PLC Programming Manual

Introduction

This manual is intended for machine builders, technicians, dealers, and others who need to "look under the hood" of the PLC system.

The PLC (Programmable Logic Controller) is the portion of the control system, which deals with accessory equipment; that is, anything installed on the machine beyond the servo motors themselves. Spindle control, coolant control, limit switches, fault signals, pushbuttons, indicator lights, etc. are all controlled through the PLC.

The PLC system has two possible configurations. The first configuration consists of a hardware I/O board which provides the electrical interface to all the switches and relays (RTK3, PLCIO2, RTH2, PLC15/15, ect.), the jog panel with its keys and LEDs, and a software logic program which is executed on the CPU10 controller board (CNC10.PLC Program).

In the first configuration (see figure 1) the PLC program, which is executed on the CPU10, is contained in the file CNC10.PLC. The control software sends CNC10.PLC to the CPU10 on startup. Once running on the CPU10, the logic program receives inputs from the PLC I/O board, from the jog panel, and from CNC programs. It sends outputs back to all three places: for example, turning on output relays on the I/O board; turning on LEDs on the jog panel; and sending signals such as Feed Hold to the CNC processor.

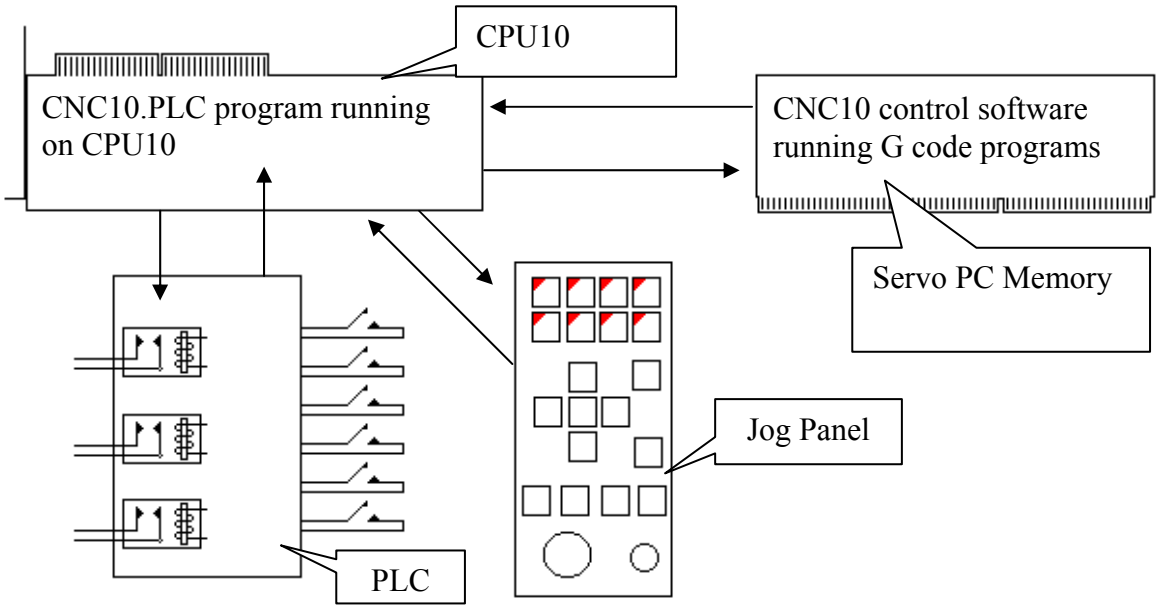


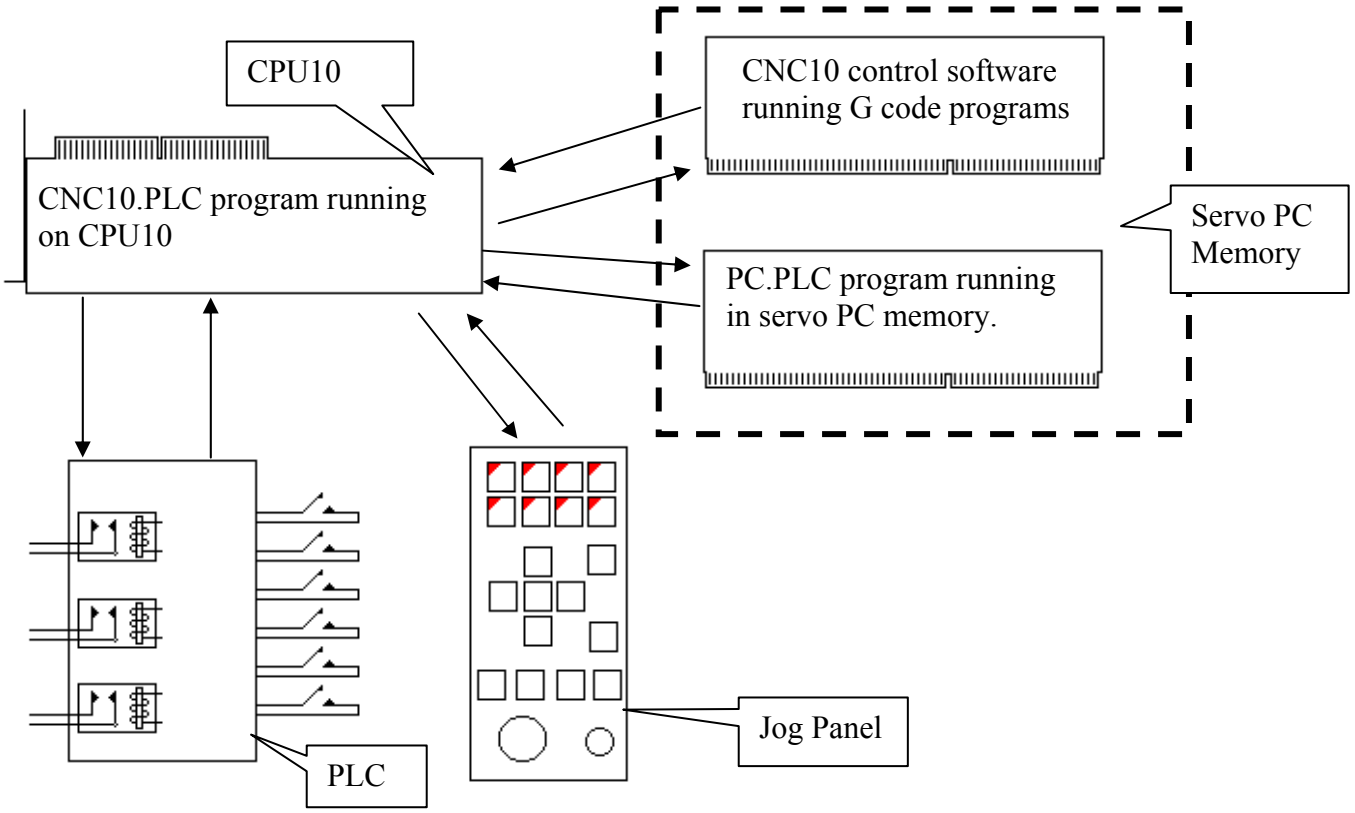
Figure1: PLC I/O Configuration 1

The first PLC configuration, using the CNC10.PLC program only, is limited in size and functionality and cannot control complex machine hardware such as a tool changer, which may require timers, counters, ect. For machines that need more PLC program functionality a second PLC configuration is used. The second configuration consists of all elements of the first configuration plus a second PLC program (PC.PLC) running in the servo PC memory.

CNC10.PLC program is still running on the CPU10 and is used in conjunction with the PC.PLC. The reason we need two PLC programs running is the new PC.PLC program doesn't have access to all of the I/O so the CNC10.PLC program needs to be there to control those I/O bits. A minimal CNC10.PLC program can be made by echoing memory bits to the outputs you need to control. Then you can control those memory bits in the PC.PLC program thus making it possible to control virtually everything needed in the PC PLC program.

Figure 2: PLC I/O Configuration 2

While the PLC program has access to many features of the jog panel, it has no access to the



jogging controls themselves. The axis jog buttons, Fast/Slow and Inc/Cont buttons, the jog increment buttons, and the MPG handwheel are all directly handled by code on the CPU7.

The first section of the PLC manual will explain programming for the CNC10.PLC program, then move into more complex issues and the PC.PLC or extended PLC programming.

Language

The PLC programming language is a simple statement-based logic language. At first it may seem far removed from the ladder logic diagrams familiar from other PLC systems.

However, each assignment statement in a PLC program is equivalent to one rung of a PLC ladder. One or more inputs are logically combined to produce one output.

The PLC supports four logical operators: AND, OR, XOR, and NOT. They may be abbreviated using the symbols & (AND), | (OR), and / (NOT). There is no abbreviation for XOR. Parentheses may be used for logical grouping. In the absence of parentheses, the operators follow normal mathematical precedence: NOT, AND, XOR, OR.

The "=" sign is used to store a logical result (an input of combination of inputs) in an output location.
These operators are used to manipulate 240 data bits: 80 inputs, 80 outputs, and 80 memory locations. Many of the inputs and outputs are permanently mapped to particular hardware devices. The memory locations are not mapped to any hardware, but some are reserved for special purposes.

The data bits are given default names such as INP1, INP2, INP80, OUT1, OUT41, MEM27, etc..

Typically the first part of a PLC program file assigns alternate names to these default tokens. For example, the line

```
Emergency_stop  IS INP11
```

tells the PLC program compiler that the name Emergency_stop is equivalent to INP11. You can then use the more intuitive name for the remainder of the program.

