I/O Plus (Logic) - Instruction and Examples

Overview

This document explains the feature set of the “I/O Plus” logic which is available in the Wireless I/O products. I/O Plus is made available to perform logic tasks to users in the available wireless products and is not intended on being utilised as a PLC or RTU replacement.

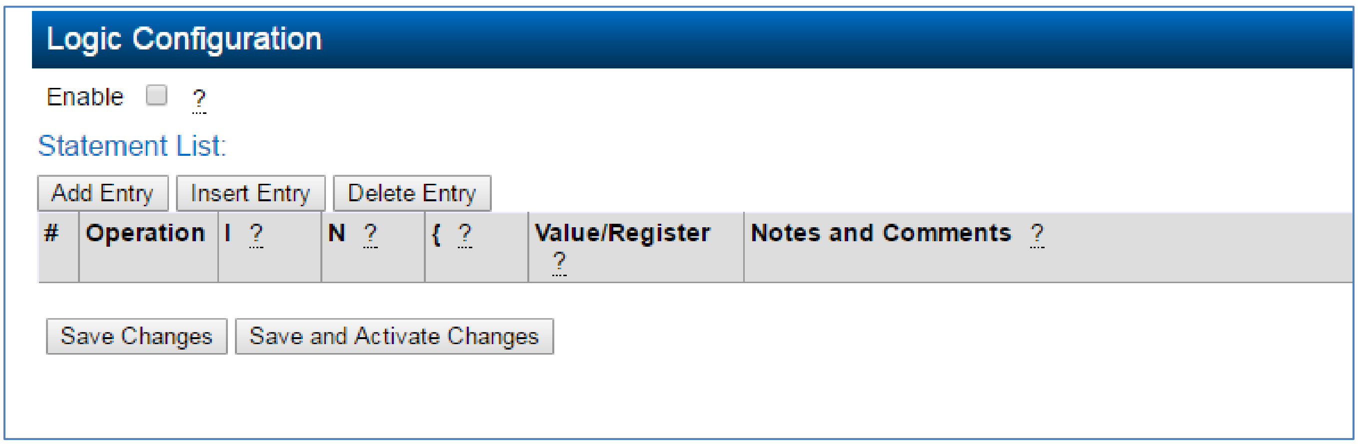
Configuration for the logic is performed via the module’s internal webpages.

The cycle time for the IO plus is fixed at 250msec with a maximum of 300 instructions per device

Logic Page

Select “IOPlusLogic” from the right hand side menu to see the Logic Configuration Screen.

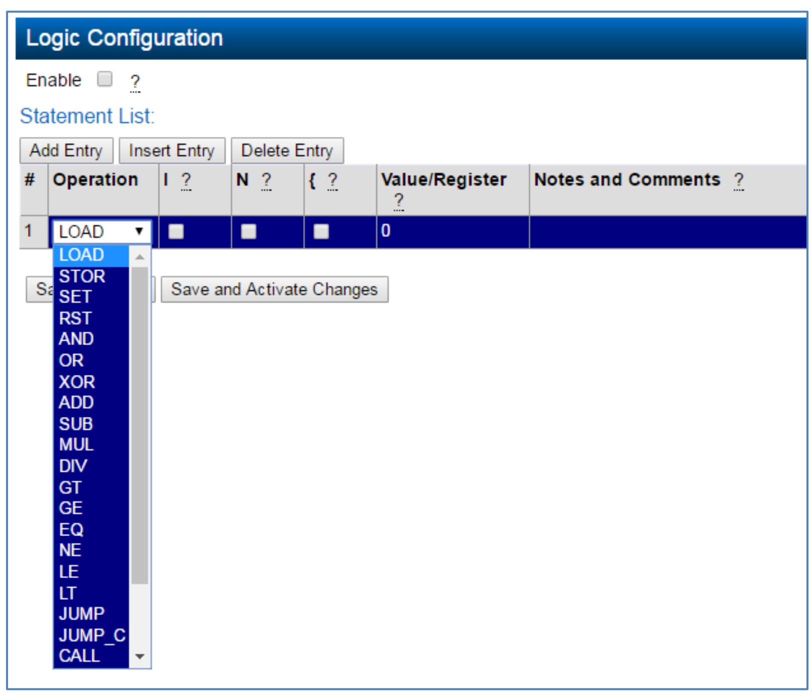
You should then see a screen like below



First you must enable the logic function by ticking the box “Enabled”

Then add/remove operational entries by clicking the Add, Insert, Delete buttons.

The configuration is made up of a list of statements and each statement has a number of configurable column entries, i.e. Operation, Immediate Value, Negate Operation, “{“ Bracketed Block, Value/Register Values and Notes/Comments.



Configuration

The device executes a software process that reads and performs the actions programmed in the statement list. The statements in the list are executed by this process in the defined order until the end of the list is reached. The logic process then waits until it is time to begin the next execution cycle, and again executes the statements in the list. This execution cycle is repeated again and again while the device is operating.

