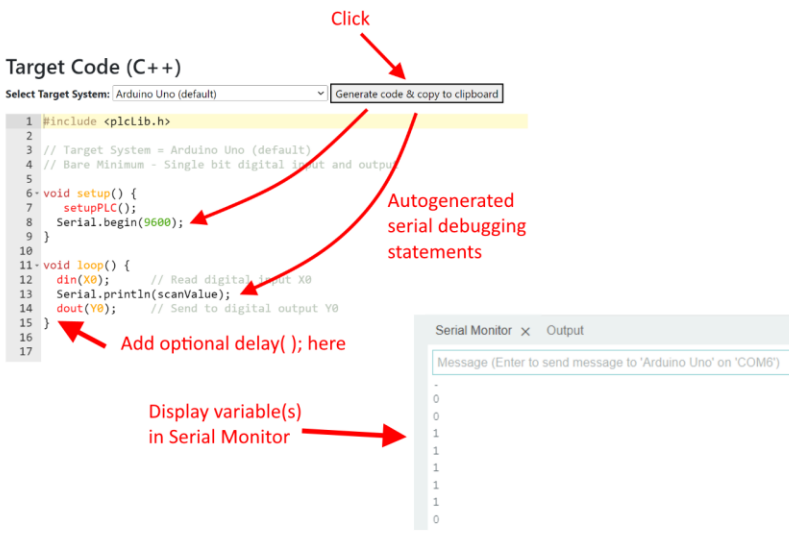


```
// Bare Minimum - Single bit digital input and output

function setup() {
}

function loop() {
  din(X0);       // Read digital input X0
  console.log(scanValue);
  dout(Y0);      // Send to digital output Y0
}
```

```
65 0
935 1
507 1
```

Figure 22. *Debugging a JavaScript-based sketch, which is running in the Simulator Code window.*

The system automatically generates the equivalent Arduino serial interface debugging commands, when the *Generate code and copy to clipboard* button is pressed, as shown in Figure 23.


Click
Target Code (C++)
Select Target System: Arduino Uno (default)
Generate code & copy to clipboard
Autogenerated serial debugging statements
Add optional delay( ); here
Display variable(s) in Serial Monitor
Serial Monitor
Output
Message (Enter to send message to 'Arduino Uno' on 'COM6')

```
#include <plcLib.h>

// Target System = Arduino Uno (default)
// Bare Minimum - Single bit digital input and output

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

void loop() {
  din(X0);       // Read digital input X0
  Serial.println(scanValue);
  dout(Y0);      // Send to digital output Y0
}
```

```
-
0
0
1
1
1
1
1
0
```

Figure 23. *Using Serial debugging features in the Arduino IDE.*

Notice from Figure 22, that the serial interface is automatically initialised, and a default baud rate set. Any `console.log` commands from the JavaScript version are automatically converted to their equivalent `Serial.println` command equivalents. The value of any monitored variables may then be displayed in the *Serial Monitor* window. An optional delay may be added to the Arduino sketch, should it be necessary to temporarily slow the speed of the system, for debugging purposes.

Additional debugging options may be available in the Arduino IDE itself, from Version 2.0 onwards, but this is beyond the scope of this manual.

Loading and Saving User Sketches

It is 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 24.

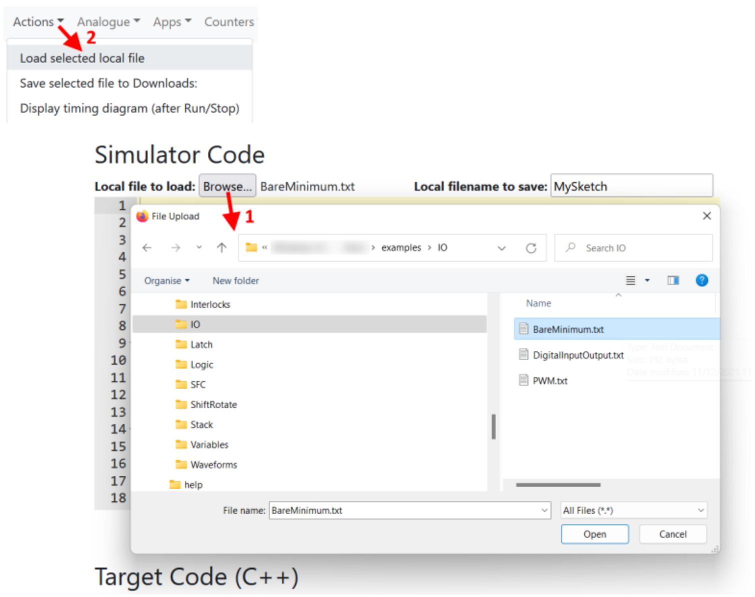
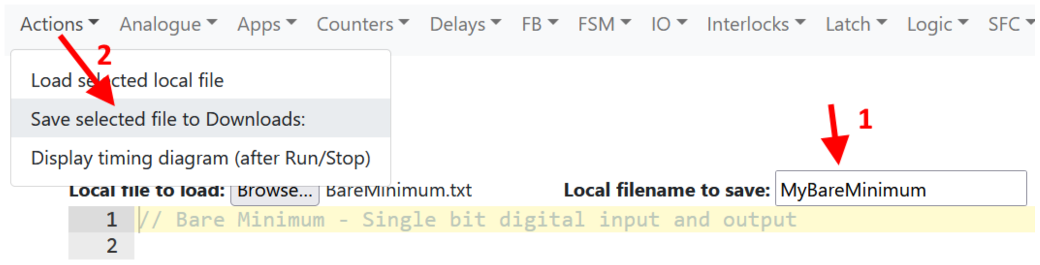