It is important to distinguish between "IS", which assigns alternate names, and "=", which actually stores computed bit values. A line using "IS" doesn't actually do anything in the PLC program; it simply provides information to the compiler about the names you will be using. A line using "=" has a real effect each time the program runs.

Let us look at a simple latched input structure: suppose that we have a relay connected to OUT1 which controls a work light. We want the Aux1 key on the jog panel to turn the light on, and the Aux2 key to turn it off. In the definitions sections of the program we would have:

```
Aux_1_key IS INP49
Aux_2_key IS INP50
```

and

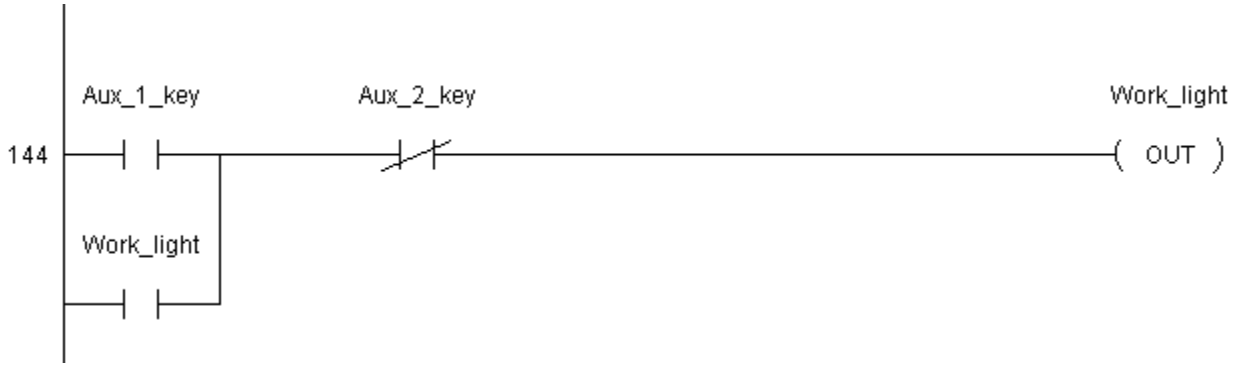
```
Work_light IS OUT1
```

In the statement section which follows we would have

```
Work_light = ( Work_light OR Aux_1_key ) AND / Aux_2_key
```

This statement says that the relay will be on if it was already on or if the Aux1 key is pressed, as long as the Aux2 key is not pressed.

In traditional ladder logic this would appear as follows:



The PLC compiler is not case sensitive. However, it is a good idea to write your PLC programs using consistent capitalization. This will make them easier to read and easier to search.

The parser is quite basic. Tokens (names, operators, and parentheses) are separated by white space (spaces, tabs, or line breaks). Names may contain almost any character. However, your programs will be more readable if you stick to alphanumeric characters and the underscore character. For example, the compiler will permit a definition like:

```
High/Low_range IS INP64
```

but the slash character (/) in the name is distracting and confusing.

All functional statements are assignments, using the '=' sign. The destination bit (e.g. an output coil) appears on the left side of the '=' sign, in the first column of the program source file. If a statement is too long to conveniently fit on one line of the program file, you can break it across multiple lines by indenting each subsequent line from the first column (e.g. by starting the line with spaces or tabs):

```
Spin_stop          = Spindle_stop_key OR
                            Stop OR Limit_tripped OR
                      ( Auto_spin_mode AND / AutoStart )
```

Comments may be included in the program source file, set off by the ';' character. The PLC compiler will ignore any text which follows the ';'. Comments are useful for explaining the intent of your PLC logic and for separating different sections of the program for readability. Comments may be on lines of their own, or may be added to the end of PLC program lines.

Any bit -- whether physically an input, an output, or a memory location -- may be assigned with the "=" operator. You are not limited to just writing to outputs.

PLC program source files are translated into the control's internal format using the PLC compiler utility, PLCCOMP. PLCCOMP and related tools are located in the C:\PLC directory on the control's hard drive.
Typically all PLC programming is done at the MS-DOS prompt. The control software, probably wisely, does not provide any readily accessible, user friendly screen to allow you to change the PLC programming.

Tutorial Example

Suppose you have an M400 control with an inverter drive for the spindle, and a spindle air brake. The default PLC logic will apply the air brake the instant the spindle-run relay is switched off, in spite of the fact that the inverter needs several seconds to decelerate the spindle. This leads to a situation where the inverter is driving the spindle motor forward against the brake, in order to maintain the programmed deceleration ramp.

One way to resolve this problem is to reprogram the PLC so that the brake is disabled whenever the spindle is turned on. The operator would then have to press the Brake key on the jog panel after the spindle has stopped if he wants to apply the brake.

<Ctrl-Alt-X>
C:\CNC10> **CD \PLC**
C:\PLC> **COPY M400.SRC K6789.SRC**
C:\PLC> **EDIT K6789.SRC**

At the beginning of the file we find a block of comments similar to the following:

```
; * * * * * * * * * * * * * * * * * * * * * * * * * * * *
; * File:    m400.src
 .
 .
 .
; * Modifications:
; * 09/10/97 KSD Added Vector drive support by echoing
; *              SpindleRelay and CCW_relay to
; *              MEM78        and MEM79
; * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

Although it is not strictly necessary, it is a very good idea to update these comments to reflect the changes we are making:

```
; * * * * * * * * * * * * * * * * * * * * * * * * * * * *
; * File:    K6789.src
 .
 .
 .
; * Modifications:
; * 09/10/97 KSD Added Vector drive support by echoing
; *              SpindleRelay and CCW_relay to
; *              MEM78        and MEM79
; * 07/25/01 MBL Revised brake logic to disable brake
```

```
; *              mode whenever spindle runs.
; * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

Now page down (or search for “Brake”) to find the section of the program which controls the brake mode and output:

```
;
; Select Brake Mode
;
Brake_key_hit       = Brake_key AND / Last_brake_key
Last_brake_key      = Brake_key
      Brake_mode          = ( Brake_mode XOR Brake_key_hit )
                              OR / Already_run
;
; Turn on the brake if spindle not running and in auto mode
;
Brake               = Brake_mode AND / Spindlerelay
Spindle_brake_LED   = Brake_mode
```

When Brake_mode is 1, the brake is in “Auto” mode, and is switched on immediately whenever the spindle is switched off. When Brake_mode is 0, the brake is left off.

We can change this so that Brake_mode is set to zero any time the spindle is running. In this case it is no longer necessary to check for spindle running before applying the Brake.

```
;
; Select Brake Mode
;
Brake_key_hit       = Brake_key AND / Last_brake_key
Last_brake_key      = Brake_key
Brake_mode          = ( ( Brake_mode XOR Brake_key_hit
                            ) OR / Already_run )
                            AND / SpindleRelay
;
; Turn on the brake if spindle not running and in auto mode
;
Brake               = Brake_mode
Spindle_brake_LED   = Brake_mode
```

Note the parentheses in the Brake_mode expression. They are required because AND has a higher precedence than OR. Note also the spaces around the parentheses. The PLC compiler needs them to separate tokens. If we wrote the Brake_mode expression as:

```
      Brake_mode          = ((Brake_mode XOR Brake_key_hit)
                          OR / Already_run)     AND
                          / SpindleRelay
```

the PLC compiler would give us error messages, saying that it did not recognize “((Brake_mode”, “Brake_key_hit)” and “Already_run)”. In other words, the compiler would think that the parentheses were parts of the names, and therefore that the names were new and different ones from what we had defined earlier in our program. So always put spaces around parentheses.

File/Save
File/Exit
C:\PLC> **PLCCOMP /I K6789**

```
PLCCOMP v. 7.17 - PLC compiler

Compilation successful
```

C:\PLC> **CNC10M4**

The /I (install) switch tells PLCCOMP to copy the compiled PLC program over to the CNC10 directory as CNC10.PLC.

If we had made any mistakes in our PLC source file, PLCCOMP would report the errors, and would not produce any compiled program. For example, if we had left out the spaces around the parentheses in the example above, we would see something like this:

```
PLCCOMP v. 7.17 - PLC compiler

Unknown item: ((BRAKE_MODE on line 238

Unknown item: BRAKE_KEY_HIT) on line 238

Unknown item: ALREADY_RUN) on line 238
Missing operator on line 239
```

PLC Hardware

We have produced a wide assortment of PLC input/output boards over the years. Programming is done in the same way regardless of the board installed; the hardware merely determines which inputs and outputs are physically available, and sometimes how you can use them. You must set the "PLC Type" selection on the CNC10 Control Configuration screen to match the PLC hardware, see chapter 1 for PLC hardware descriptions.

Locating Source for Current PLC Program

If you are working on an older control, or one which has seen custom work, you may not be certain which PLC program is actually in use. Is it a "stock" PLC program, or has someone modified it? Is it based on an earlier version of a standard PLC program?

Before doing any custom work of your own, you want to be sure you have the source file for the PLC program which has been running on the control. Otherwise you risk breaking features that are important to the customer.