The statement list can include branch instructions which cause the control flow to follow a different path, so every statement in the list will not necessarily be executed on each execution cycle. It is also possible to develop looping constructs within the statement list, so a group of instructions could be executed multiple times during one execution cycle. Care must be taken to ensure that any loops will terminate in time so that the execution of the Statement list will not exceed the maximum allowed cycle time.

The Logic engine aims to execute the full statement list once every 0.25 Sec. This is the cycle time. Each execution of the statement list has a deadline that is 1.25 seconds after the target completion time. If the execution cycle does not complete before the deadline, the execution of the cycle is aborted. When this happens, the Diagnostic register is set to the value 32768. This means that you can rely on timers being no more than 1.25 second late as long as the Diagnostic register doesn't indicate overrun. The Logic engine is designed to be capable of executing up to 1000 instructions without exceeding the deadline.

Diagnostic Register 30491

Value	Meaning
0	Logic program not running (Logic execution not Enabled or "Default" switch is ON)
256	Logic program has started execution and is executing.
32768	Logic program has failed to complete executing the full statement list within the deadline and has aborted that execution cycle of the statement list. (at least once)

You can view the content of this register from the page at "Unit Diagnostics >> I/O Diagnostics".

Configuration

The process uses a statement list to perform the various calculations and processes. Each instruction will perform the configured operation and then the result will be saved back to the accumulator. E.g. if we "Load" a register into the accumulator and perform an operation i.e. "GT" Greater-Than a Value/Register, the accumulator will then become the result of this instruction, i.e. it will hold a "1" if the operation was True or "0" if it was False.

The configuration parameters are explained below

Operations

There are a number of configurable operations and each one will perform a specific task, whether it be loading a value, storing a value, applying some sort of logical or mathematical operation or applying some other operational instruction, i.e. Jumping , setting or calling.

A full list of all of the operations and brief explanation of how it works will be at the end of the document.

"I" (Immediate)

When selected the instruction will use either the value or the register location that is entered into the "Value/Register" column.

"N" (Negate)

When selected this allows the operation to be negated (opposite). i.e. Selecting “GT” (Greater Than) and also selecting the “N” will mean the operation will become “Not Greater Than”

“{“(Starts a new Block)

Allow you start a new function block. You can have Sub blocks nested within the statement list.

Value / Register

The value or register location that will be used by the operation.

Notes and comments

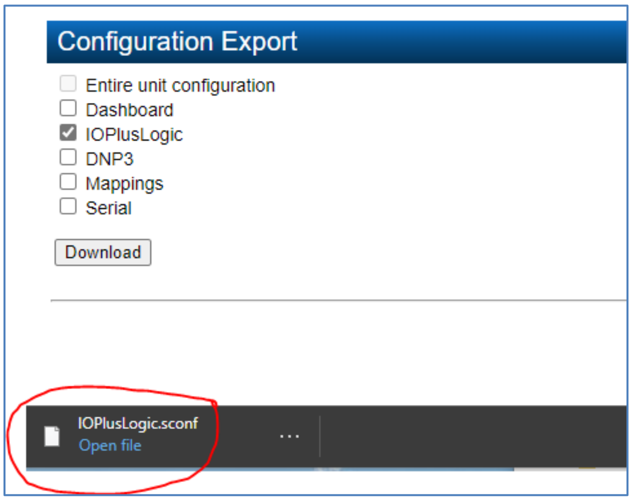
Notes or comments that help to explain the logic operation and configuration.

Note:

Configurations can be saved once they have been entered into the Web page table.

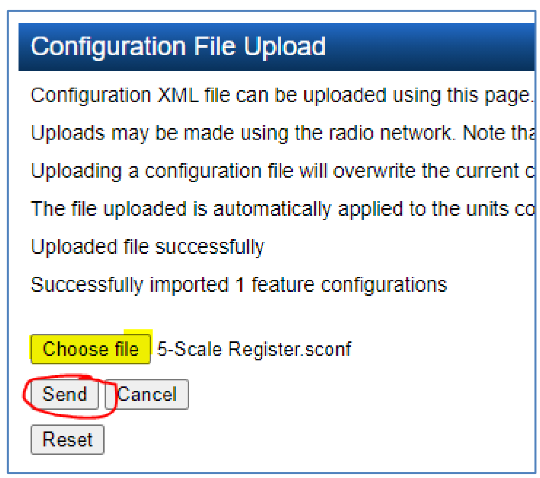
To save the logic config select “System Tools” web page link and then select “Read Configuration”

Next select “IOPlus Logic” press the “Download” button and then save the “IOPlusLogic.sconf” file.



To load an xxxx.sconf file into the module go to System Tools/Write Configuration File then “Choose file” and locate the sconf file you want to upload. Press the “Send” button then navigate to the IOPlusLogic Web link and you will see the Logic has been loaded.