Figure 24. *Selecting and loading a local file.*

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 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 25.



```
// Bare Minimum - Single bit digital input and output
```

Figure 25. *Saving to the Downloads folder as a local file.*

Extending the System

A number of options are available, should you decide to extend the system, as described in the following sections.

Adding C++ Specific Code in the JavaScript IDE

The *//$* escape sequence may be used to include code which is intended for the C++ environment, but which should be ignored by the JavaScript simulator. A typical scenario would be the inclusion of code associated with additional hardware, such as a servo or LCD display. As an example, Listing 23 shows the linkage of an analogue input and a servo output, with all servo-related code being temporarily hidden by escape sequences.

```
// 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 23. *Using '//$' escape sequences to exclude servo-specific code from the JavaScript simulator.*

Related Examples:

A range of examples is available in the *Extras* folder, demonstrating the connection of additional hardware, or use of target-specific features not directly supported by the simulator.

- *Extras* > Servo
- *Extras* > CountUpDownLCD
- *Extras* > OvenControlMooreLCD

Adding New Features

Users are free to add new features, for their own use, or to share with the wider community. Options include: -

- adding new functionality via custom C++ functions,
- testing and integrating 3rd party libraries,
- creating new function blocks,
- adding support for new hardware.

These are listed above in approximate order of difficulty, from simple to complex.

A simple C++ function, for example, could consist of a few lines of code, plus a demonstration sketch and brief commentary on what it does and how it is used. 3rd party libraries already exist, so the first task is to thoroughly test the library with plcLib, identify potential use cases, then develop some test sketches and documentation. In either case, it is important to identify any limitations of the new system or feature. For example, if you decided to interface an ultrasonic distance sensor to a plcLib-based application, you would probably start by using the built-in Arduino *`pulseIn`* function to measure the elapsed time of the return pulse. However, *`pulseIn`* is a 'blocking' function (just like *`delay`*), so it will slow down the associated scan loop! The *`pulseIn`* function has an optional timeout parameter, so there is a trade-off between the choice of timeout, the maximum distance measured (based on the speed of sound) and the slowest allowed speed of execution of the scan loop. A question is therefore whether the resulting scan loop speed would still be fast enough for the application? For a maze solving robot, the answer might be 'yes', but for a more time-critical application, the answer might be 'no'. If performance is an issue, due to the blocking behaviour of *`pulseIn`*, an alternative approach might be to develop a non-blocking equivalent. This may be more challenging in the short term, but would be likely to give better performance in the long run.

A new plcLib *function block*, is conceptually like a subroutine, which accepts a number of parameters and returns a result to the associated object/variable. A good first step is to study existing function blocks and understand how they work. One of the challenges in C++ is to produce code which will work with the range of parameter types that plcLib supports. For example, an argument may be supplied to a function block as either an integer number (0, 1, 1023 etc.), indirectly via a pin number (from where a value should be read), or may even be obtained from another object. C++ can achieve this flexibility through *function overloading* (where multiple versions of a function are written, to cope with each eventuality), or via *templates* (where the parameter type is identified at a later stage by the compiler). The latter is the approach used in plcLib (mostly), however this is an advanced topic, explanation of which is beyond the scope of the current document. Once a C++ function block has been written, by whatever means, the process of adding it to the plcLib system (if this has been agreed), consists of several further steps. Firstly, a functionally equivalent JavaScript function block must be added to the JavaScript library and tested. Secondly, syntax/keyword highlighting options need to be added to both the webpage editor and the Arduino IDE, so the new function block command(s) will be correctly highlighted. Next, one or more example sketches should be added, to demonstrate the new feature, and finally the user documentation must be updated.

When adding support for a new hardware platform, a key consideration is whether it can support the basic I/O model used by plcLib (4 digital inputs *X0-X3*, 2 analogue inputs *AD0-AD1*, 4 digital outputs *Y0-Y3*). In addition, how will users connect inputs and outputs, and what types of I/O are supported? Does it support *Grove* devices for example, or an equivalent hardware prototyping technology? Can outputs generate both digital signals and also PWM? If the answer to any of these questions is 'no' or 'only partially', this may mean that some of the built-in examples will not work. Assuming that a hardware platform passes these initial tests, then adding it to the main system (if agreed) will consist of several further steps, such as adding a thumbnail image to the simulator, plus an associated custom I/O allocation for the *Target Code (C++)* editor window.

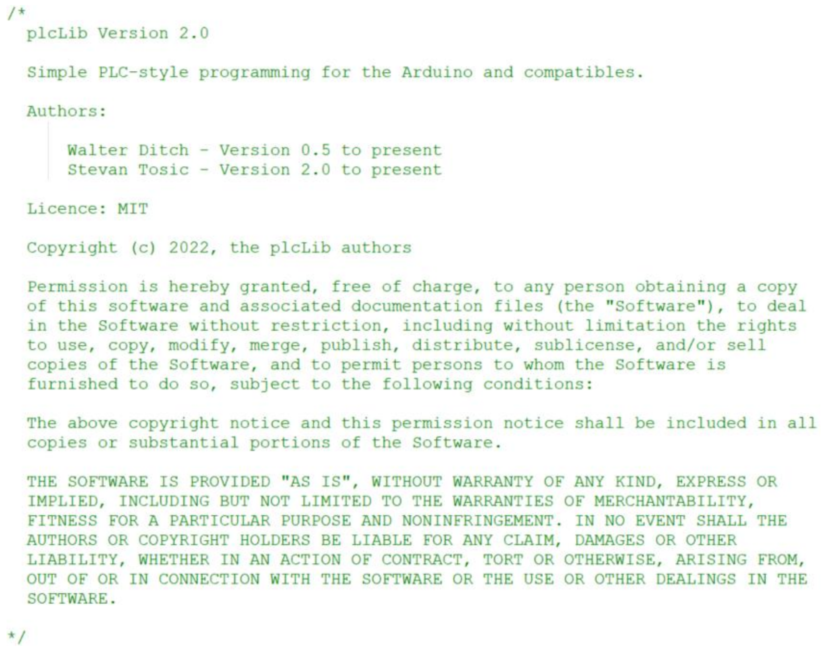
Whatever option you choose, the importance of thorough testing, development of test/demonstration sketches and production of documentation cannot be overemphasised. This is a marathon rather than a sprint! Please do get in touch if you develop a new feature which you would be willing to share.

Licensing and Disclaimer

The website, Web IDE, simulator and associated Arduino library files are provided for educational and academic research purposes only.

The plcLib library and related source code files are released under a permissive MIT licence, but do not come with any kind of warranty, and are used entirely at your own risk. Please open library source code files in a text editor to view associated licensing information.

A screenshot of the C++ licence statement is reproduced below, for information.



