MBLogicEngine 17-Sep-2010

Table of Contents

Overview:

MBLogicEngineCk is a soft logic library. It can be incorporated into other programs to give them a soft logic (PLC like) functionality. Typical applications would be in systems such as automated production line test systems, PC based labeling systems, or vision applications.

By incorporating a soft logic library in a custom application you can provide a means of programming and controlling auxiliary machine functions (conveyors, beacons, tooling, etc.) using conventional PLC programming syntax without requiring an actual PLC. This allows for closer and better integration between the machine control functions and your PC application while still providing flexible control of the auxiliary machine functions using programming languages and concepts that are familiar to ordinary plant personnel and maintenance teams. The soft logic program can be updated and reloaded without affecting your main PC application.

MBLogicEngineCk is used in the MBLogic automation server to provide its soft logic functions. This document describes the MBLogicEngineCk instruction set and also how to use MBLogicEngine as a library in your own applications.

Soft Logic Specs

Overview:

The following are the specifications for the "Ck" soft logic library.

General Specifications:

Characteristic	Limits
Data table size	50 kbytes
Digital inputs	2000
Digital outputs	2000
Analogue inputs	125 *
Analogue outputs	125 *
Timers	500
Counters	250
Maximum number of instructions	No limit **
Maximum number of subroutines	No limit **

* Plus an equal number of signed addresses. In addition to these addresses, any register type can be used for I/O, not just the "analogue" addresses.

** There is no enforced limit to the size of program that can be run. The practical limitations are related to the resulting scan rate and the time to load the program on start up.

Instructions

The instructions are documented in detail elsewhere. The following is just a brief summary.

Instruction type	# of Instructions
Boolean input	8
Edge contact	6
Boolean output	8
Comparison	18
Program control	8
Counter and timer	6
Copy	5
Search	12
Shift register	1
Math operators and functions	26

Instruction Speed:

The actual instruction speed achieved in any application will depend on the speed of the computer used. The following are some representative times from a relatively slow (1.8GHz Sempron) PC.

All times are in micro-seconds per instruction.

Boolean Inputs / Edge Contacts / Compare

Instruction	Speed
STR / STRN	0.63
AND / ANDN	0.13
OR / ORN	0.13
ANDSTR / ORSTR	0.68
ANDND / ORND	0.86
ANDPD / ORPD	0.75
STRND / STRPD	1.33
ANDE / ANDNE (reg / reg)	0.13
ANDE / ANDNE (3 char string / reg)	0.21

Boolean Outputs

Instruction	Single bit	2 bits	16 bits	128 bits
OUT	0.20	3.57	7.04	34.22
PD	0.88	4.38	7.95	35.55
RST / SET (rung true)	0.33	3.31	6.78	34.05
RST / SET (rung false)	0.11	0.11	0.11	0.11

Counter / Timer / Math

Instruction	Speed
CNTU	2.53
TMR	10.32
TMRA	9.38
MATHDEC (x + y * z / a MOD b)	5.45
MATHDEC (SIN 0.5)	5.01
MATHHEX (LSH(reg, h2))	5.16

Shift Register

Instruction	16 bits	128 bits
SHFRG	9.13	40.94

Pack / Unpack

Instruction	1 bit	16 bits
PACK	5.54	11.82
UNPACK	10.21	15.58

Single Register Copy

Instruction	Speed
COPY (constant to register without one shot) *	0.75
COPY (constant to register with one shot) *	1.5
COPY (register to register of the same type without one shot) **	0.94
COPY (register to register of the same type with one shot) **	1.7
COPY (register to register of different type) ***	9.9
COPY (integer to TXT conversion, 4 digits) ****	25
COPY (string constant to TXT registers) *****	24 (+ 0.4 per char)
COPY (constant to pointer)	16
COPY (register to pointer)	8.6
COPY (pointer to register)	17
COPY (pointer to pointer)	21
COPY (rung false - no operation)	0.11

* Copying a single character constant (e.g. "A") is also a constant to register copy operation.

** Registers are considered to be of the "same" type for copy operations when a value in the source register can always be guaranteed to fit within the destination register and are of the same basic type. For example, "DS" to "DD" will always fit, but "DD" to "DS" may not fit. "DD" and "DF" are of different basic types (integer versus floating point) and so are different for copy purposes.

*** The source type must be capable of being converted to the destination type.

**** This involves an integer to string conversion plus a string copy.

***** E.g. copying "ABCDE" to TXT10 - TXT14 will take 24 + (0.4 * 5) = 26.

Multi-Register Operations

Instruction	2 regs	100 regs	1000 regs
CPYBLK	14.2	360.32	3669.33
FILL	7.97	31.74	246.88
FINDEQ	8.23	167.66	1821.93
SUM	7.88	36.05	331.68

Miscellaneous

Instruction	Speed
NETWORK (start new rung)	0.51
FOR / NEXT (per empty loop iteration)	0.08
CALL / RT (call plus return from empty subr)	3.93

Instruction Set:

The instruction set implemented in the "Ck" logic engine is very similar to that of the Koyo "Click". The "Ck" logic engine is not however intended to be an emulator or direct replacement for the "Click". It is intended to be a useful soft logic library that is similar enough to the "Click" that someone already familiar with one will find the other easy to learn and understand.

Instructions Not Implemented, or Implemented Differently

Certain instructions present in the Koyo "Click" were either not implemented in the "Ck" logic engine, or were implemented differently.

- The communications functions were not implemented. The "Ck" logic engine is part of a complete stand alone soft logic system. Communications features are provided by the rest of the supporting software.
- The SUM math function was implemented as a separate stand alone instruction rather than being integrated into the MATHDEC and MATHHEX instructions.
- Floating point math calculations are performed as double precision rather than single precision operations. This may result in more accurate, but slightly different results.
- The DRUM instructions are not implemented at this time. (They may be added in a future version).
- The watch dog timer is not implemented at this time. (This is intended to be in a future revision).
- The digital I/O system is addressed differently. The digital I/O (X, Y) is arranged as a continuous address space. The I/O registers (XD, YD) are separate address spaces from the digital I/O, and may be used for analogue and other word I/O. To address digital I/O as words, use the PACK and UNPACK instructions.
- Signed analogue I/O addresses (XS and YS) have been added. These are more generally useful than the unsigned (XD, YD) equivalents.
- There are no immediate I/O or interrupts, as these are PLC CPU on board I/O hardware features which are not present in a soft logic system.
- There is no fixed scan mode.
- Hardware related system control relay (SC) and system register (SD) addresses which relate only to "Click" hardware features are not implemented.

Soft Logic Addresses

Overview:

Data for the "Ck" soft logic library is stored in a data table which is divided into distinct types. The sizes and limits on the data table and constants are defined below.

Data Types:

Type	Size (bytes)	Min Value	Max Value
Signed integer	2	-32768	32767
Signed double integer	4	-2147483648	2147483647
Floating point	Undefined	-3.40E+038	3.40E+038 *
Unsigned integer (hex)	2	0	65535
Single character	1	N/A	N/A

Text string	Undefined	N/A	N/A

* The maximum constant that can be entered is +/- 1.9e307

Data Table Addresses:

All data is stored in data table addresses. These addresses are either "bit" (boolean) or "register" (word) addresses. Each data table begins at "1" (not "0"), and extends to the maximum address listed below.

Prefix	Description	Max	Type	Example
X	Inputs	2000	Boolean	X9
Y	Outputs	2000	Boolean	Y1023
C	Control relays	2000	Boolean	C58
T	Timer status	500	Boolean	T421
CT	Counter status	250	Boolean	CT6
SC	System control	1000	Boolean	SC21
DS	Data register	10000	Integer	DS6432
DD	Data register	2000	Double integer	DD4
DH	Data register	2000	Unsigned integer	DH100
DF	Data register	2000	Floating point	DF57
XD	Input register	125	Unsigned integer	XD1
YD	Output register	125	Unsigned integer	YD99
XS	Input register	125	Signed integer	XS1
YS	Output register	125	Signed integer	YS99
TD	Timer current value	500	Integer *	TD45
CTD	Counter current value	250	Double integer *	CTD3
SD	System data	1000	Integer	SD9
TXT	Text data	10000	ASCII	TXT7420

* Only positive numbers (0 to max) are allowed for these addresses.

Pointers or Indirect Addressing

Pointer (indirect) addressing is provided by the COPY instruction. See the documentation on that instruction for details.

Differences from the Koyo "Click"

The maximum address permitted by the Ck soft logic library is in some cases larger than that permitted by the Koyo "Click".

The Ck library implements the X, Y, XD, and YD addresses slightly differently from the Koyo "Click". This is because the I/O hardware is different, which means that an exact correspondence would make little sense.

The differences are as follows:

- X & Y - All addresses from 1 up to the maximum are permitted as legal addresses. The mapping of physical I/O to "X" & "Y" addresses is configuration dependent.