Since approximately software version 5.10, the compiled CNC10.PLC file has contained comments indicating the source file from which it was compiled. The first step, then, is to look there:

<Ctrl-Alt-X>
C:\CNC10> **EDIT CNC10.PLC**

```
; PLC program compiled by PLC compiler PLCCOMP v. 5.26
; Source file: M400.SRC
; Date:        10 September 1997
; Time:        09:01:23
;
9B 20 FC 21 FC F2 A2 FC F1 FD
A7 01 FC 00 FC F2 03 FC F2 02 FC F2 05 FC F2 04 FC F2 FD
 .
 .
 .
```

Note that the date and time listed show when the file was compiled, not the date and time when the source file was last modified. More recent versions (since approximately software version 6.03) list the source file date and time as well as the compilation date and time:

```
; PLC program compiled by PLC compiler PLCCOMP v. 6.03
; Source file: F:\SOFTWARE\PLC\CUSTOM\M400B.SRC
; File Date:  9-20-99 11:30a
; Compiled:   7-14-01  3:40p
;
A5 01 FC 00 FC F2 03 FC F2 02 FC F2 05 FC F2 04 FC F2 FD
A4 20 FC 21 FC F2 FD
 .
 .
 .
```

File/Exit

If we did not find a useful comment at the beginning of the CNC10.PLC file, the next step is to compare its file time, date, and size to a compiled PLC file we may find in the PLC directory.

C:\CNC10> **DIR CNC10.PLC**

```
Directory of C:\CNC10

CNC10     PLC         1,675  06-09-94  5:40p
         1 file(s)          1,675 bytes
```

C:\CNC10> **CD \PLC**

C:\PLC> **DIR *.??C**

```
Directory of C:\PLC

M40-N3B  SRC        12,854  06-09-94  5:39p
M40-N3B  PLC         1,675  06-09-94  5:40p
         2 file(s)         14,529 bytes
```

Here we see that there is a file in the PLC directory, M40-N3B.PLC, which appears to be the same as the running CNC10.PLC file. We could make certain using the DOS file compare utility FC:

C:\PLC> **FC M40-N3B.PLC \CNC10\CNC10.PLC**

```
Comparing files M40-N3B.PLC and \CNC10\CNC10.PLC
FC: no differences encountered
```

Furthermore, we see we have a matching M40-N3B.SRC file which was last modified shortly before the PLC file was compiled. We can safely assume it is the source of the active PLC program.

If there are source files in the PLC directory, but no compiled .PLC files, you can always compile each source file (without installing!), then use FC to compare the resulting .PLC files to the CNC10.PLC file in the CNC10 directory. If there are no differences outside of the header comments, then you have found your source file.

If all of those methods fail, then the current PLC source is not stored on the control. Your best bet at that point is to catalog the jog panel type, PLC board type, I/O connections, and control features, then choose the most appropriate PLC program source you can find. It is a good idea in this case to preserve the old CNC10.PLC file for reference.

PLC Memory Map

Inputs	
1 - 15	Physical inputs on PLC board (0 = closed, 1 = open)
16	Integrity signal (1 = okay)
17 - 32	Inputs from PLCIO2, RTK3 or Koyo PLC (otherwise unused)
33 - 48	M function requests (M94 and M95)
49 - 58	AUX keys (normally 0; 1 when pressed)
59 - 62	PIC inputs
63	Mid range
64	High/low range
65	CNC program running
66 -	Jog panel keys

75	
77	Pause (Feed Hold) (0 = pause, 1 = run)
78	Block Mode key
79	Rapid Override key
80	Disable spindle override (force 100%)

Outputs	
1 - 15	Relays on PLC board (0 = off, 1 = on)
16	Integrity signal (1 = fault)
17 - 24	Analog spindle speed (8 bits, for use on RTK2)
25 - 28	Extended resolution analog spindle speed (12 bits total, for use by Koyo)
29 - 40	Used with PLCIO2, RTK3 or Koyo PLC
41 - 48	BCD tool number
49 - 58	AUX LEDs
59 - 62	PIC outputs
63	Low lube fault (1 = fault)
64	Servo drive fault (1 = fault)
65	Spindle fault (1 = fault)
66 - 73	Jog panel LEDs (1 = on)
75	Stop (1 = any fault)
76	PLC operation indicator (1 = busy)
78	Single block mode (1 = on)
79	Rapid override (1 = enabled)

Memory	
1 - 40	General purpose
41	Messages, or general purpose

- 48	
49	XPLC in use if set
50 - 72	Messages, or general purpose
73 - 77	Reserved
78	Copy of Spindle_relay, for PC spindle control
79	Copy of CCW_relay, for PC spindle control
80	M function macro indicator (v5.27 and earlier)

What does a 1 or 0 mean?

For outputs, the logic is quite simple: if the PLC output bit is 0, then the output is off (relay open, LED off, etc.); if the output bit is 1, then the output is on (relay closed, LED lit, etc.).

For inputs, the logic varies:

For physical inputs (INP1 through INP15), 0 indicates that the switch is closed (the input point is being pulled to ground); 1 indicates that the switch is open (the input point is maintained at 5V by the internal pull-up).

For inputs sent from a Koyo PLC (INP4 through INP15 and INP17 through INP32), the 0 or 1 state is as sent by the Koyo PLC program.

For M function request bits (INP33 through INP48), M94 sets the bit to 1 and M95 sets the bit to 0.

For jog panel keys, 0 indicates that the key is not pressed; 1 indicates that the key is currently pressed.

When you name the inputs in your PLC program, keep the normal states in mind. For example, if you connect a normally-closed inverter fault signal to INP14, you might want to name it "Spindle_fault", because it will be 0 (closed) if all is well, and 1 (open) in the event of a fault. If on the other hand you are using a normally-open fault signal, then you should name it something like "Spindle_okay", since 0 (closed) indicates a problem and 1 (open) indicates that all is well.

This naming guideline can be a little more awkward in some cases: notably for "start" buttons which are wired to PLC input points. Fail-safe design requires that start buttons be normally open, but that means the inputs will show up as 1 (open) when idle, and 0 (closed) when the button is pressed. Strictly speaking, you should name the input something like "Remote_start_not_pressed". In such cases, though, you might conclude that a shorter name, like "Remote_start", is more readable even if it is not literally correct.

Some Special Locations

INP63 - Mid_range
INP64 - High_low_range

These inputs define the current spindle gear range. A two-speed machine need only use INP64 (0 = high range, 1 = low range). A three or four-speed machine uses both inputs, as follows:

Range	INP63	INP64
Low	0	1
Low-Mid	1	1

High-Mid	1	0
High	0	0

Both the CPU7 and CNC10 use these inputs to compute motor speed for a desired spindle RPM, and to compute display RPM from motor speed. Typically these inputs are copied from physical input switches: e.g. a range detect switch under the back gear lever which is closed (0) in high range and open (1) in low range. These bits could also be set by internal PLC logic which is controlling an automatic gear changer.

INP65 - CNC_program_running

This input tells the PLC program when a job is in progress. It will be set (1) whenever a CNC program is running. It will also be set in MDI mode and during homing, digitizing, probing, and other automatic machine movement. It will not be set if the operator is merely jogging an axis.

The standard PLC programs use INP65 to run the lube pump during jobs, and to cancel all automatic M functions when the job ends or is canceled.

INP77 - Pause

This bit reflects the FEED HOLD state. It would be more appropriately named "Run", since a value of 0 means the machine is paused, and a value of 1 means the machine is allowed to run.

Standard PLC programs do not use this bit, but custom applications can use it to implement additional FEED HOLD buttons; to activate FEED HOLD when a guard or cover is opened; or to allow or disallow certain operations during FEED HOLD.

OUT75 - Stop

This bit indicates a fault condition. When it is set, the CPU7 will immediately cancel any running CNC job, and will release power to the servo motors. The standard PLC programs set this bit in response to the five standard faults: servo drive fault; spindle fault; PLC failure; low lube; and emergency stop. Note that a low lube fault is held off as long as INP65 (CNC_program_running) is set, allowing the current job to finish.

Custom applications may set the Stop bit as needed to force a fault. The on-screen message will be determined by four reserved fault bits:

OUT16 - PLC_fault_out
OUT63 - Lube_fault_out
OUT64 - Drive_fault_out
OUT65 - Spindle_fault_out

If one of these bits is set, CNC10 will display either the matching custom message (see Custom On-Screen Messages below) or the default message for that fault if no custom message is defined.

If none of the four known faults is indicated, then the control will assume it is just E-stop, and display “Emergency stop detected”.

If your application requires a special fault condition, but does not require all four standard faults (e.g. there is no auto lube system) then you can redefine one of the standard faults (e.g. set OUT63 in the event of trouble, and define a custom message to go with it).

If you need a special fault condition, and still need to use all four standard faults, then you will have to live with the display of “Emergency stop detected” when your fault occurs. You can still get a custom message to appear in addition, by setting one of MEM41-MEM72 and defining a message for that bit.

OUT76 - PLC_op_signal

This bit allows the PLC program to tell the CNC processor that some PLC-related operation is not yet complete, and therefore that the CNC job should be paused awaiting completion.

The standard PLC programs set this bit whenever the CNC program attempts to start the spindle in automatic mode (using M3 or M4), but the operator has switched to manual spindle control (using the Spindle AUTO/MAN button on the jog panel). In this case the message “Waiting for PLC operation (M3)” will be displayed. OUT76 will be cleared and the job will proceed only when the operator switches back to auto spindle control.

OUT76 is set to 1 automatically at the beginning of each PLC program scan. If there are no conditions in your PLC program which might need to set OUT76, you will still need an assignment in order to clear it back to 0.

```
PLC_operation = Zero
```

MEM41- MEM48 and MEM50- MEM72

These bits can be used to trigger custom messages in the status window. See Custom On-Screen Messages below. When not being used for that purpose, they can be used freely like any other memory bits.