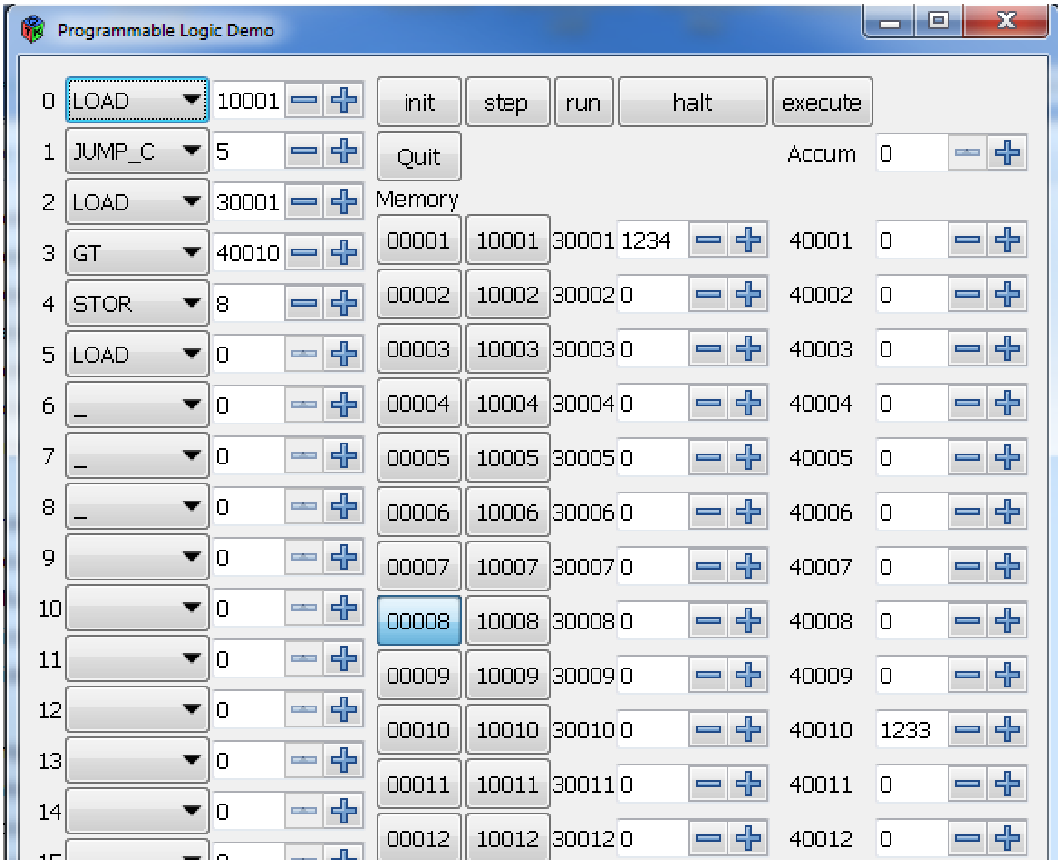
Press the “Save and Activate Changes” button to activate it.



Testing

Testing can be done by using the “DemoLogic” application. This will allow you to enter the Logic operations as they would be entered into the module however it allows you to step through or run the logic and tests the application.

Contact your Technical Support for the Demo Logic Application



Note: You can use this application to test logic command sequences and see how the logic engine operates. This application simulates a limited version of the logic engine provided in the product. Future application will provide full device simulation and the ability to save the test logic steps to a file which you will then be able to upload to the module.

There are many different ways of configuring the statement list, below are some examples that have previously been used and may help explain the different operations, how they function and how they can be implemented.

It is advised to test the IOPlus Logic prior to it being implemented using the “DemoLogic” application to ensure correct operation and outcome.

Examples

#1: Run pump to fill a tank.

This example uses an analog input to measure tank level (Analog 1 at register 30001). It fills the tank if it is below a remotely configured set-point (register 40501), and stops when the level reads 1000 counts over the set-point.

Digital inputs 2 and 3 provide a pump stop signal (10002), and a manual pump run signal (10003) which override the normal operation.

The pump is controlled by a contactor connected to Digital output 1 (0001).

#	Operation	I ?	N ?	{ ?	Value/Register ?	Notes and Comments ?
1	LOAD	☐	☐	☐	10002	Get Pump STOP Signal.
2	RST	☐	☐	☐	1	If Pump STOP, then diable (reset) Pump Control outpu
3	JUMP_C	☐	☐	☐	1000	If Pump STOP, then finished. Exit this execution.
4	LOAD	☐	☐	☐	10003	Get Manual RUN signal
5	SET	☐	☐	☐	1	If Manual RUN, then enable (set) Pump Control output
6	JUMP_C	☐	☐	☐	1000	If Manual RUN, then finished. Exit this execution.
7	LOAD	☐	☐	☐	30001	Load in the current Tank Level
8	LT	☐	☐	☐	40501	Check if below target level in register 40501
9	SET	☐	☐	☐	1	If below target level, then enable (set) Pump Control
10	LOAD	☐	☐	☐	30001	Load tank level and Check if above target level +1000
11	GT	☐	☐	☑	40501	{ Start sub-calculation for GT, by loading target level
12	ADD	☑	☐	☐	1000	Add 1000 (Immediate) to target level
13	}	☐	☐	☐	0	Close sub-calculation - level GT {target + 1000}
14	RST	☐	☐	☐	1	If above {target + 1000}, disable (reset) pump control