- XD & YD - These registers are not related to the "X" or "Y" bit addresses. That is, the states of "X" or "Y" addresses are not reflected by the values in the "XD" or "YD" registers. Instead, these registers are intended for analogue and other word I/O. The mapping of physical I/O to "XD" & "YD" addreses is configuration dependent.
- XD & YD register addresses begin at "1" in order to be consistent with the numbering system used for other registers.
- XS & YS register addresses are an additional register type. They differ from the XD and YD addresses in that they are signed rather than unsigned integers. These may be more convenient when dealing with signed integer data.
- Any data type may be used for interfacing with external I/O. For example, it is possible to transfer data directly between external I/O and C or DS addresses. X, Y, XD, YD, XS, and YS addresses however are recommended for this purpose.

Representation

Data table addresses must be represented exactly as defined. Case is significant, so while X1 is a valid address, x1 is not. Leading zeros are not permitted. X1 is a valid address, X01 is not.

Constants:

Many instructions which accept registers addresses as inputs will also accept constants. The type of constant must be compatible with the parameter expected by the instruction.

Code	Type	Example
KInt	Signed integer	125
KDInt	Signed double integer	-50000
KF	Floating point	123.456 or 1.23456E+2
KHex	Unsigned integer (hex)	f73h
KTxtChar	Single character	"A"
KTxtStr	Text string	"abc123"

- Hexadecimal constants must have an "h" suffix to be valid. The "h" must be lower case.
- Text constants must be enclosed in double quotes (").
- Integers and double integers are distinguished by the magnitude of the value. An integer constant is assumed to be a normal (single) integer unless it exceeds the maximum permitted value, in which case it is automatically promoted to a double integer. Any instruction which expects a double integer will also accept a regular integer.
- Floating point numbers may be represented in decimal or exponential notation.
- Leading zeros are permitted, and are not significant.

System Control Relays:

System control relays are used to provide convenience functions, or to signal error conditions.

- Error relays are set immediately by the instruction which encountered the error, and remain in effect until they are reset by another instruction which uses that relay, or until the system is restarted.
- Free running clock relays are updated at the start of each scan.
- The "always on" relay is set at the start of scan.

Address	Description
SC1	Always ON

SC2	ON for the first program scan
SC3	Turns on and off on alternate scans
SC4	Free running clock with a 10ms period
SC5	Free running clock with a 100ms period
SC6	Free running clock with a 500ms period
SC7	Free running clock with a 1sec period
SC8	Free running clock with a 1m period
SC9	Free running clock with a 1 hour period
SC19	ON if an error occurs
SC24	There is no PLC program, or the program is invalid.
SC25	The run time system is not compatible with the program version.
SC26	The watch dog timer timed out. *
SC27	The data table contents were lost.
SC40	An attempt was made to divide by zero.
SC43	Data overflow, underflow, or data conversion error.
SC44	An invalid address was used
SC46	A math error occurred
SC50	Set run mode to stop. *
SC51	Resets watchdog timer to zero. *

Notes: * = Not implemented at this time. The SC relays are not writable by a user instruction.

System Control Registers:

System control registers are used to provide convenience functions, or to signal error conditions.

Address	Description
SD1	Current PLC error code *
SD5	Low word of runtime version *
SD6	High word of runtime version *
SD9	Counts number of scans (rolls over at 32,767)
SD10	The current scan time
SD11	The minimum scan time since starting
SD12	The maximum scan time since starting
SD20	Real time clock year
SD21	Real time clock month
SD22	Real time clock day
SD23	Real time clock day of week (where Mon = 0) #
SD24	Real time clock hour
SD25	Real time clock minute
SD26	Real time clock second

* = Not implemented at this time. # = This is different than the Click hardware.

Subroutines:

Subroutines are referenced by names which may be from 1 to 24 characters long and which may contain the charcters a to z, A to Z and 1 to 9. No other characters are permitted, and no spaces are permitted within the name. Examples, "SubroutineName1", or "123ValidName". The following are invalid names "ThisNameIsMuchTooLongToBeAccepted", and "Bad&Characters*"

There is no limit to the number of subroutines permitted, but each subroutine name must be unique.

Soft Logic Instructions

Overview:

This page provides a description of each instruction list (IL) instruction used in the Ck soft logic library. "Ck" is a soft logic library which is modeled closely on the Koyo "Click" PLC. The Ck library is not intended to provide an exact emulation of the "Click". Rather it is intended to provide an instruction set which is similar enough to the "Click" that someone who is already familiar with the "Click" will understand the "Ck" library fairly easily.

The most significant difference between the two systems is the Koyo "Click" does not have a documented IL instruction set. The IL instructions used below have been interpolated from a combination of the ladder instructions used by the "Click", and the IL instructions used in the Koyo DL-205.

This document is not intended as a primer in PLC programming in general or in IL programming. The reader is assumed to be already familiar with PLC programming concepts, conventions, and terminology.

Instruction Summary:

Instruction	Instr Type	Description
//	Program Formatting	Comment line
AND	Boolean input	AND bit with top of logic stack
ANDE	Compare	AND if parm1 equals param2
ANDGE	Compare	AND if parm1 >= parm2
ANDGT	Compare	AND if parm1 > parm2
ANDLE	Compare	AND if parm1 <= parm2
ANDLT	Compare	AND if parm1 < parm2
ANDN	Boolean input	AND NOT bit with top of logic stack
ANDND	Edge contact	AND negative differential
ANDNE	Compare	AND if parm1 is not equal to parm2
ANDPD	Edge contact	AND positive differential
ANDSTR	Boolean input	AND top two values on logic stack
CALL	Program control	Call subroutine
CNTD	Counter/Timer	Count down
CNTU	Counter/Timer	Count up

COPY	Copy	Copy a single value to a register
CPYBLK	Copy	Copy a block of data
END	Program control	Program end
ENDC	Program control	Program end conditional
FILL	Copy	Fill a block of data
FINDEQ	Search	Search table for equal to
FINDGE	Search	Search table for >=
FINDGT	Search	Search table for >
FINDIEQ	Search	Incremental search table for equal to
FINDIGE	Search	Incremental search table for >=
FINDIGT	Search	Incremental search table for >
FINDILE	Search	Incremental search table for <=
FINDILT	Search	Incremental search table for <
FINDINE	Search	Incremental search table for not equal
FINDLE	Search	Search table for <=
FINDLT	Search	Search table for <
FINDNE	Search	Search table for not equal
FOR	Program control	For/next loop
MATHDEC	Math	Decimal math
MATHHEX	Math	Hexadecimal math
NETWORK	Program Formatting	Network
NEXT	Program control	Next in For/next loop
OR	Boolean input	OR bit with top of logic stack
ORE	Compare	OR if parm1 equals parm2
ORGE	Compare	OR if parm1 >= parm2
ORGT	Compare	OR if parm1 > parm2
ORLE	Compare	OR if parm1 <= parm2
ORLT	Compare	OR if parm1 < parm2
ORN	Boolean input	OR NOT bit with top of logic stack
ORND	Edge contact	OR negative differential
ORNE	Compare	OR if parm1 is not equal to parm2
ORPD	Edge contact	OR positive differential
ORSTR	Boolean input	OR top two values on logic stack
OUT	Boolean output	Output logic stack to bit
OUT	Boolean output	Output logic stack to multiple bits
PACK	Copy	Pack bits into a register
PD	Boolean output	Output logic stack one shot to multiple bits
PD	Boolean output	Output logic stack one shot

RST	Boolean output	Reset multiple bits if logic stack true
RST	Boolean output	Reset bit if logic stack true
RT	Program control	Return from subroutine
RTC	Program control	Return from subroutine conditional
SBR	Program control	Define a subroutine
SET	Boolean output	Set multiple bits if logic stack true
SET	Boolean output	Set bit if logic stack true
SHFRG	Shift register	Shift register move bits to right
STR	Boolean input	Store bit onto logic stack
STRE	Compare	STR if parm1 equals parm2
STRGE	Compare	STR if parm1 >= parm2
STRGT	Compare	STR if parm1 > parm2
STRLE	Compare	STR if parm1 <= parm2
STRLT	Compare	STR if parm1 < parm2
STRN	Boolean input	Store NOT bit onto logic stack
STRND	Edge contact	STORE negative differential
STRNE	Compare	STR if parm1 is not equal to parm2
STRPD	Edge contact	STORE positive differential
SUM	Math	Sum a range of registers
TMR	Counter/Timer	On delay timer
TMRA	Counter/Timer	On delay accumulating timer
TMROFF	Counter/Timer	Off delay timer
UDC	Counter/Timer	Up/down counter
UNPACK	Copy	Unpack bits from a register

The Logic Stack:

A logic stack is one of the fundamental features of all conventional PLCs, although its presence is often not explicitly mentioned. The logic stack is the location where the results of logic operations are stored, and where many instructions implicitly draw one or more of their Boolean parameters.

The logic stack is implemented as a stack. This stack has an undefined maximum size, but the limit may be assumed to be very large, and for all practical purposes unlimited. The minimum size is zero.