```
/*
  plcLib Version 2.0

  Simple PLC-style programming for the Arduino and compatibles.

  Authors:

      Walter Ditch - Version 0.5 to present
      Stevan Tosic - Version 2.0 to present

  Licence: MIT

  Copyright (c) 2022, the plcLib authors

  Permission is hereby granted, free of charge, to any person obtaining a copy
  of this software and associated documentation files (the "Software"), to deal
  in the Software without restriction, including without limitation the rights
  to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
  copies of the Software, and to permit persons to whom the Software is
  furnished to do so, subject to the following conditions:

  The above copyright notice and this permission notice shall be included in all
  copies or substantial portions of the Software.

  THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
  IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
  FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
  AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
  LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
  OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
  SOFTWARE.

*/
```

Figure 26. C++ plcLib software is licensed under a permissive MIT licence.

Earlier versions of the C++ library (0.5 – 1.4) were released under the GNU GPL, but this changes to MIT for Version 2.0 and beyond. The JavaScript version of the plcLib library is also released under the same MIT licence, but Version 2.0 is the first public release.

Acknowledgements

This Web-based editor, simulator and associated library makes use of a number of advanced technologies which have been generously shared by their creators, in line with their respective licensing agreements. This includes: -

- ACE Editor, which is released under the BSD licence.
- Bootstrap, which is released under the MIT licence.
- Plotly.js, which is released under the MIT licence.