- Lines 1-3 Check the STOP signal and stop the pump if it is active (RST on line 2). If the stop signal is active, then line 3 exits from the statement list execution by jumping beyond the end of the program (Line 1000).
- Lines 4-6 check the Manual RUN signal and start the pump if it is active (SET on line 5). If the run signal is active, then line 6 exits from the statement list in the same way as line 3.
- Lines 7-9 check if the tank level (30001) is less than (LT) the target level (40501), and if so then the pump is started (SET on line 9).
- Lines 10-14 check if the tank level is more than 1000 counts above the target level (40501). The addition of register 40501 with the immediate value 1000 is performed as a sub-calculation of the comparison (GT on line 11 through to “}” on line 13). Line 14 turns off the pump (RST) if the comparison (GT) is true.

#2: “Truflow” pump controller masking logic:

Truflow Pump Controller needed to mask certain register values based on other values. i.e. If register “x” (pump status value) is Less than “a value” then make register “y” = register “x”; otherwise (when register “x” Greater than “a value”) make register “y” = 0

Statement List:

Add Entry | Insert Entry | Delete Entry

#	Operation	I ?	N ?	{ ?	Value/Register ?	Notes and Comments ?
1	LOAD	☐	☐	☐	40501	Load Input Value 1
2	LT	☑	☐	☐	8000	If Input value is less than X (8000)
3	JUMP_C	☐	☑	☐	7	If Input is not less than X (8000) Jump to line 7
4	LOAD	☐	☐	☐	40501	Load Input value 1 again if value was less than X
5	STOR	☐	☐	☐	40502	Store Input value 1 into Register 40502
6	JUMP	☐	☐	☐	10	End Jump out to line 10
7	LOAD	☑	☐	☐	0	If Input Value was not less than X (Line 2) load 0 into Accumulator
8	STOR	☐	☐	☐	40502	Store Accumaltor value 0 into 40502
9	LOAD	☐	☐	☐	0	

- Loads the input value from the register (40501) into the accumulator.
- It then compares it to the value of 8000 (Because “I” is enabled this means it will use the value instead of reading the register). The result of this operation will then either be“1” if register 40501 is less than 8000 or “0” if greater than 8000
- If the result is “0” it jump to line 7, loads 0 then stores this to Register 40502
- Else if the outcome is “1” it will load register 40501 and then write this to register 40502. (Lines 4&5).

#3: Pump Run Time Accumulator, i.e. Counts the time an input has been activated.

This statement is reading the status of an input, i.e. Pump Run and then starting a timer and accumulating a run time in seconds. Used for measuring the total run time of a pump or motor for preventative maintenance purposes.

Add Entry | Insert Entry | Delete Entry

#	Operation	I ?	N ?	{ ?	Value/Register ?	Notes and Comments ?
1	LOAD	☐	☐	☐	10001	Load value of Input 1
2	JUMP_C	☐	☑	☐	6	If value is 0 jump to routine line 6
3	LOAD	☐	☐	☐	40501	Load timer counter register (General Purpose reg)
4	ADD	☑	☐	☐	1	Add 1 to the Accumulator
5	STOR	☐	☐	☐	40501	Save back into Timer Counter Register (40501) this is quarter seconds
6	GE	☑	☐	☐	4	If value of 40501 is greater than or equal to 4 (1 sec)
7	JUMP_C	☐	☑	☐	13	If value is not GE to above line jump to the end of the routine (line 13)
8	LOAD	☑	☐	☐	0	Set accumulator back to 0
9	STOR	☐	☐	☐	40501	Save back into Timer Counter Register
10	LOAD	☐	☐	☐	40502	Load Second register (40502)
11	ADD	☑	☐	☐	1	Add 1
12	STOR	☐	☐	☐	40502	Re-save the value in seconds
13	LOAD	☐	☐	☐	0	End

- First, we load the pump run input (register 10001).
- If the input is ON it starts a ¼ second counter routine (line 3) which loads register 40501, adding "1" to it and then writes this back to register 40501.
- It then checks if the ¼ second timer has counted 1 second i.e. if the ¼ second timer routine has counted 4 times (register 40501 is 4).
- If one second has passed it will reset this register so as to start the counter from zero again.
- Then load register 40502 (Timer Counter Value), increments it by 1.
- Then saves this back to register 40502. (This is the register that will store the number of seconds the pump has run).
- If one second has not passed (1/4 sec timer routine) it ends the statement, so it can scan again (jumps to line 13).
- If the Pump Run input is OFF (from the start) it basically jumps over the rest of the routine and re-scans

#4: Pump Number of Starts, i.e. Counts the number of times an input has been activated.

This statement is counting the number of times an input has been activated, e.g. measuring the number of times a pump has started for maintenance purposes.