Instructions which use the logic stack may push a value onto the stack, modify the top of the stack, read the top of the stack, or read any value below the top of the stack. In the instruction documentation, the top of the logic stack is refered to as "the top of the logic stack", "top of stack", or just "logic stack". Values in locations below the top of the stack are refered to as the logic stack value (top - 1) (the value just below the top of stack), or logic stack value (top - 2)(the value two positions below the top of stack). No instruction reads more than two positions below the top of stack.

Whenever a new network (rung) is started, the logic stack is cleared and the top of the stack is initialised to "false". If an instruction attempts to read a logic stack location which does not exist a value of "false" is returned to the instruction. This means that attempting to read an invalid logic stack location does not result in a run time error, but it does mean the instruction will never become true.

When a subroutine is called, the logic stack is re-initialised. When the subroutine returns, the state of the logic stack is undefined. This means that a subroutine must not rely on the logic stack being in a particular state when called, and the calling location must not rely on the logic stack being either set by the subroutine or being in the same state as before the subroutine call.

Example:

```
// Re-initialise the logic stack.
// Push a value onto the stack. The top of stack is now true.
NETWORK 1
STR SC1

// Re-initialise the logic stack again.
// The previous value is now gone.
NETWORK 2

// Re-initialise the logic stack again.
// Push four values onto the stack. The top is true,
// (top - 1) is false, (top - 2) and (top - 3) are true.
NETWORK 3
STR SC1
STR SC1
STRN SC1
STR SC1

// Re-initialise the logic stack again.
// The previous values are now gone.
NETWORK 4
```

Program Formatting Instructions:

Program formatting instructions do not directly affect the program flow or results, but they do act to help document and organise a program.

Instruction	Description	# Params	Parameters
//	Comment line	Undefined	Arbitrary text
NETWORK	Network	1	Integer

Anything following a comment token ("//") is considered to be a comment and is ignored. The comment token must be separated from the following comment text by one or more spaces.

Example:

```
// This is a comment.
```

A network is also often known as a "rung". The "NETWORK" token is an instruction that resets the logic stack and begins a new block of code. The NETWORK instruction takes an one integer as a parameter. Network (rung) numbers do not have to be unique or consecutive.

Network numbers are used to report syntax errors when the program is compiled, and are also used to report run time errors.

Example:

```
NETWORK 1
```

Boolean Input Instructions:

Boolean input instructions read a Boolean value from the data table or the logic stack, and output the result to the logic stack.

Instruction	Description	# Params	X	Y	C	T	CT	SC
AND	AND bit with top of logic stack	1	X	X	X	X	X	X
ANDN	AND NOT bit with top of logic stack	1	X	X	X	X	X	X
ANDSTR	AND top two values on logic stack	0						
OR	OR bit with top of logic stack	1	X	X	X	X	X	X
ORN	OR NOT bit with top of logic stack	1	X	X	X	X	X	X
ORSTR	OR top two values on logic stack	0						
STR	Store bit onto logic stack	1	X	X	X	X	X	X
STRN	Store NOT bit onto logic stack	1	X	X	X	X	X	X

Boolean input instructions have two types of input parameters, explicit and implicit. A Boolean input instruction will accept no more than one explicit parameter, and zero, one, or two implicit parameters.

- An explicit parameter is a Boolean (bit) address such as X, Y, C, etc.
- An implicit parameter is the value on the top of the logic stack.
- Some instructions use the top two logic stack values as implicit parameters.

All Boolean input instructions output their result by modifying the logic stack.

Boolean input instructions fall into three categories: AND, OR, and STORE.

- AND instructions perform a logical AND on two inputs and store the result by replacing the original value on the top of the logic stack with the new value.
- OR instructions perform a logical OR on two inputs and store the result by replacing the original value on the top of the logic stack with the new value.
- STORE instructions simply store a value on the top of the logic stack, pushing the original value further down on the stack without modifying it.

All Boolean input instructions have both normal and "NOT" versions.

Example:

```
NETWORK 1
STR X1
AND X2
OR X3
OUT Y1

NETWORK 2
STRN C10
ANDN C11
ORN C12
OUT C21

NETWORK 3
STR Y1
STR Y2
```

```
ANDSTR
STR Y3
ORSTR
OUT Y10
```

Edge Contact Instructions:

Edge contact instructions are a type of Boolean input instruction, and operate in a manner similar to that of the normal Boolean input instructions. The difference is that they are one-shot instructions and output a true for one scan only.

Instruction	Description	# Params	X	Y	C	T	CT	SC
ANDND	AND negative differential	1	X	X	X	X	X	X
ANDPD	AND positive differential	1	X	X	X	X	X	X
ORND	OR negative differential	1	X	X	X	X	X	X
ORPD	OR positive differential	1	X	X	X	X	X	X
STRND	STORE negative differential	1	X	X	X	X	X	X
STRPD	STORE positive differential	1	X	X	X	X	X	X

Edge contact instructions can be categorised as "positive differentiate" and "negative differentiate" instructions. "Positive differentiate" instructions are true for one scan when the logic stack makes a false to true transition. "Negative differentiate" instructions are true for one scan when the logic stack makes a true to false transition.

The address used as a parameter is used for an AND, OR, or STORE the operation. It is not necessary to specify an address to store the one shot state, as the instruction stores this information automatically internally.

Example:

```
NETWORK 1
STR X1
ANDPD C1
OUT Y1

NETWORK 2
STR X2
ANDND C2
OUT Y2

NETWORK 3
STR X3
ORPD C3
OUT Y3

NETWORK 4
STR X4
ORND C4
OUT Y4

NETWORK 5
STRPD X5
OUT Y5
```

```
NETWORK 6
STRND X6
OUT Y6
```

Boolean Output Instructions:

Boolean output instructions output the value on the top of the logic stack to an address given as an explicit parameter. Output instructions do not modify the logic stack. This means that multiple output instructions can be used in series without the result of one instruction affecting the next.

Instruction	Description	# Params	X	Y	C	T	CT	SC	Bit Range
OUT	Output logic stack to bit	1		X	X				
OUT	Output logic stack to multiple bits	2		X	X				X
PD	Output logic stack one shot	1		X	X				
PD	Output logic stack one shot to multiple bits	2		X	X				X
RST	Reset bit if logic stack true	1		X	X				
RST	Reset multiple bits if logic stack true	2		X	X				X
SET	Set bit if logic stack true	1		X	X				
SET	Set multiple bits if logic stack true	2		X	X				X

Boolean output instructions come in two forms: single output and bit range output. The system distinguishes between the two forms by the number of parameters given.

The PD instruction is a one-shot instruction. The output will be turned on for one scan if the logic stack input is true.

Example:

```
// Output to a single bit.
NETWORK 1
STR X1
OUT Y1
RST Y2
SET Y3

// Output to a range of bits.
NETWORK 2
STR X2
OUT C1 C9
RST C10 C19
SET C20 C21

// Output using one-shot.
NETWORK 3
STR X3
PD C30
PD C40 C60
```

Comparison Instructions:

Comparison instructions compare two word parameters and output the Boolean result to the logic stack.

Instruction	Description	# Params
ANDE	AND if parm1 equals param2	2
ANDGE	AND if parm1 >= parm2	2
ANDGT	AND if parm1 > parm2	2
ANDLE	AND if parm1 <= parm2	2
ANDLT	AND if parm1 < parm2	2
ANDNE	AND if parm1 is not equal to parm2	2
ORE	OR if parm1 equals parm2	2
ORGE	OR if parm1 >= parm2	2
ORGT	OR if parm1 > parm2	2
ORLE	OR if parm1 <= parm2	2
ORLT	OR if parm1 < parm2	2
ORNE	OR if parm1 is not equal to parm2	2
STRE	STR if parm1 equals parm2	2
STRGE	STR if parm1 >= parm2	2
STRGT	STR if parm1 > parm2	2
STRLE	STR if parm1 <= parm2	2
STRLT	STR if parm1 < parm2	2
STRNE	STR if parm1 is not equal to parm2	2

All comparison instructions take two explicit parameters and output their result to the logic stack. Each parameter is a word register or constant.

There are three categories of comparison instruction: AND, OR, and STORE. Each of these operates in a manner similar to to the corresponding Boolean input instructions, with the exception that they apply the Boolean result of the comparison operation to to top of the logic stack.

For any comparison operation, both parameters must be compatible with each other. Parameters are compatible if they are both within the same compatibility group. The compatibility groups are defined as follows:

- Signed numeric - DS, DD, DF, TD, CTD, SD, KInt, KDInt, KF
- Unsigned numeric (hex) - DH, XD, YD. KHex
- Text - TXT, KTxtChar, KTxtStr

The type and address abbreviations are defined in the section on addresses and constants.

If one parameter is a string constant (e.g. "abc123"), and the other is a text register, the string constant will be compared to the series of text registers beginning at the register specified. The comparison will be on a character by character basis, such that the comparison will fail at the first character that does not meet the comparison criteria.