MEM49

MEM49 is used by the XPLC program to communicate to the standard program that the XPLC program is being used.

Custom On-Screen Messages

There are several ways you can make custom messages appear in the CNC10 status window.

In some cases you substitute your message for a default message the control would otherwise have displayed in the course of its operation. In other cases you specify a message to be displayed “asynchronously”, without regard to what the control is currently doing.

Custom messages are stored in the file CNC10XMSG.TXT, in the CNC10 directory. Each message occupies two lines:

```
MEM41
"Unplug the probe!"
```

The first line specifies the PLC bit which triggers the message; the second line is the message itself, in double quotes.

Waits

Whenever the CNC processor is executing an M100 or M101 (waiting for any input) or M0 (waiting for INP75 - CYCLE START) you can substitute your own message by associating your message with the relevant bit. For example, the following definitions would cause appropriate messages to be displayed while executing M80 and M81 operations on a tool changer mill running the ATC6 PLC programs:

```
INP27
"Waiting for carousel to advance"
INP28
"Waiting for carousel to retract"
```

Limits

Whenever a limit switch (any switch number specified on the Machine Configuration screen) is tripped, the control displays a message such as "Y+ limit (#3) tripped". You can replace this message with your own by associating a message with the limit switch number (e.g. INP3).

Faults

As mentioned previously, you can associate messages with INP16, INP63, INP64, and INP65 to replace the default messages for PLC failure, low lube, servo drive fault, and spindle fault.

Asynchronous Messages

If you want your message to appear whenever a condition is present, regardless of what the CNC processor might be doing at the moment, you can associate it with a memory bit ranging from MEM41 to MEM72. Your message will be displayed any time the bit is set.

Common Expressions

Unconditional Output

It is often useful to set or clear a PLC bit without regard to any inputs. For example, a PLC program which does not detect any fault conditions (e.g. on a machine with no PLC I/O board installed) might clear the fault indicator with the line:

```
Stop = Zero
```

Likewise, a PLC program for a control with no Rapid Override key (e.g. M15 models 1 through 7) might enable Rapid Override mode with the line:

```
Rapid_override = / Zero
```

Many PLC programs give the name "Zero" to one of the memory bits (often MEM11). There is nothing special about this location. All memory bits are initialized to 0 at power up. If you never change one, then you can use it for unconditional outputs like this.

If you change PLC programs after power up, it is possible that the earlier program set a memory bit that the new program would like to assume is 0. You can explicitly clear or set bits with lines such as the following:

```
Zero = Zero AND / Zero        ; always cleared
One = One OR / One            ; always set
```

Latched Fault

The latch and reset expression we saw earlier is used with fault conditions. If a fault occurs, we want to catch it and report it even if the problem was only momentary. Once we have seen the error message and fixed the problem, then we can clear the fault by pressing and releasing Emergency Stop.

```
PLC_fault_out       =  / PLC_ok OR ( PLC_fault_out AND
                                     / E_stop )
```

The PLC fault output will be set if there is a problem with the PLC integrity signal on INP16. Once set, it will remain set until the problem is fixed (INP16 comes back on) and Emergency stop is pressed.

Detecting Changed Inputs and Toggling an Output

We often want a single input to switch an output either on (if it was off) or off (if it was on). We can do this easily with the XOR operator. However, we need to ensure that the output changes state only one each time the input is activated. If we wrote:

```
Work_light = Work_light XOR Aux_1_key
```

Then the light would switch rapidly on and off as long as the key was held down.

To make this work properly, we need to detect when the key is pressed and toggle the output only in that one scan of the PLC program. We can do this by storing the previous state of the key in a memory location.

```
Last_aux_1 IS MEM23
Aux_1_hit  IS MEM24
[...]
Aux_1_hit  = Aux_1_key AND / Last_aux_1
Last_aux_1 = Aux_1_key
```

```
Work_light = Work_light XOR Aux_1_hit
```

If the key is down now, and was not down in the previous scan, then the "Aux_1_hit" memory location will be set for this one scan, and the output will toggle just once.

Initializing Bits on First Scan

Another common need is to set a bit on the first scan of the PLC program, then let it change according to normal logic. We can detect the first scan by setting a memory location as the last operation in our program. If that location is set, then we know we have been through the program at least once. If it is not set, then we know this is the first scan.

At the end of the program:

```
Already_run = / Zero
```

or

```
Already_run = Already_run OR / Already_run
```

Then, earlier in the program, we can refer to this when we want to set some mode explicitly on startup. We saw this previously in the spindle brake example, where the automatic braking mode is set on startup, then toggled with the BRAKE key on the jog panel:

```
Brake_mode          = ( Brake_mode XOR Brake_key_hit ) OR / Already_run
```

Application Examples

Remote FEED HOLD

Some operators, particularly those who also use older M10 and M40 controls, find the FEED HOLD key on M15/M39/M400 jog panels too small, and hard to locate quickly.

It is fairly simple to install an external pushbutton as an auxiliary FEED HOLD button. Unlike the button on the M40, it cannot latch: it must be a momentary button, since CYCLE START is required to restart out of FEED HOLD.

Assume we have an illuminated red pushbutton with a normally-closed contact block (e.g. a Cutler-Hammer E22TB2X4B). We wire the switch contacts to INP9, and wire the light through OUT8.

In the definitions section we have:

```
Remote_pause        IS INP9    ; 0 = idle      1 = pressed
```

and

```
Feed_hold_light     IS OUT8    ; 0 = off       1 = on
```

In the program we have:

```
;
; Turn on feed hold if remote feed hold button is pressed
;
Pause = Pause AND / ( Remote_pause AND CNC_program_running ) OR
Cycle_start
Feed_hold_light = / Pause
```

Recall that INP77 (“Pause”) is 0 to pause, 1 to run.

The first statement changes Pause to 0 if the remote pause button is pressed while a job is running, and changes it back to 1 if CYCLE START is pressed. The “OR Cycle_start” clause is not necessary if we only have the built-in CYCLE START button: CPU7 will automatically cancel FEED HOLD when that button is pressed. However, if we have a remote CYCLE START, as shown below, CPU7 will not automatically recognize it as canceling FEED HOLD. In that case we have to do it explicitly as shown above.

The second statement turns on the light any time we are in FEED HOLD.

Remote CYCLE START

Many operators also like to have an additional CYCLE START button in a handheld pendant or mounted on the machine head. If you have installed a remote FEED HOLD as described above, it is particularly convenient to have a remote CYCLE START located nearby for restarting out of FEED HOLD.

Assume we have an illuminated green pushbutton with a normally-open contact block. We wire the switch contacts to INP8, and wire the light through OUT7. The light will be an in-cycle indicator, illuminated any time a job or other automatic cycle is running.

In the definitions we have:

```
  Remote_start        IS INP8    ; 0 = pressed   1 = idle
```
and
```
  In_cycle_light      IS OUT7    ; 0 = off       1 = on
```
and
```
  Last_remote_start   IS MEM30
```

Fail-safe design demands that we use a normally-open switch contact for any START button. Unfortunately this results in "reversed" logic in the PLC program. INP8 will be 1 when the button is not being pressed, and will change to 0 when the button is pressed.

The logic for combining the remote start button with the existing input for the built-in CYCLE START button is fairly complicated. I will not attempt to explain here how it is derived. Also, normally-open inputs like this one must usually be explicitly set in the first scan of the PLC program.

In the program, then, we have:

```
;
;  Set Remote_start to 1 (not pressed) on first pass through PLC
program
;
Remote_start = Remote_start OR / Already_run
;
;  Combine Start and Remote_start into Start
;
```

```
Cycle_start =  ( / Cycle_start AND / Remote_start AND /
Last_remote_start )
            OR (   Cycle_start AND   Remote_start AND /
Last_remote_start )
            OR (   Cycle_start AND / Remote_start )
Last_remote_start = / Remote_start
;
In_cycle_light = CNC_program_running
```

The light, of course, is optional.

Remote CYCLE CANCEL

Remote Cancel logic is a little simpler than Remote Start, because the result (in INP74) only needs to stay on for a single scan.

```
Rem_cancel          IS INP9   ; 0 = idle, 1 = pressed
[...]
Cycle_cancel        IS INP74  ; 0 = continue, 1 = cancel
[...]
Last_remote_cancel  IS MEM26
Last_cancel         IS MEM28
[...]
;
;  Combine local and remote cancel into cancel; clear after one cycle
;
Cycle_cancel =  ( / ( Cycle_cancel AND Last_cancel )
                  AND Remote_cancel AND / Last_remote_cancel )
                OR ( Cycle_cancel AND / Last_cancel )
;
Last_remote_cancel = Remote_cancel
Last_cancel = Cycle_cancel
```

Cycle_cancel will be set if the remote cancel button was just pressed, or if the built-in cancel button was just pressed; it will be cleared again once it has been set for one scan.

AUX key to toggle an output

The standard PLC programs which ships on M39 and M400 controls use the four Aux keys as on-off toggles. For example, Aux1 normally controls OUT6. Pressing the key switches the output from off to on, or from on to off.

The code to implement this is as follows:

```
Aux_1_out           = Aux_1_out XOR ( Aux_1_key AND / Last_aux_1 )
Aux_1_LED           = Aux_1_out
Last_aux_1          = Aux_1_key
```

The output retains its last value unless the key was just depressed. In that case the output changes state (1 becomes 0, 0 becomes 1). XOR is a convenient way to achieve this.