Add Entry | Insert Entry | Delete Entry

#	Operation	I ?	N ?	{ ?	Value/Register ?	Notes and Comments ?
1	LOAD	☐	☐	☐	0	This section Counts Pump Starts in Reg 40503. Uses 501 to save the pump status
2	LOAD	☐	☐	☐	10001	Get the input value
3	EQ	☐	☐	☐	501	Has it changed since last time?
4	JUMP_C	☐	☐	☐	300	No change, then we're done
5	LOAD	☐	☐	☐	10001	Load input again
6	JUMP_C	☐	☐	☐	8	If the input ON then we need to count it at line 8
7	JUMP	☐	☐	☐	12	Input was off - dont count, just need to save the input for next time.
8	LOAD	☐	☐	☐	40503	Here we count the start - Load the counter register
9	ADD	☑	☐	☐	1	Add 1
10	STOR	☐	☐	☐	40503	and save back to counter register.
11	LOAD	☑	☐	☐	1	The input was on - Setup to save the lst value (Could alos LOAD 10001)
12	STOR	☐	☐	☐	501	Save the value of 10001 so we can check if it changed on the next scan.

- First, we load 0 to clear all previous values.
- Load Digital Input #1- Pump run input (register 10001).
- Next, we check to see if the value has changed since last time by checking if it is equal (EQ) to value in reg 501.
- Register 501 has the last saved status of the input (Saved from the last operation in this statement list).
- If it has not changed then jump to the end of the Statement list (line 300)
- If it has changed then check if it is ON and if so jump to Line 8.
- Line 8 will read register 40503 (Pump Start Counter), increments it by one then saves it back to 40503.
- 40503 is the register that will hold the Number of Starts Counter.
- Lastly, we load the value of 1 and store this to register 501 to be used in the next scan.

#5: Scale Register, i.e. Scaling an internal register from a 4-20mA value.

This statement is scaling an internal Register (30510) to a value within the ranges, 0-5000 or 0-10000 based on a normal 4-20mA analog range (16384 - 49152) in another internal register (30501).

#	Operation	I ?	N ?	{ ?	Value/Register ?	Notes and Comments ?
1	NO_OP	☐	☐	☐	0	Scale Reg for 0-5000 or 0-10000 from 4-20mA value in Reg 30501 & Store in 30510
2	LOAD	☐	☐	☐	30501	Read Modbus Register
3	GE	☑	☐	☐	16378	If Greater than 16384 (4mA) with rounding offset
4	JUMP_C	☐	☑	☐	8	Jump to Step 8, Value will be zero if less than 16378
5	LOAD	☐	☐	☐	30501	Reload Modbus Register again
6	SUB	☑	☐	☐	16378	Subtract 16384 (4mA) with rounding offset
7	DIV	☑	☐	☐	13	Divide by 13 (gives 2520 max).
8	MUL	☑	☐	☐	2	Multiply by 2 (0-5000 scale), X4 for 10000 Scale.
9	STOR	☐	☐	☐	30510	Store value in 30510

- NO_OP is just there to allow a comment to be added.
- Loads internal analog reg 30501.
- Checks if it is greater than 16378 (4mA) with a slight rounding offset.
- Jump to step 8 if Not True (less than 4mA) (value will be zero if less than 16378).
- If True it loads register 30501 again
- Subtract 16384 from the Value in 30501
- Divide this by 13 (32768/2520)
- Multiply by 2 for a scale 0-5000 or 4 for 0-10000.
- Store scaled value (or zero if less than 16378 from step 4 jump) into register 30510.

#6 Accumulate a flow rate for Total Flow

This example accumulates measured flow at analog input 1, to calculate a totalized flow. The analog input reads 0-100 l/min for 4-20mA (register value 16384 to 49152). The totalized flow is calculated in units of litres, and is saved in the 32-bit register 36021/36022.

The analog value is sampled once on each logic execution (four times per second). This results in a scaling factor of 240 to scale to minutes (4 samples/sec X 60 secs/min), and a scaling factor of 1/100 to scale to litres (full scale is 100 l/min) . Because the full-scale register value is 32768 (49152-16384), this scaling factor of 2.4 (240 / 100) scales the initial accumulated value to units of 1/32768 litre.