For example, if the string "hijklanop" is compared to to see if it is greater that a series of registers containing "abcdefghi", the comparison will fail on the sixth character because "a" is not greater than "f".

Example:

```
// Compare signed numbers.
NETWORK 1
STRE DS1 DS2
ANDGT 123 DD10
ORLE DF20 123.876
ORGE DF5 DS3
ANDLT 93 CTD3
OUT Y1

// Compare unsigned (hex) numbers.
NETWORK 2
STNE 123fh DH52
ANDGE YD4 DH2
SET C20

// Compare text.
NETWORK 3
STR X3
ANDE TXT5 TXT10
ORGE "A" TXT12
ANDE "pass" TXT50
OUT C35
```

Program Control Instructions:

Program control instructions control the flow of the PLC program.

Instruction	Description	# Params	Subr	DS	KInt	One Shot
END	Program end	0				
ENDC	Program end conditional	0				
FOR	For/next loop	1 or 2		X	X	X
NEXT	Next in For/next loop	0				
SBR	Define a subroutine	1	X			
CALL	Call subroutine	1	X			
RT	Return from subroutine	0				
RTC	Return from subroutine conditional	0				

Program END

END and ENDC will terminate a program scan. END terminates the scan unconditionally, while ENDC will terminate the scan if the logic stack is true.

Example:

```
// END.
NETWORK 100
STR SC1
AND X5
ENDC
STRN SC1
END
```

Subroutines

SBR, CALL, RT, and RTC are subroutine instructions. SBR is used to define a subroutine. CALL will call a subroutine. SBR and CALL expect a valid subroutine name as a parameter. RT will return from a subroutine unconditionally. RTC will return from a subroutine if the logic stack is true.

Example:

```
// Subroutines.
NETWORK 1
STR SC1
OUT Y1
CALL SubTest

NETWORK 2
STR C47
OUT Y19

END

// Define subroutine.
SBR SubTest
NETWORK 1
STR SC1
RST Y10
AND X21
RTC
SET C47
RT
```

For/Next

FOR and NEXT are used to define a loop which executes a specified number of times. FOR begins the loop, and NEXT defines the end of the loop. FOR expects an integer parameter specifying the number of loops to perform. This may be a register or constant. FOR also accepts an optional parameter specifying whether the instruction should execute as a one-shot. FOR/NEXT loops may be nested.

Example:

```
// FOR/NEXT with constant.
NETWORK 1
STR SC1
FOR 10
OUT Y1
NEXT

// FOR/NEXT with register.
NETWORK 2
STR SC1
FOR DS1
OUT Y2
NEXT

// FOR/NEXT with one-shot.
NETWORK 2
STR SC1
AND C1
```

```
FOR 100 1
OUT Y2
NEXT
```

Errors

Using RT or RTC in the main program will cause a run-time error. Using NEXT without FOR may cause a run-time error. Using FOR without NEXT may cause unpredictable run-time operation.

Subroutines may call other subroutines, and they may even call themselves (recursion is permitted). However, if if the nesting level reaches an excessive level, a run time error will occur and the program will exit. The maximum nesting level is undefined, but is typically about 1000.

Counter and Timer Instructions:

Instruction	Description	# Params	T	CT	DS	DD	KInt	KDInt	Time Base
CNTU	Count up	2		X	X	X	X	X	
CNTD	Count down	2		X	X	X	X	X	
UDC	Up/down counter	2		X	X	X	X	X	
TMR	On delay timer	3	X		X		X		X
TMRA	On delay accumulating timer	3	X		X		X		X
TMROFF	Off delay timer	3	X		X		X		X

Counters

Counter instructions come in three types:

- CNTU - Up counter. This counts up by one each time the logic stack transitions from off to on. If the logic stack value (top - 1) turns on, the counter is reset.
- CNTD - Down counter. This counts down by one each time the logic stack transitions from off to on. If the logic stack value (top - 1) turns on, the counter is reset.
- UDC - Up/down counter. This counts up by one each time the top of the logic stack transitions from off to on, and down by one each time the logic stack value (top - 1) transitions from off to on. When the logic stack value (top - 2) turns on, the counter is reset

All counter instructions take two parameters. The first is a counter number, and the second is the counter preset. The counter preset may be either a constant or a register address. When the present value of the counter is equal to the preset, the counter status bit turns on. The address of the counter status bit is the same as the counter number. The present value of the counter is stored in the counter data register (CTD) of the same number as the counter number. E.g. for counter CT5, the counter status bit is also CT5, and the present count is in CTD5.

Example:

```
// Up counter.
NETWORK 1
STR C1
STR C2
CNTU CT5 155793
STR CT5
```

```
OUT Y1

// Down counter.
NETWORK 2
STR C11
STR C12
CNTD CT15 DD11
STR CT15
OUT Y11

// Up/down counter.
NETWORK 3
STR C21
STR C22
STR C23
UDC CT25 DS5
STR CT25
OUT Y21

// Examine counter present values.
NETWORK 7
STRE CTD15 157
OUT Y112
```

Timers

Timers come in three types:

- TMR - On delay timer. The timer begins timing up when the top of the logic stack is true, and resets when the top of the logic stack is false. When the present value reaches the preset value, the timer status bit turns on.
- TMRA - On delay accumulating timer. The timer begins timing up when the top of the logic stack is true, and stops when the top of the logic stack is false. When the top of the logic stack becomes true, it continues timing from the point at which it stopped. When the logic stack value (top - 1) turns on, the timer is reset. When the present value reaches the preset value, and the top of the logic stack is true, the timer status bit turns on.
- TMROFF - Off delay timer. The timer begins timing when the top of the logic stack transitions from on to off. When the timer begins timing, the status bit turns on immediately. The status bit turns off when the timer reaches the preset.

All timer instructions take three parameters. The first is the timer address. The second is the timer preset. The timer preset may be either a constant or a register address. The third parameter is the time base. This may be either "ms" (milliseconds), "sec" (seconds), "min" (minutes), "hour" (hours), or "day" (days).

Example:

```
// On delay timer.
NETWORK 4
STR X1
TMR T5 25000 ms
STR T5
OUT Y51

// Accumulating on delay timer.
NETWORK 5
```

```
STR X11
STR X12
TMRA T6 DS9 sec
STR T6
OUT Y52

// Off delay timer.
NETWORK 6
STR X62
TMROFF T192 DS412 min
STR T192
OUT Y57

// Examine timer present values.
NETWORK 7
STRE TD6 97
OUT Y120
```

Copy Instructions:

Copy instructions are used to copy data between registers, convert register data between types, and copy data between registers and Boolean (bit) addresses. Copy instructions are conditional output instructions which execute only if the logic stack is true. They also have an option to execute in a one-shot manner. Errors are indicated in the system control relays (SC43, SC44).

Instruction	Description	# Params	One Shot	Pointers	SC43	SC44
COPY	Copy a single value to a register	2 or 3	X	X	X	X
CPYBLK	Copy a block of data	3 or 4	X		X	
FILL	Fill a block of data	3 or 4	X		X	
PACK	Pack bits into a register	3 or 4	X			
UNPACK	Unpack bits from a register	3 or 4	X			

Copy Single

The Copy Single (COPY) instruction is used to copy a single register, or constant (including a text string) to another register (or series of registers for text string constants). It is also the only instruction which will accept pointers as a source or destination.

The COPY instruction expects the following parameters:

- Source - This may be a register, a constant, a string constant, or a pointer. Register types may be DS, DD, DF, DH, TD, CTD, SD, TXT, XD, YD, XS, or YS. Constant types may be KInt, KDInt, KF, KHex, KTxtChar, KTxtStr.
- Destination - This must be a register. Register types may be DS, DD, DF, DH, TD, CTD, TXT, XD, YD, XS, or YS.
- An optional one shot parameter of 0 or 1 (see below).

COPY *source destination (one-shot)*

The COPY instruction will convert types as necessary when copying between otherwise incompatible registers. If a copy is made from a numeric register or constant to a text register, the digits will be converted to a text string, and one character being placed in each consecutive register. As many registers will be used as are necessary to hold the resulting string.

The error flags are set under the following conditions:

- SC43 - The data could not be converted to the correct type, or the data value is out of range of the destination register.
- SC44 - The pointer address is out of range, the destination address is out of range or the data is of an unknown type.

Pointer (or indirect) addresses are specified by enclosing the variable part of the address in square brackets. For example: DS[DS100]. This specified a "DS" type register where the numeric part of the address is stored in DS100. If in example the value stored in DS100 was 42, then the effective address for DS[DS100] would be "DS42".

The following register types may use pointer (indirect) addresses:

- DS - For example: DS[DS100]
- DD - For example: DD[DS100]
- DF - For example: DF[DS100]
- DH - For example: DH[DS100]

The variable part of the address (the part enclosed in square brackets) must always be stored in a "DS" register. For example, "DD[DS569]" is correct. However, "DD[DD569]" is incorrect.