The LED in the Aux key indicates the current state of the output.

The last state of the key itself must be saved in a memory location so that changes can be detected.

AUX key for a momentary output

Sometimes you only want the output to remain on as long as the button is held down. Suppose you have a knee mill retrofit with the Z axis in the quill. You would like to power the knee, but cannot justify the cost of a full four-axis control package. Instead, you install a DC gear motor as a knee power feed, and use the Aux1 and Aux2 keys as knee up/down jog keys. You install a pair of relays suitable for the motor load, and wire their coils to OUT6 (knee up) and OUT7 (knee down).

In the definitions, then, you would have:

```
Knee_up_relay   IS OUT6
Knee_down_relay IS OUT7
```

In the program, you could write:

```
Knee_up_relay   = Aux_1_key
Knee_down_relay = Aux_2_key
```

Each relay would close as long as the corresponding key was held down, jogging the knee in the selected direction.

However, you would have to be sure to wire interlocks through your reversing relays. Otherwise the operator could cause a dead short by pressing both Aux keys at the same time.

A safer solution would be as follows:

```
Knee_up_relay   = Aux_1_key AND / Aux_2_key
Knee_down_relay = Aux_2_key AND / Aux_1_key
```

This has the same effect, but turns off both outputs in the event both Aux keys are pressed.

Custom M Functions

You can easily create custom M functions for a variety of applications. Any M function from M0 through M90 can be defined or redefined with an M function macro file.

Suppose you have a lathe with an air spindle for live tools. You want to use M32 to turn the air spindle on, and M33 to turn it off.

To do this, you need to find an unused bit in the range INP33 through INP48. These bits are M function requests, sent by M functions in the running CNC job to the PLC program. The first few are usually already in use for spindle and coolant control, and sometimes for a clamp or collet closer. For this example we will assume that INP38 is available.

The M94 and M95 codes control the M function request bits. M94 turns the specified request on; M95 turns the request off. With M94 and M95, the M function requests are numbered 1 through 16 (even though they are mapped to inputs 33 through 48).

Therefore, to turn on INP38 we would execute M94/6. To turn off INP38 we would execute M95/6.

We create two files in the control software directory (e.g. C:\CNC10T):

CNC10.M32:

```
; M32 - turn on air spindle
M94/6
```

CNC10.M33:

```
; M33 - turn off air spindle
M95/6
```

This is all that is needed for the control to recognize M32 and M33 as valid M functions, and to execute the command(s) in the files whenever a program calls on them.

In the PLC program we need to accept and act upon the M function request. Setting INP38 with M94 does not by itself do anything.

In the definitions we have:

```
Air_spindle IS OUT5
```

and

```
M32 IS INP38
```

In the program, we have

```
Air_spindle = M32
M32 = M32 AND CNC_program_running
```

The first line turns on the output whenever it is requested by the M functions. The second line is necessary to ensure that the output is turned off when the program ends, even if the

programmer forgot to put in a final M33 (or the operator stops the job with CYCLE CANCEL or E-stop).

Note that there is no definition for M33 in the PLC program. M33 is simply the absence of M32.

Custom M Function with AUX key toggle (manual override)

In the previous example, we assumed that the air spindle would be used only with M functions, and only during a programmed job.

Suppose we would also like to have manual control of the air spindle, and would like to be able to turn it on even if no job is running (for example, to drill a cross hole using jog panel controls).

In that case we will need a couple more definitions (explained below):

```
Last_prog_running   IS MEM28
Continue_M_functs   IS MEM29
```

And a little more work handling M32 in the program:

```
Continue_M_functs = ( CNC_program_running OR / Last_prog_running )
                        AND / Stop
Last_prog_running = CNC_program_running

M32 = ( M32 XOR ( Aux_1_key AND / Last_aux_1 )  ) AND Continue_M_functs

Air_spindle = M32
Aux_1_LED = M32
```

The "Continue_M_functs" indicator says that dual-mode M functions (automatic and manual) should continue to run unless the job just ended or a fault condition is present. This allows the spindle to stop automatically at the end of the job, while still allowing the operator to restart it while no job is running.

The M32 request bit will be set and cleared by M functions as they appear in a job. However, the request will also be toggled whenever the Aux1 key is pressed. Aux1 can then be used both to control the air spindle when no job is running, and also to override M32 and M33 during a job.

Deferring Spindle Brake Until Inverter Stops

In the first tutorial example we looked at a way to prevent the inverter from fighting the spindle air brake, by disabling the automatic brake mode whenever the spindle was running. That solution is convenient because it does not require any additional control wiring. However, it is not ideal. If the operator wants the brake to be applied whenever the spindle is stopped, he always has to press the BRAKE key; and if he pressed the BRAKE key

immediately after SPIN STOP then the brake will be applied, fighting the inverter just like it did before.

A better solution is to have the inverter inform the PLC when the spindle has decelerated to a stop. Most inverters have spare programmable outputs, and most allow you to program one of those outputs to be a zero-speed (sometimes called "baseblock") indicator.

Suppose we program an inverter output to close at baseblock, and wire that output to INP8 on our PLC board. We can then write the following PLC program:

```
Spindle_running IS INP8   ; 0 = stopped  1 = running

Brake           = Brake_mode AND / ( SpindleRelay OR
                             Spindle_running )
```

We can keep the standard logic for toggling Brake_mode on and off with the BRAKE key.

The line above simply inhibits the brake output until the spindle winds down. We might be tempted to simplify it to:

```
Brake           = Brake_mode AND / Spindle_running
```

But it is likely that there is a lag between the time the inverter receives the Run signal (SpindleRelay) and the time the zero-speed output opens. During this lag the inverter would be trying to start the spindle against the brake. Thus it is better to leave both conditions in place.

Tool Number Feedback

On a machine with an automatic tool changer, the on-screen tool number display is driven by the tool number in OUT41 through OUT48. This preserves the last automatic tool change command (actually the last M107) even if we exit and later restart CNC10.

However, suppose we implement a manual tool change (for example, using the Turret Index key on a T400 to rotate a new tool into position). In this case the machine has changed tools, but there has been no new M107; no new bits in OUT41 - OUT48; and no change in the screen display.

We would like to have our PLC program override the tool number bits after a manual index, without interfering with the tool numbers for automatic tool changes.

Assume we have a 6-station turret controlled by a Koyo PLC. The CPU7 sends the state of the Turret Index key to the Koyo, and the Koyo acts upon it to rotate the turret to a new position. When indexing is complete, the Koyo PLC knows the new position, and wants to plug that number into the tool number bits. We provide three bits of tool number information, plus a strobe bit to indicate that they should be copied into OUT41 through OUT43.

In the definitions section we have:

```
;
;  Actual tool information from Koyo
;
Actual_tool_1    IS INP17
Actual_tool_2    IS INP18
Actual_tool_4    IS INP19
Actual_tool_set  IS INP20

;
;  Tool number for ATC
;
Tool_BCD_1          IS OUT41
Tool_BCD_2          IS OUT42
Tool_BCD_4          IS OUT43
```

And in the program we have:

```
;
; Use Koyo signal to force tool number to selected pattern
;
Tool_BCD_1 = ( Tool_BCD_1 OR ( Actual_tool_set AND Actual_tool_1 ) )
             AND / ( Actual_tool_set AND / Actual_tool_1 )
Tool_BCD_2 = ( Tool_BCD_2 OR ( Actual_tool_set AND Actual_tool_2 ) )
             AND / ( Actual_tool_set AND / Actual_tool_2 )
Tool_BCD_4 = ( Tool_BCD_4 OR ( Actual_tool_set AND Actual_tool_4 ) )
             AND / ( Actual_tool_set AND / Actual_tool_4 )
;
```

This preserves the state of the Tool_BCD_x bits until Actual_tool_set is selected. Then it sets or clears each tool bit based on its corresponding bit from the Koyo.

Switching Optional Stops During Job Run, with Aux Key

Some operators request a more convenient way of turning on Optional Stops; a way to turn Optional Stops on after the job is started; and an Optional Stops feature that is not automatically canceled at the end of each job.

We can do this by bypassing the Optional Stops feature on the F4/Run menus, creating a custom macro for M1, and assigning control of the Optional Stops mode to an Aux key.

In this example we use the Aux4 key for Optional Stops control. INP38 and MEM32 are used internally between the M1 macro and the PLC program; any other available M function request and memory bit could be used instead.

The logic is that, whenever an M1 is encountered, we will stop and wait for CYCLE START if Optional Stops are on. The Aux4 key will toggle Optional Stops. The Optional Stops status is both stored and displayed in the Aux key LED.

Edit the PLC program source file as follows:

```
M1  IS INP38      ; 0 = idle   1 = opt. stop    M94/6 M95/6
                  ; (or any available request bit –
                  ;see M1 macro)

[...]

~~Aux_4_LED       IS OUT52~~
Optional_stops IS OUT52

[...]

Wait_at_M1 IS MEM32 ; <-- or any available memory bit -- see M1 macro

[...]

~~Aux_4_out       = Aux_4_out XOR ( Aux_4_key AND / Last_aux_4 )~~
Optional_stops  = Optional_stops XOR ( Aux_4_key AND / Last_aux_4 )
~~Aux_4_LED       = Aux_4_out~~
Last_aux_4      = Aux_4_key
;
; Handle optional stops whenever an M1 comes in
;
Wait_at_M1 = ( Wait_at_M1 OR ( M1 AND Optional_stops ) ) AND /
Cycle_start
;
```

Create a file CNC10.M1 as follows:

```
M94/6               ; <-- M1 input bit minus 32
M95/6               ; <-- M1 input bit minus 32
M101/192            ; <-- "Wait" memory bit plus 160
```

Add an entry to the CNC10XMSG.TXT file similar to the following:

```
MEM32
"Optional Stop - press CYCLE START"
```

The M1 macro flashes its M function request on and off. The PLC program takes this as a signal to set the "Wait_at_M1" memory bit if Optional Stops are enabled. Once the "Wait..." bit is set, it is not cleared until the operator presses CYCLE START. Thus the control will not restart from an M1 just because the operator turns off optional stops; he must also press CYCLE START one last time.