#	Operation	I ?	N ?	{ ?	Value/Register ?	Notes and Comments ?
1	LOAD	☐	☐	☐	30001	Get the flow rate value from Analog 1
2	STOR	☐	☐	☐	40502	Save to temporary location for later (Line 6)
3	GT	☑	☐	☐	16384	Check not less than zero (16384 is zero scale)
4	JUMP_C	☐	☑	☐	11	If not less than zero, then skip next instruction.
5	LOAD	☑	☐	☐	16384	Set value to zero scale.
6	LOAD	☐	☐	☐	40502	Re-load flow value saved iin Line 2
7	SUB	☑	☐	☐	16384	Subtract zero scale value to get 0-32768 = 0-100l/min.
8	ADD	☑	☐	☐	6	Now divide by scale (compensate for rounding first)
9	DIV	☑	☐	☐	12	And divide. Scale is 2.4 = 4 samples/sec x 60 sec/100l
10	MUL	☑	☐	☐	5	Divide by 12, multiply by 5 = divide by 2.4.
11	ADD	☐	☐	☐	40501	Add to accumulator in register 40501
12	STOR	☐	☐	☐	40501	And save back for next time.
13	LT	☑	☐	☐	13768	Check if 1 liter accumulated yet.
14	RET_C	☐	☐	☐	0	Not reached 1 liter yet. All done.
15	LOAD	☐	☐	☐	40501	Subtract 1 liter from accumulator.
16	SUB	☑	☐	☐	32768	32768 is 1 liter.
17	STOR	☐	☐	☐	40501	And save back.
18	LOAD	☐	☐	☐	36021	Now increment the accumulator for total flow (liters)
19	ADD	☑	☐	☐	1	Increment low word.
20	STOR	☐	☐	☐	36021	And store back.
21	RET_C	☐	☐	☐	0	Check for overflow. If not zero, then all done.
22	LOAD	☐	☐	☐	36022	Low word overflowed, so increment high word.
23	ADD	☑	☐	☐	1	Increment
24	STOR	☐	☐	☐	36022	And store back.

- Lines 1-7 Load the analog value and shift the scale to zero offset by subtracting 16384 (Analogs are 16384 for 4mA, 49152 for 20mA). Lines 3-7 check the value is above the zero point (4mA) and set it to this value if it isn't. The analog value is saved at line 2 and restored at line 6 to ensure that the value doesn't change after the check for less than 4mA at line 3. At this point, the accumulator holds the flow rate scaled for 0=0 l/min, 32768=100 l/min
- Lines 8-10 Multiply by the scale factor 2.4. This is done by dividing by 12, then multiplying by 5. Note line 9 pre-compensates the result by adding ½ of the divide value that is used in line 10. This compensates for rounding that occurs during the integer divide. At this point, the accumulator holds the total litres measured in this sample scaled so that 32768 corresponds to 1 litre.
- Lines 11-12 Add the scaled value to the accumulator value in register 40501. This register holds accumulated flow in units (litres/37268).
- Lines 13-14 check if one litre (32768 counts) has been accumulated. If not, then the execution completes (RET_C).
- Line 15-17 subtract 1 litre (32768) from accumulator register 40501, in preparation to adding one litre to the accumulated value in 32-bit register 36021.
- Lines 18-20 add 1 to the low word of the 32-bit register, 36021 (corresponding to 1 litre)
- Line 21 checks if the register overflowed as a result of the addition (if the accumulator value is zero after incrementing it). If there is no overflow, then the program exits (RET_C).
- Lines 22-24 increment the high word of the 32-bit register (36022) if the low register rolled over to zero.

#7 Use a Timer Function to de-bounce two digital inputs

This example shows you how to implement a general-purpose timer function. It does this by implementing two timers to separately de-bounce two digital inputs. Digital input 1 is de-bounced for 2.5 seconds, and output to Digital output 3. Digital input 2 is de-bounced for 5 seconds, and output to digital output 4.
Two registers (40501 and 40502) are used as the de-bounce timers for the two inputs. The timer function takes the memory address of the timer as an argument, so the same function can operate on either timer. The timer function uses register address 46000 as a temporary location to save the timer address to use for the delayed calculation in the following LOAD and STOR instructions.