Example:

```
// Copy a constant to a register.
NETWORK 1
STR SC1
COPY 100 DS1

// Copy a register to a register.
NETWORK 2
STR SC1
COPY DS1 DD101

// Copy a character to a text register.
NETWORK 3
STR SC1
COPY "A" TXT100

// Copy a character string to six consecutive text registers.
NETWORK 4
STR SC1
COPY "abc123" TXT200

// Copy a number to a text register, converting to text.
NETWORK 5
STR SC1
COPY 123 TXT300

// Copy with a one shot.
NETWORK 6
STR X1
COPY 100 DS200 1

// Copy a constant to a pointer.
NETWORK 7
```

```
STR SC1
COPY 567 DS1000
COPY 100 DS[DS1000]

// Copy a register to a pointer.
NETWORK 8
STR SC1
COPY 567 DS1000
COPY DS100 DS[DS1000]

// Copy a pointer to a register.
NETWORK 9
STR SC1
COPY 567 DS1000
COPY DS[DS1000] DS100

// Copy a pointer to a pointer.
NETWORK 8
STR SC1
COPY 567 DS1000
COPY 568 DS1001
COPY DS[DS1000] DS[DS1001]
```

Copy Block

The Copy Block (CPYBLK) instruction is used to copy a block of consecutive registers to another block of consecutive registers.

The Copy Block instruction expects the following parameters:

- Source Start - The start of the source register range. Register types may be DS, DD, DF, DH, TD, CTD, SD, or TXT.
- Source End - The end of the source register range. This must be a higher address of the same type as the source start.
- Destination Start - The start of the destination register range. The end of the destination register range is calculated automatically from the number of source registers. This must be of types DS, DD, DF, DH, TD, CTD, or TXT and must be of a type compatible with the source registers. There must be a sufficient number of registers available before the end of the address range to accomodate all the source registers.
- An optional one shot parameter of 0 or 1 (see below).

CPYBLK *sourcestart sourceend destinationstart (one-shot)*

The error flags are set under the following conditions:

- SC43 - The data could not be converted to the correct type, or the data value is out of range for the destination register.

Example:

```
// Copy a set of registers.
NETWORK 1
STR Y3
CPYBLK DS1 DS10 DD1
```

```
// Copy with one shot.
NETWORK 2
STR Y3
CPYBLK DS1 DS10 DD1 1
```

Copy Fill

The Copy Fill (FILL) instruction is used to copy a single register or constant to a series of consecutive registers.

The Copy Fill instruction expects the following parameters:

- Source - This may be a register or a constant. Register types may be DS, DD, DF, DH, TD, CTD, SD, or TXT. Constant types may be KInt, KDInt, KF, KHex, or KTxtChar.
- Destination Start - The start of the destination register range. This must be of types DS, DD, DF, DH, TD, CTD, or TXT and of a type compatible with the source.
- Destination End - The end of the destination register range. This must be a higher address of the same type as the destination start.
- An optional one shot parameter of 0 or 1 (see below).

FILL *source destinationstart destinationend (one-shot)*

The error flags are set under the following conditions:

- SC43 - The data could not be converted to the correct type, or the data value is out of range for the destination register.

Example:

```
// Fill a set of registers with a numeric value.
NETWORK 1
STR X1
FILL 1 DS1 DS100

// Fill a set of registers with a value from another register.
NETWORK 2
STR X2
FILL DS1 DD100 DD150

// Fill using a one-shot.
NETWORK 3
STR X3
FILL "A" TXT300 TXT 321 1
```

Copy Pack

The Copy Pack (PACK) instruction is used to pack a series of boolean (bit) values into a single register. Unused bits will be be set to zero.

The Copy Pack instruction expects the following parameters:

- Source Start - The start of the Boolean address range. Boolean types may be X, Y, C, T, CT, SC.
- Source End - The end of the Boolean address range. This must be a higher address of the same type as the source start. There must be no more than sixteen boolean addresses to be

packed.

- Destination - The the destination register. This must be of type DH or YD.
- An optional one shot parameter of 0 or 1 (see below).

PACK *sourcestart sourceend destination (one-shot)*

The PACK instruction does not affect the error relays.

Example:

```
// Pack some inputs into a register.
NETWORK 1
STR C2
PACK X1 X16 DH1
PACK X20 X25 DH2

// Pack, with a one shot.
NETWORK 2
STR C5
PACK X100 X116 DH10 1
```

Copy Unpack

The Copy Unpack (UNPACK) instruction is used to unpack a single register into a series of boolean (bit) addresses.

The Copy Unpack instruction expects the following parameters:

- Source - The register to unpack. Register types may be DH.
- Destination Start - The start of the Boolean address range to unpack to. This must be of types Y, or C.
- Destination End - The end of the Boolean address range. This must be a higher address of the same type as the destination start. There must be no more than sixteen boolean addresses to be unpacked.
- An optional one shot parameter of 0 or 1. If the paramter is set to "1", the one-shot option is enabled and the instruction executes only when the logic stack transitions from false to true. If the paramter is set to "0", the one-shot option is disabled and the instruction executes whenever the logic stack is true. If the parameter is missing, it has the same effect as setting it to "0".

UNPACK *source destinationstart destinationend (one-shot)*

The UNPACK instruction does not affect the error relays.

Example:

```
// Unpack a register.
NETWORK 1
STR C2
UNPACK DH1 C1 C16
UNPACK DH2 C25 C28

// Unpack, with a one shot.
NETWORK 2
STR C5
UNPACK DH5 Y1 Y16 1
```

Error Flags

When any copy instruction which uses either error flag is executed, both error flags (SC43, SC44) are reset. If an error is encountered, the appropriate error flag is set and the operation is aborted.

Search Instructions:

Search instructions are used to search a range of registers to find a value which matches the search criteria. The search value may be a register, constant, or string constant.

Instruction	Description	# Params	One Shot
FINDEQ	Search table for equal to	5 or 6	X
FINDGE	Search table for >=	5 or 6	X
FINDGT	Search table for >	5 or 6	X
FINDIEQ	Incremental search table for equal to	5 or 6	X
FINDIGE	Incremental search table for >=	5 or 6	X
FINDIGT	Incremental search table for >	5 or 6	X
FINDILE	Incremental search table for <=	5 or 6	X
FINDILT	Incremental search table for <	5 or 6	X
FINDINE	Incremental search table for not equal	5 or 6	X
FINDLE	Search table for <=	5 or 6	X
FINDLT	Search table for <	5 or 6	X
FINDNE	Search table for not equal	5 or 6	X

Search instructions each come in two forms - continuous and incremental. A continuous search will begin at the start of the search register range each time it is invoked. An incremental search will begin at the next position after the last match. If the last incremental search did not result in a match before it reached the end of the range, the next search will start at the beginning again.

The search instructions expect the following parameters:

- Search Value - The value to search for. This may be a register, constant, or string constant.
- Start - The beginning of the range of registers to start the search at. This must be compatible with the value being searched for. This must be of type DS, DD, DH, DF, or TXT.
- End - The end of the range of registers being searched. This must be of the same type and a higher address than "start".
- Result - The register to place the search result in. This must be of type DS or DD.
- Result flag - A Boolean address to place the status of the search in. This will be true if the search was successful, or false if not. This must be a "C" address.
- An optional one shot parameter of 0 or 1. If the paramter is set to "1", the one-shot option is enabled and the instruction executes only when the logic stack transitions from false to true. If the paramter is set to "0", the one-shot option is disabled and the instruction executes whenever the logic stack is true. If the parameter is missing, it has the same effect as setting it to "0".

FINDEQ *searchvalue start end result resultflag (one-shot)*

The result of a search is placed in the specified register. If the search was successful (a match was found), the register is set to an integer value equal to the index (number of elements) from the beginning of the search range to where the match was found. If the search was not successful, the result register is set to -1.

When a string constant is used as the search value, the search is conducted by comparing the entire string to a number of registers equal to the length of the string. If a match is not found, the search position increments by one register position and repeats until a match is found or the end of the search range is reached.

Example:

```
// Search for constant.
NETWORK 1
STR X1
FINDEQ 1 DS100 DS110 DS111 C111

// Search for register.
NETWORK 2
STR X2
FINDGT DD112 DD100 DD110 DS112 C112

// Search for character.
NETWORK 3
STR X3
FINDEQ "c" TXT100 TXT110 DS113 C113

// Search for character string.
NETWORK 3
STR X3
FINDEQ "cde" TXT100 TXT110 DS114 C114

// Search with one shot.
NETWORK 5
STR X5
FINDIEQ 15 DS100 DS110 DS115 C115 1
```

Shift Register Instructions:

The shift register instruction implements a shift register which may be of any arbitrary length. It takes two parameters which define the start and end of the shift register. Both must be "C" Boolean addresses. The first parameter may be lower or higher than the second parameter. The shift register will always shift from the lower address to the higher address.