There is one minor side effect to this change. Consider a program like the following:

```
N1 G0 X0 Y0
N2 G1 F20 X1
N3 M1
N4 X2
```

With the built-in optional stops feature, if optional stops are turned off, the X axis will move continuously from X0 to X2.

With the macro-based version presented here, even if optional stops are turned off, the X axis will decelerate to a stop at X1, then resume the move to X2.

In most cases this will not matter. M1 codes are traditionally placed at tool changes and between operations, not in the middle of a cut.

Rapid Override on Early Controls

The Rapid Override feature was added to the control software in early 1994. Controls produced up to that time (M10 and M40 family) did not have a dedicated Rapid Override key on the jog panel. We modified the standard PLC programs to provide Rapid Override control using the Aux1 key:

```
Rapid_over_key IS INP49 ; AUX1_key
[...]
Rapid_over_LED IS OUT49 ; AUX_1_LED
[...]
Rapid_override        IS OUT79
[...]
Rapid_over_key_hit    IS MEM18
Last_rapid_over_key   IS MEM19
[...]
;
;   Toggle Rapid Override mode with AUX1 key
;
Rapid_over_key_hit      = Rapid_over_key & / Last_rapid_over_key
Last_rapid_over_key     = Rapid_over_key
Rapid_override          = ( Rapid_override XOR Rapid_over_key_hit )
                          OR / Already_run
Rapid_over_LED          = Rapid_override
```

Since most operators prefer to have Rapid Override enabled, the “OR / Already_run” clause is used to turn the mode on during the first scan of the PLC program. Thereafter it is toggled off or on each time the jog panel key is pressed.

The M400 and M39 jog panels have a dedicated Rapid Override key, mapped to INP79. The LED indicator in this key is mapped to OUT79, so it reflects the Rapid Override mode automatically, without any assignment needed in the PLC program. Otherwise, the logic in an M39 or M400 PLC program is identical to that above.

Some early M40 PLC programs have Rapid Override control, but do not turn it on automatically on the first scan. As a result, operators are often unaware of the feature. It is a fairly simple matter to edit or replace the PLC program to enable the Rapid Override mode on the first scan.

Some router tables used an M40 style jog panel mounted in a walk around pendant. Because most of these machines did not have PLC boards, and therefore did not have automatic spindle or coolant control, the pendants cover up the right half of the jog panel with a metal plate. Often the underlying keypad does not even have snap domes installed on that side.

In that case, there is no Aux1 key to be used for Rapid Override control. The best solution is to simply turn the Rapid Override mode on unconditionally. Again, this is a simple edit:

```
Rapid_override = / Zero
```

The jog panels on CNCDROs and M15s through model 7 have neither a Rapid Override key nor Aux keys, so their PLC programs should use the same line to turn on Rapid Override mode unconditionally.

PLC Diagnostics

Press <ALT I> at he main menu of CNC10 to turn on PLC diagnostics. Red and Green indicators will display the status of the current state of the PLC inputs, outputs and memory locations. Using the arrow keys you can highlight different PLC bits to show the label given to the selected bit.

Interfacing with a Koyo DirectLogic PLC

We provide an optional interface which allows you to connect the CPU7 to a Koyo DirectLogic DL205 PLC using a D2-240 CPU. The DL205 is a modular PLC system supporting up to 256 I/O points with a wide array of input and output devices.

The connection between the CPU7 and the D2-240 PLC CPU is via the tiny CPU7ADD board, plugged into a header on the CPU7, and the larger OPTIC232 board mounted in the cabinet. OPTIC232 connects to CPU7ADD via three optical fibers; it connects to the D2-240 via an RJ12 cable and optionally a two-wire interrupt signal from one of the PLC output modules.

With this system, the CPU7 sends 48 of its output bits to the Koyo CPU, and receives back 32 input bits (only 29 of which can be used, for reasons explained below).

Obviously, with perhaps 40 or 50 physical inputs, the Koyo cannot send every input to the CPU7. Fortunately, it doesn't have to. Typically the two PLC programs are written so that the Koyo does all the hard work, and only reports final results back to the CPU7.

Consider an automatic tool changer application:

To request a tool change, the CPU7 sends the desired tool number and a "start tool change" signal to the Koyo. The Koyo may have to deal with a large number of outputs (hydraulic or pneumatic solenoid valves; carousel motor starters; etc.) and inputs (tool counter switches, arm position switches, clamp/unclamp switches, etc.). However, it only needs to report back to the CPU7 with one bit indicating that the tool change is complete. It is usually not even necessary to report failure, since a properly written tool-change macro will have a timeout on the CPU7 side, forcing an error if the completion signal is not received in the expected length of time.

Although the CPU7 sends OUT1 through OUT48 to the Koyo, and received INP1 through INP32 back, it assumes that OUT1 through OUT3 and INP1 through INP3 are part of a 3/3 PLC board used along with the Koyo. Because of this, it does not update its copies of INP1 through INP3 with data from the Koyo, but instead saves those locations for the 3/3 PLC.

The 3/3 PLC should be used for any inputs where predictable timing is important. The lag time from when an input is actually triggered to when the CPU7 receives the bit change is unpredictable. It may be anywhere from 10ms to 100ms. If a Koyo input were used for a touch probe, this would lead to inconsistent probing and digitizing measurements, as the amount of movement between the time the probe tripped and the time the control stopped motion would vary.

Therefore you should always connect probe or tool detector inputs through a 3/3 PLC.

The 3/3 PLC can also be useful for electronic compatibility. Its inputs are 5VDC current sourcing inputs, while Koyo inputs are typically 24VDC sourcing or sinking. The fault output on our servo drive board is designed to connect only to a 5V sourcing input. Therefore you can connect the servo fault directly to a 3/3 PLC input, while you would have to use an intermediate relay if you wanted to connect it to a Koyo input.

When a Koyo PLC is used, it is typically equipped with an Analog output module (usually an F2-02DA-2) to control spindle speed. This provides 12-bit spindle speed resolution, instead of the 8 bits available with the RTK2 or 15/15 PLC. On the CNC10 Machine Parameters screen, set Parameter 31 to a value of -1.0 to select 12 bit spindle speed control via the Koyo. The current spindle speed request will be sent out in OUT17 through OUT28.

The F2-02DA-2 module provides two independent analog output channels. Since only one is needed for spindle speed control, the second may be used as a convenient solid-state output for the CPU7 interrupt signal mentioned above. This is particularly desirable if all your other Koyo output modules are relay outputs: a relay output could quickly exceed its expected service life if it had to cycle every time any input changed.

Limit Switch Interface between PLC and Servo Drive

In most control applications, the axis limit switches are connected both to PLC and to the servo drive. Connecting them to the PLC allows the CPU7 and CNC10 to "see" the switch, and therefore to know what happened and why axis motion has stopped. Connecting them to the servo drive allows the drive to inhibit motor current in the direction of the tripped switch, ensuring that the move is shut down even if software is misbehaving or the axis is out of control (e.g. failed encoder).

It is not strictly necessary to connect the limit switches to PLC inputs. With software travel limits set, the control will never hit a limit switch once machine home has been set. The control is even capable of homing to a limit switch which inhibits the drive, but does not appear on a PLC input. If you choose not to connect the limit switches to the PLC, enter zeros for the limit and home switch numbers on the Machine Configuration screen.

You will still be able to jog into a limit switch before machine home has been set. If you do, the message will be “full power w/o motion” or “position error”, not “limit tripped”. You will still be able to slow jog off the switch as usual.

If you have an unbalanced axis that tends to “fall off” the tripped limit switch, and you want to use that switch as a home, you should connect the switch to a PLC input as well as the drive inhibit. The software logic for homing to a drive inhibit alone is prone to “full power w/o motion” stalls if the axis drops back after hitting the switch.

The method for wiring the limit switches to both PLC inputs and drive inhibits depends on the PLC and servo drive hardware. The most common examples are as follows:

RTK2 and Servo Drives

The DC servo drives (SERVO1 and QUADDRV) use 5VDC pull-down (current sourcing) inputs for the limit switch inhibits. Since these are electrically identical to the inputs on our PLC I/O boards, the limit switches can be connected to both devices in parallel.

On the RTK2, the limit switches themselves are wired to headers H1 and H3 on the middle board (RTK2B). H3 connects the X, Y, and Z limits to inputs INP1 through INP6. Internal connections also pass these signals through to a nearby 10-pin Waldom connector. Fourth axis limit switches, if any, are wired to H1. These signals are not mapped to any PLC logic inputs but they are also passed through to the Waldom connector.