#	Operation	I ?	N ?	{ ?	Value/Register ?	Notes and Comments ?
1	JUMP	☐	☐	☐	14	Jump over the Timer function to first line of DIN1 debounce
2	NO_OP	☐	☐	☐	0	## Timer Function Timer is in the Accumulator ##
3	STOR	☐	☐	☐	46000	Save Timer to temporary address for Later.
4	LOAD	☐	☐	☑	0	Load the Timer value from memory
5	LOAD	☐	☐	☐	46000	- Use saved address in register 46000
6	}	☐	☐	☐	0	- to load the timer value.
7	RET_C	☐	☑	☐	0	If the timer has already timed out. Return zero.
8	SUB	☑	☐	☐	1	Else decrement the timer by 1.
9	STOR	☐	☐	☑	0	Add store back to its loaction.
10	LOAD	☐	☐	☐	46000	- Use saved address in register 46000
11	}	☐	☐	☐	0	- Store the updated timer value.
12	RET	☐	☐	☐	0	Return (Zero if timed out. Else return the timer value)
13	NO_OP	☐	☐	☐	0	## End timer function ##
14	LOAD	☐	☐	☐	10001	## DIN1 Debounce ## Digital input 1
15	XOR	☐	☐	☐	00003	Compare to Digital ouput 3
16	JUMP_C	☑	☐	☐	4	If different. Then jump to line 20 (this Line +4) to check the timer.
17	LOAD	☑	☐	☐	10	If they are the same. Then restart the timer (2.5=10x1/4sec)
18	STOR	☐	☐	☐	40501	Use register 40501 for DIN1 debounce timer
19	JUMP	☑	☐	☐	6	Jump to end of debounce (this line +6)
20	LOAD	☑	☐	☐	40501	Load taddress of the timer register.
21	CALL	☐	☐	☐	2	Call the timer function at line 2
22	JUMP_C	☑	☐	☐	3	If timer not expired. Jump to end of debounce (this line +3)
23	LOAD	☐	☐	☐	10001	Load DIN1
24	STOR	☐	☐	☐	3	Save to DOT3
25	NO_OP	☐	☐	☐	0	## End DIN1 Debounce ##
26	LOAD	☐	☐	☐	10002	## DIN2 Debounce ## Digital input 2
27	XOR	☐	☐	☐	00004	Compare to Digital ouput 4
28	JUMP_C	☑	☐	☐	4	If different. Then jump to line 32 (this Line +4) to check the timer.
29	LOAD	☑	☐	☐	20	If they are the same. Then restart the timer (5=20x1/4sec)
30	STOR	☐	☐	☐	40502	Use register 40502 for DIN2 debounce timer
31	JUMP	☑	☐	☐	6	Jump to end of debounce (this line +6)
32	LOAD	☑	☐	☐	40502	Load taddress of the timer register.
33	CALL	☐	☐	☐	2	Call the timer function at line 2
34	JUMP_C	☑	☐	☐	3	If timer not expired. Jump to end of debounce (this line +3)
35	LOAD	☐	☐	☐	10002	Load DIN2
36	STOR	☐	☐	☐	00004	Save to DOT4
37	NO_OP	☐	☐	☐	0	## End DIN2 Debounce ##
38	NO_OP	☐	☐	☐	0	## End Program ##

- The timer function is defined in lines 2-13. The timer function implements a count-down timer, and returns zero if the timer has expired, otherwise it returns the current timer value. The function is called with the address of the timer register in the accumulator.
    - Line 3 saves this address to temporary location 46000.
    - Lines 4 -6 load the timer value, using the address stored in location 46000.
    - Line 7 checks if the timer value is already zero, and if it is then returns with the value zero still in the accumulator. This indicates the timer has expired.
    - Lines 8-11 subtract 1 from the timer value in the accumulator (Line 8) then save the new value back to the timer register using the address stored in location 46000 (lines 9-11).
    - At Line 12, the timer value is still in the accumulator. This is returned. It will be zero if the timer expired on this call. Otherwise, it will be non-zero and will have the current timer value.
- Digital input 1 is de-bounced on lines 14 – 25. The de-bouncing uses register 40501 as a timer register.
- Lines 14 and 15 load the DIN1 value and compare it against the DOT3 value that it is de-bounced to. (XOR will be 1 if the values are different, and 0 if they are the same).
- Line 16 checks the comparison, and if DIN1 and DOT3 are different the program jumps ahead to line 20 (using the offset jump instruction to jump ahead 4 lines).
- If DIN1 and DOT3 are the same, then the code in lines 17 – 19 initialise the de-bounce timer (register 40501) to its required value. The de-bounce for DIN1 is 2.5 seconds, corresponding to 10 counts at a rate of ¼ second per count. Line 19 then jumps ahead 6 lines to the end of the DIN1 de-bounce code at line 25.
- Lines 20 and 21 are executed when DIN1 and DOT3 are different. These lines load the address of the DIN1 timer (40501) and call the timer function. The timer function returns 0 if the timer has expired.
- Line 22 checks if the timer has not yet expired, and if not then jumps ahead 3 lines to the end of the DIN1 de-bounce code.
- Lines 23 -24 execute when the de-bounce is complete. This happens when the timer has expired (Accumulator is zero after CALL ing the timer function). These two lines copy the value in DIN1 to DOT3.
- Digital input 2 is de-bounced on lines 26 – 37. The de-bouncing uses register 40502 as a timer register. The code is very similar to the code for DIN1 de-bounce. Because the JUMP instructions are offset jumps (“I” flag), the JUMP and JUMP_C instructions at lines 28, 31, and 34 have the same offsets as the corresponding instructions in DIN1 code.
- Line 29 loads the value 20, which corresponds the DIN2 de-bounce time (5 = 20 x ¼ second)
- Lines 30 and 32 use the address 40502 as this is the timer register used for DIN2.
- The NO_OP (no-operation) instructions at Line 13, 25,37 and 38 are not required. They are only included to assist with readability.