Instruction	Description	# Params	X	Y	C	T	CT	SC	One Shot
SHFRG	Shift register	2			X				

The value on the top of the logic stack is used as an input to the shift register the first location in the shift register will always be equal to the top of the logic stack. When the logic stack value (top - 1) transitions from off to on, the shift register will shift one position from the lower address to the higher address. When the logic stack value (top - 2) is on, the shift register is cleared and all bits are turned off.

The shift register instruction expects the following parameters.

- Start - A Boolean address specifying one end of the shift register.
- End - A Boolean address specifying the other end of the shift register. This must be of the same type as the first parameter.

SHFRG *start end*

Example:

```
// Shift register.
NETWORK 1
STR X1
STR X2
STR X3
SHFRG C10 C20
```

Math Instructions:

Math instructions perform mathematical operations on registers and constants, and output the results to registers.

Instruction	Description	# Params	One Shot	SC40	SC43	SC46
MATHDEC	Decimal math	3	X	X	X	X
MATHHEX	Hexadecimal math	3	X	X	X	X
SUM	Sum a range of registers	3 or 4	X		X	

There are only three math instructions. However, they accept complete equations as input parameters allowing them to perform many different operations.

The decimal and hexadecimal math instructions are similar except for the functions and operators they offer, and for the type of registers they operate on. The general operation of both is desribed here with the details of the functions and operators listed separately below.

The MATHDEC and MATHHEX instruction expects the following parameters:

- Destination - The the destination register. Register types may be DS, DD, DF, DH. The destination register must be compatible with the source registers. DH registers may not be mixed with the other register types.
- A mandatory one shot parameter of 0 or 1. If the paramter is set to "1", the one-shot option is enabled and the instruction executes only when the logic stack transitions from false to true. If the paramter is set to "0", the one-shot option is disabled and the instruction executes whenever the logic stack is true. The one shot parameter must always be specified for these instructions as the variable number of equation elements would otherwise make it impossible to distinguish from from the equations.
- Math equation - All elements following are treated as part of the math equation and must form a legal mathematical equation.

MATHDEC *destination one-shot equationparameters*

MATHHEX *destination one-shot equationparameters*

The error flags are set under the following conditions:

- SC40 - A division by zero was attempted.
- SC43 - The data could not be converted to the correct type, or the data value is out of range for the destination register.
- SC46 - An unspecified math error has occurred. This includes all math errors not covered by the other math error flags.

Example:

```
// Decimal math. Will resolve to 10.77245
NETWORK 1
STR X1
```

```
COPY 2 DS1
MATHDEC DF1 0 (1 + DS1) ^ 2 + SQRT(PI)

// Hexadecimal math. Will resolve to 22h (in hexadecimal)
NETWORK 2
STR X2
COPY 4 DH2
MATHHEX DH1 0 (LSH(DH2, 2h) + 1h) * 2h
```

Decimal Math Operations

Decimal math operations are conducted by functions and operators. The input values may be DS, DD, or DF registers, or decimal constants. The destination may be a DS, DD, or DF register.

Angles for transcendental functions (SIN, COS, etc.) are in radians.

Using a function that operates in floating point (transcendental, logmarithic, square root), or using any floating point number (including the PI constant) will cause the entire equation to be conducted in floating point. The result will then be converted to a type that is compatible with the destination register. If a floating point number is copied to an integer register, the decimal part of the result is truncated, not rounded.

Operation	Description	Example
SIN	Sine	SIN(DF1)
COS	Cosine	COS(DF2)
TAN	Tangent	TAN(DF3)
ASIN	Arcsine	ASIN(DF4)
ACOS	Arccosine	ACOS(DF5)
ATAN	Arctagent	ATAN(DF6)
LOG	Log (base 10)	LOG(DF7)
LN	Natural log	LN(DF8)
SQRT	Square root	SQRT(DD3)
RAD	Convert degrees to radians	RAD(DF21)
DEG	Convert radians to degrees	DEG(DF22)
+	Add	DS1 + 7
-	Subtract	DS3 - DS4
*	Multiply	DS5 * 10
/	Divide	DF7 / DS1
MOD	Modulus (remainder)	DS22 MOD DS21
^	Exponentiate	DF52 ^ 3
PI	The constant PI	PI * DF102
()	Parentheses (group operations)	(DS2 + 71) * DF9

Hexadecimal Math Operations

Hexadecimal math operations are conducted by functions and operators. The input values may be DH registers, or hexadecimal constants. The destination must be a DH register.

Shifts and rotates take place within a 16 bit register size.

Operation	Description	Example
LSH	Shift left a specified amount	LSH(DH1, 2h)
RSH	Shift right a specified amount	RSH(DH2, 2h)
LRO	Rotate left a specified amount	LRO(DH3, 10h)
RRO	Rotate right a specified amount	RRO(DH4, 10h)
AND	AND words	DH5 AND Fh
OR	OR words	DH6 OR 7Fh
XOR	XOR words	DH7 XOR 1Ah
+	Add	DH1 + 7h
-	Subtract	DH3 - DH4
*	Multiply	DH5 * 1fh
/	Divide	DH7 / DH1
MOD	Modulus (remainder)	DH22 MOD DH21
()	Parentheses (group operations)	(DH2 + a1h) * DH9

Sum

The SUM function is provided as a separate instruction instead of being called from within the other math instructions.

The Sum instruction expects the following parameters:

- Source Start - The start of the register address range. Register types may be DS, DD, DF, DH.
- Source End - The end of the register address range. This must be a higher address of the same type as the source start.
- Destination - The the destination register. Register types may be DS, DD, DF, DH. The destination register must be compatible with the source registers.
- An optional one shot parameter of 0 or 1. If the paramter is set to "1", the one-shot option is enabled and the instruction executes only when the logic stack transitions from false to true. If the paramter is set to "0", the one-shot option is disabled and the instruction executes whenever the logic stack is true. If the parameter is missing, it has the same effect as setting it to "0".

SUM *sourcestart sourceend destination (one-shot)*

The error flags are set under the following conditions:

- SC43 - The data could not be converted to the correct type, or the data value is out of range for the destination register.

Example:

```
// Sum a series of registers.
NETWORK 1
STR X1
SUM DS10 DS19 DF20
```

```
// Sum with a one shot.
NETWORK 2
STR X2
SUM DH10 DH19 DH20 1
```

MBLogicEngine - Embedded Library

Overview:

MBLogicEngine can be used as an embedded library inside another application. This allows any application to include soft logic capabilities. This is useful when creating things like automated test systems, labelling systems and other custom PC applications which may have to interact directly with manufacturing equipment. Using MBLogicEngine allows the core of your application to be written as a normal PC application while providing a familiar PLC programming language for the parts which you wish to allow users to customise and change.

Using MBLogicEngine as an Embedded Library:

MBLogicEngine is a pure Python library and can be embedded directly into a custom program. Data is passed to and from the soft logic data table using function calls, and each scan cycle is initiated using another function call. This means that the data table can be integrated directly into a custom program, and each scan cycle coordinated with the overall system.

Soft Logic Scan Cycle:

PLCs have what is generally termed a "scan cycle". The PLC program executes one complete "logic scan" (runs the PLC program all the way through once). Then the PLC updates its I/O and performs any general overhead functions before doing another logic scan.

To use MBLogicEngine as an embedded library in your own program you would need to regularly call the function which executes the PLC program scan. A typical program would do the following:

- 1. Do whatever non-PLC functions the program normally performs.
- 2. Update the input portions of the PLC data table.
- 3. Call the function which executes the PLC program scan.
- 4. Read the output portions of the PLC data table.
- 5. Sleep for a short period of time.
- 6. Repeat the process from step 1.

Initialising the Library:

Library initialisation is performed as follows:

Importing the Libraries

First you must import the required libraries. These must include the following:

Library	File	Provides

PLC compiler	PLCCompile.py	This is the soft logic compiler. This takes the PLC program source file, checks it for errors, and converts it into a form which can be executed.
PLC interpreter	PLCInterp.py	This executes the soft logic program each scan.
Instruction library	DLCkInstructions.py	This is a library which defines each of the soft logic instructions and provides the information required to compile them.
Data table	DLCkDataTable.py	This is a library which defines the soft logic data table (the data memory). The data table size and addressing is pre-defined.
Run time libraries	DLCkLibs.py	This is a library which provides run time support for the soft logic instructions (e.g. math functions, etc.).

Example:

```
from mbsoftlogicck import PLCInterp
from mbsoftlogicck import PLCCompile
from mbsoftlogicck import DLCkInstructions
from mbsoftlogicck import DLCkDataTable
from mbsoftlogicck import DLCkLibs
```

Initialising the Compiler

Next, you must initialise the compiler. This provides the compiler with the list of valid instructions, plus a private data table for use by instructions which must store state information outside of the normal data table (e.g. some PLCs store differentiate or "one shot" state outside of the normal data table) and also the name of the file containing the soft logic program.

Example:

```
PLCCompiler = PLCCompile.PLCCompiler(DLCkInstructions.InstrDefList,
                DLCkDataTable.InstrDataTable, 'demoplcprog.txt')
```

Parameter	Type	Description
DLCkInstructions.InstrDefList	List	The list of instructions defining the instruction set.
DLCkDataTable.InstrDataTable	Dictionary	An empty dictionary for storing private instruction data.
'demoplcprog.txt'	String	The name of the file containing the soft logic program. This must be a valid file name.

The compiler should be initialised once only when your application starts up. Once it is initialised, it may be called as many times as desired until your application exits.

Loading the PLC Source Program into Memory

The compiler has a function called "ReadInFile" which reads in the source file from disk. The file will be the one specified when the compiler was initialised (see above).

Example:

```
PLCCompiler.ReadInFile()
```

If the file is not present or cannot otherwise be read, an exception will be raised which you must deal with.

Example:

```
try:
        PLCCompiler.ReadInFile()
except:
        pass # Do something about the error.
```

Compiling the PLC Source Program

Once the PLC source file has been loaded into memory, it must be "compiled". This involves checking the PLC program for obvious errors and then converting it into a form which allows it to be executed.

Example:

```
plcprogram, instrcount, CompileErrors = PLCCompiler.CompileProgram()
```

The "CompileProgram" function takes no parameters, but it does return three values. These are:

Return Value	Type	Description
plcprogram	Code Object	The compiled PLC program. This will be passed to the interpreter to execute.
instrcount	Integer	Indicates the number of instructions compiled.
CompileErrors	Boolean	Indicates whether and errors were encountered. If this is False, then no errors were found. If it is True, then at least one PLC program error was found, and the compiled program is not valid.

If there were any compile errors, we can ask for a description. The error messages are contained in a list.

Example:

```
# Get any compiler error messages.
CompileErrMsgs = PLCCompiler.GetCompileErrors()
for i in CompileErrMsgs:
        print(i)
```

This will produce an output something like the following:

Example:

```
Error in line 12 for STR CTD100 - One or more parameters is of an incorrect type.
```

Initialising the Interpreter

The interpreter must be initialised with the PLC program as well as the data tables and various libraries.

Example:

```
MainInterp = PLCInterp.PLCInterp(plcprogram,
                DLCkDataTable.BoolDataTable, DLCkDataTable.WordDataTable,
                DLCkDataTable.InstrDataTable,
                DLCkLibs.TableOperations, DLCkDataTable.Accumulator,
                DLCkLibs.BinMathLib, DLCkLibs.FloatMathLib, DLCkLibs.BCDMathLib,
                DLCkLibs.WordConversions, DLCkLibs.CounterTimers, DLCkLibs.SystemScan)
```

The parameters in the above example are :

Parameter	Type	Description
plcprogram	Code object	The compiled PLC program from the compiler.
DLCkDataTable.BoolDataTable	Dictionary	The part of the data table used to store boolean data table values.
DLCkDataTable.WordDataTable	Dictionary	The part of the data table used to store word data table values.
DLCkDataTable.InstrDataTable	Dictionary	The part of the data table used to store private data table values.
DLCkLibs.TableOperations	Library	This is a library used to provide libraries for instructions which operate on multiple registers.
DLCkDataTable.Accumulator	Library	A library which can provide accumulator and stack functions.
DLCkLibs.BinMathLib	Library	The binary math library. This provides all the math functions.
DLCkLibs.FloatMathLib	Library	The floating point math library.
DLCkLibs.BCDMathLib	Library	The BCD math library.
DLCkLibs.WordConversions	Library	A library which provides miscellaneous word conversion functions.
DLCkLibs.CounterTimers	Library	The library which provides counter and timer functions.
DLCkLibs.SystemScan	Library	A library which provides system scan overhead functions. These are things which must be updated between each scan.

MBLogicEngine is intended as a general purpose soft logic system that can be extended to provide different "personalities". Some of the libraries are intended for capabilities which are not used in the "Ck" personality which is described here.

When to Compile the Program and Initialise the Interpreter

The PLC program must be loaded, compiled, and the interpreter initialised at least once before the PLC program is executed. In addition, the PLC program may also be reloaded, recompiled, and reinitialised at any time. Re-initialising the interpreter does not affect the data table contents.

Executing the Soft Logic Program:

Once the PLC program has been compiled and the interpreter initialised with it, the PLC program may be executed.

Updating the Data Table

The application must read and write the data table in order for MBLogicEngine to receive updated input information or to write the output information to the actuators. The interpreter provides several functions to perform this. The functions to perform this are:

Function	Parameters	Return Value	Description

GetBoolData	List	Dictionary	This is used to read the boolean data table. It takes a list of strings. Each list element represents a boolean data table address label. It returns a dictionary with the list elements as dictionary keys, and the data table data as values.
SetBoolData	Dictionary	N/A	This is used to write to the boolean dat table. It takes a dictionary. The dictionary keys must be data table address labels, and the dictionary values are to be the updated data table values.
GetWordData	List	Dictionary	This is used to read the word data table. It takes a list of strings. Each list element represents a word data table address label. It returns a dictionary with the list elements as dictionary keys, and the data table data as values.
SetWordData	Dictionary	N/A	This is used to write to the word data table. It takes a dictionary. The dictionary keys must be data table address labels, and the dictionary values are to be the updated data table values.
GetInstrDataTable	N/A	Dictionary	This returns the entire instruction data table as a dictionary. The instruction data table is used to hold private instruction data and there is normally no reason to use this function.
SetInstrDataTable	Dictionary	N/A	This takes a dictionary as a parameter and replaces the entire instruction data table. There is normally no reason to use this function.

Example:

```
// Set the boolean data table.
MainInterp.SetBoolData({'X17' : True, 'X277' : False})
// Read the boolean data table.
BData = MainInterp.GetBoolData(['Y1', 'Y367', 'C50', 'CT100'])
```

With GetBoolData, SetBoolData, GetWordData, and SetWordData, there is no need to request or write to data table addresses which are not used.

Special Scan Overhead Functions

Some additional functions may need to be executed as part of the scan cycle but are not part of the interpreter. These must be called each cycle to update things such as timers and system data.

Calling the Interpreter

When the interpreter function "MainLoop" is called, the PLC program is executed for one complete logic scan. This function takes no parameters. Some run-time errors may cause an exception which must be handled by the application program.

Example:

```
MainInterp.MainLoop()
```

Reading the Exit Code

After the interpreter returns, a function ("GetExitCode") may be called to read exit and diagnostic information. The following information is returned:

Return Value	Type	Description
ExitCode	String	This is a text string indicating the reason the PLC program exited. The exit code 'normal_end_requested' means the program has exited normally.
ExitSubr	String	This is the name of the subroutine which was active when the program exited.
ExitRung	Integer	This is the rung number which was active when the program exited.

Example:

```
ExitCode, ExitSubr, ExitRung = MainInterp.GetExitCode()
```

Exit Codes

If no run-time errors were encountered, the program should exit with an exit code of "normal_end_requested". This exit code is set by the IL instructions that normally end the program scan. Any other exit code indicates that a run time error was encountered.

An exit code of "unexpected_end" indicates that the end of the program was encountered unexpectedly. The usual cause for this is forgetting to include the IL instruction that ends the program scan. If an "end" instruction is not encountered, the program scan will "fall off the end" of the program. There is usually no adverse consequence to this other than the expected exit code not being set.

An exit code of run "exception_error" indicates that an undefined run time error was encountered. This is always an error that terminates the scan immediately. This error should not be encountered under any normal circumstances and always indicates a serious error which prevents the program from executing correctly.

Updating the Program While Running

The soft logic program can be updated "on the fly" without restarting the whole system. This is achieved by calling "SetPLCProgram" in the interpreter. This replaces the current soft logic program with a new one.

Example:

```
PLCInterp.PLCInterp.SetPLCProgram(newplcprogram)
```

This replaces the current soft logic program with the new one. The new program will execute on the next scan. Since the interpreter itself has not been re-initialised, the data table contents will not be affected by this. *However*, any one-shot states will be reset to their defaults, which may cause an unexpected change in operation of the machine (this is because one shot states are part of the instructions themselves). Aside

from this side effect however, reloading a soft logic program effectively amounts to on line programming because the soft logic system does not have to stop executing.

Reloading a program is accomplished as follows:

- Load and compile the new program (see above). This does not affect the running state of the current program.
- Check for any compile errors in the new program.
- Use "SetPLCProgram" to set the currently running program to the new one.
- The new program will now be executing using the current data table contents.

Complete Example:

The following shows a simplified example. This examples omits exception handling.