Logic Arguments

Instruction	I	N	{	Description	Argument
LOAD				**Load the Accumulator**	
LOAD				Load a value from memory to the accumulator. 32-bit counter: MSW at the high (Even) address. Float: Loads the integer part only (0-65535)	Memory Register to load from
LOAD	I			Load an immediate value to the accumulator	The actual value to load to accumulator
LOAD		N		Invert and Load to accumulator Discrete: ON gives “0”; OFF gives “1”. Other types: bitwise invert e.g. 0xFACE gives 0x0531	Memory Register to load from
LOAD			{	Calculate Memory Register to Load from within the { }. The accumulator value is loaded from the location that has been calculated when the “}” statement is reached.	Initial value for the Memory Register calculation
STORE				**Store the Accumulator to memory**	
STOR				Save value from accumulator to memory	Memory location to save to
STOR		N		Invert accumulator and save. When storing to a bit Register, a non-zero value is stored as off, and zero is stored as on.	Memory location to save to
STOR			{	Calculate Memory Register to Store to within the following instructions { }. The current accumulator value is saved to the location that has been calculated when the “}” statement is reached.	Initial value of Register calculation
Delayed Calculation			{	**Calculate the Second Argument of a statement** Use this feature when you need multiple steps to calculate the second argument of a statement.	
			{	Check the “{“Column to begin calculation of the argument to a statement. This works for LOAD, STOR and for all of the Logic and Math operations, as well as for the Test/Comparison operations.	Initial value to load for the calculation
}				Complete and execute a delayed Calculation. This matches the opening brace flag “{“in the LOAD, STORE, Arithmetic, Logical, and Comparison commands. It completes the calculation of the	Argument Ignored

				argument value and executes the original command.	
SET/RESET				**Set or Clear a bit**	
SET				Set memory register to "1" if accumulator is nonzero. Unchanged if accumulator is zero.	Memory location to set
SET		N		Set memory register to "1" if accumulator is zero.	Memory location to set
RES				Clear memory register if accumulator is non-zero. Unchanged if accumulator is zero.	Memory location to clear
RES		N		Clear memory register if accumulator is zero.	Memory location to clear
LOGIC/MATH				Bitwise Logical and Arithmetic operations	
AND OR XOR ADD SUB MUL DIV				Perform Logical / Arithmetic operation between Accumulator and memory. Result is saved in the accumulator. AND, OR, XOR – Bitwise Op ADD – 16-bit addition SUB – 16-Bit subtraction MUL – Multiplication (Mod 65535) DIV – Division (x / 0 = 0)	Register index of the value to use for the second operand
AND …. DIV	I			Perform Logical / Arithmetic operation between Accumulator and Immediate value	Immediate value to use for the second operand
AND …. DIV		N		Negate the argument (Bitwise invert) before performing the operation.	Applies to Register, Immediate and delayed calculation.
AND …. DIV			{	Perform Logical / Arithmetic operation between Accumulator and the result of the following calculation within the { }	Initial memory location or immediate value (I) for calculation of second operand.
TEST				**Compare two Values**	
GT GE EQ NE LE LT				Perform Comparison operation between Accumulator and memory. Accumulator gets "1" if comparison true. "0" if false. GT – Greater Than GE – Greater or Equal EQ – Equal To NE – Not Equal LE – Less or equal LT – Less Than	Register index of the value to use for the second operand of the comparison
GT …. LT	I			Perform Comparison operation between Accumulator and Immediate value. Accumulator gets "1" if comparison true. "0" if false.	Immediate value to use for the second operand of the comparison
GT …. LT		N		Negate the argument (two's compliment) before performing the comparison	Applies to Register, Immediate and delayed calculation forms.
GT …. LT			{	Perform Comparison operation between Accumulator and the result of the following calculation within the { }	Initial memory location or immediate value (I) for calculation of second operand.
JUMP				**Transfer a Control to a new Location**	
JMP				Jump to instruction	Line number to jump to
JMP	I			Jump forward or backward from the current location	0-9999: Jump Forward 10000+: Jump backward

				the number of lines specified	
JMP_C				Conditional Jump if accumulator is non-zero	Line jump to if accumulator is non-zero
JMP_C		N		Conditional Jump if accumulator is zero	Line number to jump to if accumulator is zero.
JMP_C	I			Conditional Jump forward or backward from the current location the number of lines specified	0-9999: Jump Forward 10000+: Jump backward
CALL / RETURN				**Call a Subroutine and Return**	
CALL				Call a subroutine. A subroutine will execute the listed statements until a “RET” statement is reached, where control returns to the line following the CALL statement.	Line number of first instruction of the subroutine to call
CALL	I			Call a subroutine forward or backward from the current location offset from current location	0-9999: call Forward 10000+: call backward
CALL_C				Conditional Call if accumulator is non-zero. (otherwise continue to next line)	Line number to call if accumulator is non-zero
CALL_C		N		Conditional Call if accumulator is zero	Line number to call if accumulator Is zero.
CALL_C	I			Conditional Call a subroutine forward or backward from the current location, offset from current location	0-9999: Jump Forward 10000+: Jump backward
RET				Return from subroutine. Returns to the instruction following the last executed CALL instruction.	Argument Ignored
RET_C				Return to calling address if accumulator is non-zero	Argument Ignored
RET_C		N		Return to calling address if accumulator is zero	Argument Ignored

Amendment Register:

Issue No.	Date	Details of Amendment
1.8	6/3/18	Minor Edits
1.9	5/6/20	Added Save file and made more module generic.
1.10	1/12/20	Minor changes, added Scale Register example (#5), Load Config.