Initialisation

The first example shows the code used to initialise and compile the PLC program.:

```
############################################################

from mbsoftlogicck import PLCInterp
from mbsoftlogicck import PLCCompile
from mbsoftlogicck import DLCkInstructions
from mbsoftlogicck import DLCkDataTable
from mbsoftlogicck import DLCkLibs

############################################################

# Compiler for PLC program.
PLCCompiler = PLCCompile.PLCCompiler(DLCkInstructions.InstrDefList,
        DLCkDataTable.InstrDataTable, 'demoplcprog.txt')

# Read in the PLC program.
PLCCompiler.ReadInFile()

# Compile the PLC program.
plcprogram, instrcount, CompileErrors = PLCCompiler.CompileProgram()

# Get any compiler error messages.
CompileErrMsgs = PLCCompiler.GetCompileErrors()

for i in CompileErrMsgs:
        print(i)

# Initialise the interpreter with the PLC program and data table.
if (not CompileErrors):
        MainInterp = PLCInterp.PLCInterp(plcprogram,
                DLCkDataTable.BoolDataTable, DLCkDataTable.WordDataTable,
                DLCkDataTable.InstrDataTable,
                DLCkLibs.TableOperations, DLCkDataTable.Accumulator,
                DLCkLibs.BinMathLib, DLCkLibs.FloatMathLib, DLCkLibs.BCDMathLib,
                DLCkLibs.WordConversions, DLCkLibs.CounterTimers, DLCkLibs.SystemScan)
```

Cyclic Scan Code

The second half of the example below shows code which must be called each scan.:

```
# Update the data table with new inputs. In a real application the values
# would be variables passed in from the main application. A similar function
```

```
# is available for the word data table.
MainInterp.SetBoolData({'X17' : True, 'X277' : False})

# This causes the PLC program to be executed once.
MainInterp.MainLoop()

# This is gets the updated data table elements to be passed to the rest of
# the application.
BData = MainInterp.GetBoolData(['Y1', 'Y367', 'C50', 'CT100'])

# Get the program exit code to see if the PLC program exited normally, or
# if there was an error.
ExitCode, ExitSubr, ExitRung = MainInterp.GetExitCode()
```

Developer Documentation - Description of Files

Overview:

This section briefly describes some of the internals of MBLogicEngine-Ck. This may be of interest to developers who wish to modify or extend the library. This would normally be of no interest to anyone who wished to simply use the library as is.

Standard Files:

The following are not specific to any particular soft logic architecture.

File	Provides
__init__.py	Module intialisation (this is a standard Python feature).
PLCCompile.py	This is the standard IL compiler. It compiles IL (instruction list) code to executable code.
PLCInstStdLib.py	This provide standard definitions for common instructions, including many boolean logic functions.
PLCInterp.py	This is the standard interpreter which is used to execute the compiled PLC program.
PLCLadder.py	This provides functions for formatting IL instructions as SVG vector graphics for displaying ladder diagrams on a web page. The use of this is not documented here.
PLCStdMem.py	This provides standard memory functions which are used to help generate data tables.

Personality Specific Files

The following are specific to the "Ck" soft logic architecture.

File	Provides
DLCkAddrValidate.py	This provides address validation functions used to verify instructions and parameter addresses for the compiler. The functions in this module act as extensions to the standard compiler to adapt it to each instruction.

DLCkCounterTimer.py	This implements counter and timer instructions as well as updating system flags.
DLCkDataTable.py	This implements the data table.
DLCkInstrLib.py	This provide definitions for any instructions not provided by the standard library.
DLCkInstructions.py	This contains a large data structure which is used to define the characteristics of each instruction.
DLCkLibs.py	This is used to create instances of all libraries used.
DLCkMath.py	This implements all the math functions. This includes compiling the equations and other functions that are normally performed by DLCkAddrValidate for other instructions.
DLCkTableInstr.py	This provides libraries which are used to implement multi-register and multi-bit instructions.
DLCkTemplates.py	This provides template functions which allow external modules to automatically generate compatible blocks of IL. The use of this is not documented here.

"Standard" Versus Personality Specific Features

MBLogicEngine is intended to be extensible to allow other PLC architectures (data table addressing and instruction sets) to be created. The files beginning with "PLC" provide generic functionality which is not tied to any particular architecture. The files beginning with "DLCk" implement the "Ck" specific functions.

Most basic boolean logic (AND, OR, OUT, etc.) functions are provided by "PLCInstStdLib.py" as these are more or less identical in most PLCs. A simple custom soft logic system which just implements basic boolean logic can be created relatively easily. This can include any type of addressing (e.g. IEC style "I", "Q", "M", etc.) as the boolean logic instructions themselves do not care about the form of the address. Counter, timer, and word (memory copying and searching, as well as math) operations are however much more involved and tend to be very architecture specific.

General System Design

The overall system works as follows:

Compiling

- "PLCCompile" is used to read in the text file containing the soft logic program under user control.
- "PLCCompile" then looks up the instruction in a data structure contained in "DLCkInstructions".
- If the instruction name is found, then "PLCCompile" calls the associated "validator" function contained in "DLCkAddrValidate". The validator function analyses the instruction and its parameters to see what instruction it is, and whether its parameters are correct. Validators can be cascaded so that if there are several different instructions which have the same name but are distinguished by different parameters, then the validators for each can be tried in turn.
- If the instruction and its parameters pass validation, "PLCCompile" the looks up the "implementation" of that instruction in either "PLCInstStdLib", "DLCkInstrLib", or elsewhere (the location for each instruction definition is defined in the data structure in "DLCkInstructions" which defines the complete instruction set). The implementation for each instruction consists of the code (including library calls) required to actually implement the instruction while the system is running.
- When all the instructions have successfully gone through the initial compile step, the list of subroutine calls is analysed to see if all the subroutines which were called are present.
- If everything has passed, then the program goes through a final compilation step to turn it into a program which the interpreter can execute.

Executing

At run time:

- The soft logic program is executed by calling the "MainLoop" function in "PLCInterp". This calls the scan update functions ("DLCkLibs.SystemScan" in the "Ck" version), resets the subroutine call stack, and sets the exit code to "unexpected_end".
- The interpreter then calls the soft logic program *once*. The soft logic program will execute until its end is reach, and will normally raise an exception to exit. This exception is not normally an error, it is simply a means for the soft logic program to provoke an exit back to the interpreter at any arbitrary point. The actual exit code is defined by the instruction which raised it. "PLCInstStdLib" defines several, including "normal_end_requested", which is raised by the "END" instruction. However, it is possible for additional arbitrary exit codes to be created by creating additional instructions create exceptions.
- Each rung and each subroutine stores its current rung number or subroutine name as it executes. These can be examined (along with the exit code) by calling "GetExitCode". This can be used to tell where the soft logic program exited, due to either a normal exit or a fatal error.

Data Table

The soft logic data table is implemented as a pair of dictionaries. One dictionary is used for boolean instructions, and the other is used for work instructions.

The data table address are used as keys to the dictionaries. For example, 'X1' would be the key to access the 'X1' address in the boolean dictionary. These keys are referred to as address "labels". Because the address labels are only directly examined by the validators (and possibly in a few other minor areas), most instructions are agnostic as to what the address actually looks like once it has been validated. This means for example that "X9" and "I1.0" would work equally well as boolean addresses, *provided* the validator passed the address as valid while it was being compiled.

Boolean and word addresses are implemented separately in order to optimise them for the most common usage. The alternative would have been to pack the boolean addresses into bytes or words. However, since boolean addresses are rarely accessed as bytes or words this would have made the system slower and more complex for the most common cases while providing very little benefit to the less common ones. The "Ck" library has PACK and UNPACK instructions to transfer words between boolean and word addresses, so this does not raise any actual implementation problems.

Updating the data table is achieved by calling functions which read and write the data table dictionaries. These are documented above.

In addition to the normal data table, an "instruction data table" is present. The instruction data table is a private data structure which each instruction can access to store private data which that instruction needs to execute. This is typically used by "one-shot" instructions. In many PLCs (including the "Ck" soft logic library) one shot data is not stored in the main data table. Instead each instruction is expected to allocate a hidden memory address which is used to store the one shot state. The instruction data table provides a separate address for each instruction instance that needs it. That is, each instance of "ANDPD" (for example) would get its own address (rather than one address which is shared by all instances of "ANDPD"). The addresses are automatically assigned by the compiler as required. If an instruction does not require an instruction data table address, then no address is allocated. There is normally no reason to read or modify the instruction data table at run time.

Additional System Libraries

The system provides for the presense of a number of system libraries which are not used in the "Ck" personality (they were used in a previous obsolete personality). These are the "FloatMathLib" (floating point math is handled by "BinMathLib"), "BCDMathLib" (there is no BCD math in the "Ck" personality), "WordConversions" (conversions are handled by the COPY instruction in "Ck") and "Accumulator" there is no accumulator in "Ck"). In the "Ck" personality these are set to "None".

Despite the names, the function of these libraries is not in any way predefined. They can be used to implement additional features if desired. Note however that some of these unused library interfaces may be removed in a future version of MBLogicEngine and replaced by a simpler, more generic, and more extensible interface.

END.