A separate cable carries the limit switch signals from the RTK2B Waldom connector to the servo drive's limit switch header. In addition to the eight switch signals, this cable has positions for +5VDC and ground, allowing the two devices to match their reference levels. In current production, only the +5 wire is carried from the RTK2 to the servo drive.

A three axis drive has a bank of eight DIP switches next to the limit switch header. The first six are limit switch defeaters: if thrown up (away from the board), the respective limit switch is disabled. Motor current will not be inhibited regardless of limit switch state. In normal operation these DIP switches should all be thrown down (towards the board) so that limits are enabled. The remaining two switches, labeled "G" and "5" select whether the ground and +5VDC signals which drive the inputs are taken from on board the drive, or from the PLC via the limit switch cable. In normal operation with an RTK2 these switches should be thrown down (towards the board) to select PLC signals.

A four axis drive has two banks of DIP switches: a bank of eight which enable or defeat the eight limit switches, just like on a three axis drive; and a separate bank of two which select +5V and ground reference signals. The latter function the same as on a three axis drive, except that the switch position is reversed: up away from the board for normal (PLC driven) operation, and down towards the board to use the drive's on board reference signals.

In a three axis control, connections on RTK2B H1 do not matter. In a four axis control, fourth axis limit switches should be wired to H1. If there are no limit switches (e.g. the

fourth axis is a 360° rotary table) then jumpers should be installed on H1 to simulate closed limit switches.

Although H1 does not directly control any PLC inputs, you could connect one or both fourth axis limits to a PLC input by jumpering from H1 to the appropriate connector. For example, you could connect the W+ limit to INP8 (terminals 7 and 8 of H5) in order to use it more reliably as a home switch on an unbalanced axis.

15/15 PLC and Servo Drives

The electrical interface between the 15/15 PLC and the servo drive is identical to that between the RTK2 and the drive, except that the 15/15 PLC does not have any pass-through header.

Instead, the servo drive limit cable connections are piggybacked with the physical switch connections on the input header H1.

In current production, both the +5VDC and ground signals in the limit switch cable are wired from H3 on the PLC to the limit header on the servo drive.

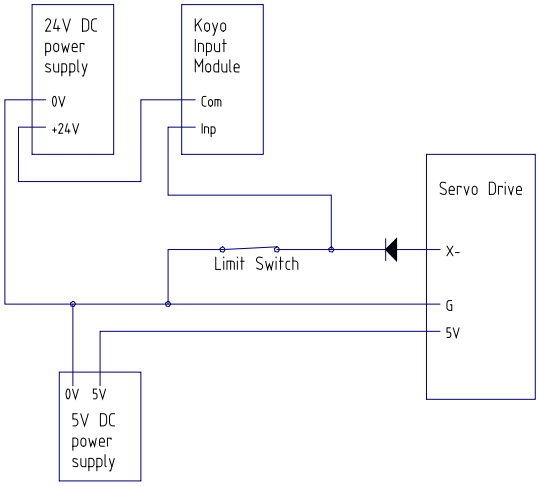
M15DRV1

Since the M15DRV1 combines PLC and servo drive on one board, the pass-through connection is made internally. Limit switches act as drive inhibits, and also appear on PLC inputs 7 through 12.

Koyo PLC and Servo Drives

Koyo PLC inputs typically operate on 24VDC, either current sourcing or current sinking. The 5VDC pull-up, which the servo drive can supply, is not enough to maintain a Koyo input.

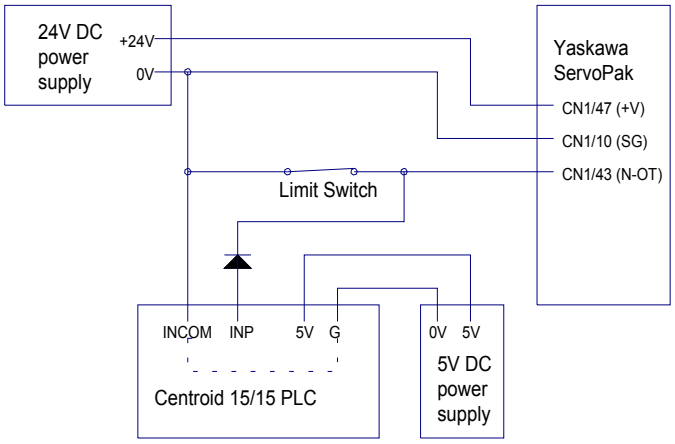
The solution to this incompatibility is to wire the limit switches to the Koyo as 24V current sourcing inputs, then connect each input to the servo drive limit header through a diode. The diode allows the limit switch input to be pulled to ground when the switch is closed, but prevents the Koyo's 24V from feeding into the drive's input when the switch is open. For this to work, the Koyo input and the servo drive ground references must be matched (connected). Typically the servo drive limit switch circuit is supplied by an external DC power supply; often the same one which powers a 3/3 PLC board and the OPTIC232 board.



RTK2 or 15/15 PLC and Yaskawa AC Drives

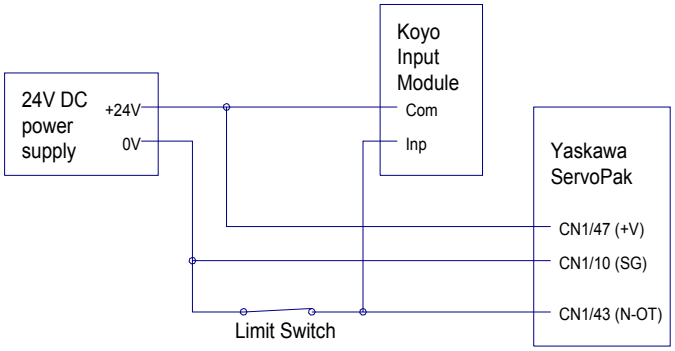
A similar electrical compatibility issue arises when using our PLC I/O board with Yaskawa or similar AC servo drives. The Yaskawa drives require 24V limit switch inputs, while our boards operate at 5V.

The solution is likewise similar: wire the limit switches to the Yaskawa drives at 24VDC, then connect them to PLC inputs through diodes. Again match the ground references by connecting the ground side of the 24V supply to the PLC input common.



Koyo PLC and Yaskawa AC Drives

Since both these devices use 24V inputs, no diodes are needed. Just connect the positive side of the 24V supply to both the PLC and the drives. Connect the negative side to the limit switches.



A note on AC servo tuning

With our DC drives, the standard PID parameters of Kp = 1.0, Ki = 0.004, and Kd = 10-20 work quite well in nearly all installations.

These gains are much too high for Yaskawa AC drives. One visible effect is that the axis will "jump" or "bang" when moving back off a tripped limit switch. Since moving on and off the limit switches is a normal part of machine homing, we would like the operation to be smooth and quiet.

Some retrofitters have reported defeating the drive inhibits in order to resolve the problem. They rely solely on the control software to enforce the limit switches. That is not the best solution. The best solution is to use lower PID gains, so that movement is smooth even when a limit switch is inhibiting current in one direction.

A good starting point for tuning is Kp = 0.2, Ki = 0.001, Kd = 2.0. The final numbers you use will vary depending on your installation and drive/motor selection.

XPLC Programming (PC.PLC)

The XPLC language is an addition to the standard Centroid programming language that provides more functionality. It must be used in conjunction with a "standard" program to solve more complex control applications that may have timing and arithmetic requirements, such as automatic tool changers. This manual assumes that one is familiar with the material in the standard PLC programming manual.

In order to write effective programs one must understand the way the standard program and the XPLC programs interact. When the phrase "standard" program is used, it refers to the program that is being executed on the motion control board. When the phrase "XPLC" program is mentioned it refers to the program that is executing on the computer, or PC.

Figure 2 shows the interaction between the standard and XPLC programs. Note that all INP, OUT, and MEM locations can be read in both a standard program and by an XPLC program. The figure below shows which program has control to write or change these bits. There are several questions that one may have at this point.

Why would any of the inputs need written or changed by a program? The answer is that not all of these INP bits are actually physical inputs. In fact, INP33-INP48 are actually M-function outputs that are controlled by M94 and M95 commands in M&G code programs.

How can operations in an XPLC program change or write bits that are controlled by the standard program? The answer is that the XPLC program must write to or change a location, often a memory bit, that it, XPLC, has access to. The standard program must read that location and write the corresponding bit. For example, suppose the XPLC program wanted to turn on the AUX1 LED on the jog panel when a timer had expired so as to alert the operator of a potential problem. The AUX1 LED is mapped to OUT49, which can only be changed by the standard program. In this case, the XPLC program would set, say MEM49, when it wanted to turn on the LED. The standard program would then set OUT49 based upon the value of MEM49. The actual code is listed below:

The lines common to both programs:

```
AUX1_LED      IS OUT49
XPLC_AUX1_LED IS MEM49
```

The standard PLC program lines:

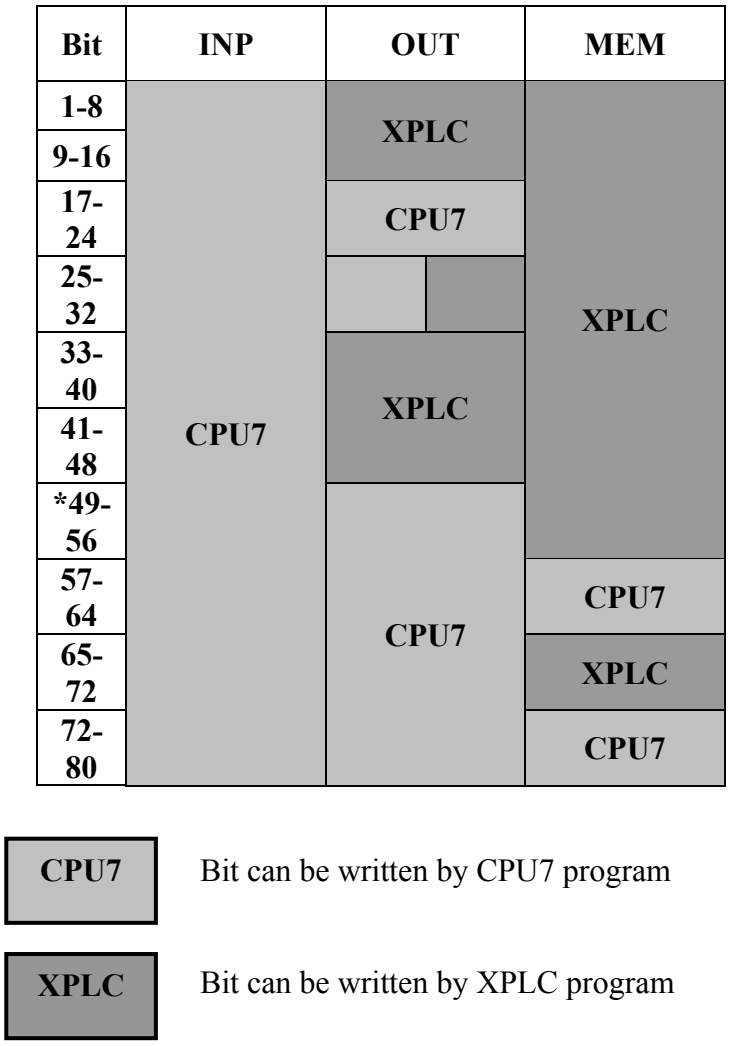
```
AUX1_LED = XPLC_AUX1_LED
```

The XPLC program lines:

```
IF T1 THEN (XPLC_AUX1_LED)
```

Not every location that can be written by a standard PLC program can be mapped to another location as there are not enough locations. It would be a very desirable goal to be able to map all these locations as it would mean only one program, the XPLC program, would ever need to be changed. Not so obvious is that even if there were enough memory locations to reach this goal, there are still underlying hardware and software conditions that require both programs to be maintained in certain situations. Later in the manual, a set of standard and XPLC programs will be provided that allow most all changes and additions to be made by changing only the XPLC program.

Figure 2. CPU7 vs. XPLC write control.



Bit can be written by CPU7 program

Bit can be written by XPLC program

*** MEM49 is used by the XPLC program to communicate to the standard program that the XPLC program is being used. Therefore, MEM49 must be set by the XPLC program for the programs to work together.**

Compiling and naming of XPLC programs.

The XPLC program, like the standard program, is text based. The structure is less stringent than a standard program. In other words, spaces are usually not required and individual lines may be broken across several lines without problems. Blank lines are also permitted.

Just like standard plc programming, an XPLC program must be compiled and must reside in a certain directory and be given a predetermined name.

A program is compiled by supplying the name of the XPLC source program to the compiler, XPLCCOMP.EXE.

Assuming the XPLC source program was named XPLC.SRC, the following command would be used to compile the program, with **boldface** type indicating what would be typed.

```
C:\PLC>XPLCCOMP XPLC.SRC
```

If the compilation is successful, a message similar to the following is displayed:

```
XPLCCOMP v. 1.00 - XPLC compiler
Copyright 2001-2003.

Compilation successful
Program size: 1
```

XPLCCOMP will create the compiled plc file named `XPLC.PLC`. If the compilation was not successful, there will be error messages displayed. See the section **Compilation Errors** for more information.

In order for this compiled program to be used by the system, it must be named `PC.PLC` and reside in the `C:\PLC` directory. The following command can be used to accomplish this:

```
C:\PLC>COPY XPLC.PLC PC.PLC
```

The system must be rebooted for any changes in the XPLC program to take effect.

The program that executes the PC.PLC program is PCPLC.EXE, which is located in the C:\PLC directory and is automatically called at startup by commands in the AUTOEXEC.BAT file.

The XPLC program has the same syntax for definition lines and comments as a standard program. For example:

```
X_MINUS    IS INP1 ; 0 = ok, 1 = limit tripped
```

XPLC Program lines are of the form

IF <*boolean_expression*> **THEN** <*actions*>

Standard programs are of the form

<*plc_bit*> = <*boolean_expression*>

It may be helpful to see how a line in a standard program would be written using a standard program, an XPLC program, and traditional relay ladder logic (RLL).

(1) Standard program

<*bit_type*> = <*boolean_expression* >

```
Lube      = CNC_program_running and not Lube_low
```

(2) XPLC program

IF < *boolean_expression*> **THEN** <*actions*>

IF `CNC_program_running and not Lube_low  THEN (Lube)`

(3) Relay Ladder Logic (RLL) program

<*boolean_expression*> <*actions*>

Note here that an "IF" is represented graphically as |-- , with THEN being interpreted as a coil connection at the end –(.

All three of these examples accomplish the same thing, namely that if a CNC program is running AND the low lube signal is not triggered, the lube will turn on.

The ***<boolean_expression>*** is a combination of the AND, OR, XOR and NOT operations between various inputs, outputs, memory locations, timer contacts, one-shot positive differential contacts, stage bits, and comparisons between integer word types.

The boolean operators are:

AND *or* **&**
OR *or* **|**
NOT *or* **!**
XOR *or* **^**

which are used with bit type operands, and

== (equal)
!= (not equal)
< less than
<= less than or equal
> greater than
>= greater than or equal

which are used with integer type operands, namely word memory and timer current values.

XPLC Bit Types

(the nn represents a number from 1-256):

INPnn - used for inputs
OUTnn - used for outputs
MEMnn - used as a temporary storage location or memory bit

Note that these are the same as used in standard programs.

STGnn - used for stage bits, explained later
PDnn - used for one-shot positive differential (leading edge one-shot)
Tnn - used for timer contact values

XPLC Integer Types

Wnn - a number that ranges between -2147483648 to +2147483647
TMRnn - how long a timer has been on, measured in 10ms (0.01sec) increments.
FLT - an internal fault number can have one of the following values:
0 = No error
1 = Stack fault
2 = Division by Zero fault
8 = Illegal instruction
<numerical_expression> – can be just a plain number, such as 78, or a combination of plain numbers and other integer types using *, /, -, and +, along with parentheses. For example W1 + W2/(3*TMR7), or 5/8, etc.

For those who may be familiar with relay ladder logic (RLL) programs, several examples with equivalent XPLC syntax are demonstrated below. For those who may not be familiar with RLL programming, the convention is that

X is used for an input type.
Y is used for an output type.
C is used for a memory bit.

The following XPLC definitions will be used in these examples.

X0 IS INP1
X1 IS INP2
X2 IS INP2
Y0 IS OUT1.

The terms **on, set, high, closed,** and **true** are normally used to denote a **logic (1)** and the terms **off, reset, low, open,** and **false** are normally used to denote a **logic (0)**. Whether a bit is "open" or "closed", "high" or "low", etc., depends upon the particular bit type. When using the aforementioned terms for logic 0 or 1, it is mainly used to differentiate between two different states.

In conventional RLL programming and hardware, an electrical input that is closed is a "1" and an input that is electrically open is a "0". In this system, however, a physical input on the PLC hardware that is electrically closed is a "0" whereas a physical input on the PLC hardware that is electrically open is a "1". In conventional RLL programming and with this system, an output that is "1" is considered on.

In order not be confusing for those who may already be familiar with RLL programming, the explanations below will assume conventional industry standard RLL conventions as noted above- just keep in mind that these industry standard conventions are opposite of the actual working in this PLC system for physical inputs on the PLC hardware.

Example 1. A simple one contact statement.


```
IF      X0                                          THEN    (Y0)
```

This is the most basic program statement. What it means is that if X0 is on, then turn on Y0. If X0 is off, turn off Y0. There are only two combinations that are possible using one contact of logic. These combinations are in the table below. This kind of table is referred to as a "truth table".

Table 1. Truth table for example 1.

X0	Y0
0	0
1	1

Example 2. The basic NOT statement.

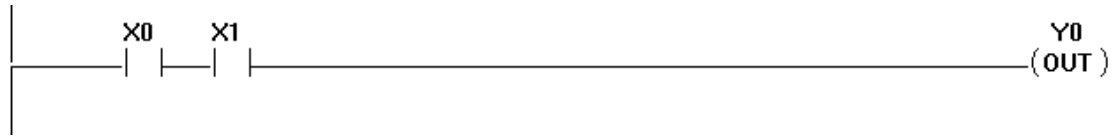

```
IF    !X0                                    THEN     (Y0)
```

This line of program has the opposite meaning. It is an example of the NOT operation in XPLC programming. What it means is that if X0 is not off, then turn off Y0. If X0 is off, turn on Y0. Again, there are only two possibilities:

Table 2. Truth table for Example 2.

X0	Y0
0	1
1	0

Example 3 – Basic AND statement




```
IF     X0 & X1                                THEN    (Y0)
```

This line of program is the basic AND operation. What it means is that if X0 is on **AND** X1 is on then Y0 will be on. Otherwise, Y0 will be off.

There are four different possibilities using logic with two contacts. These combinations and the result of Y0 are shown in the table 3 below.

Table 3. Truth table for example 3 (AND).

X0	X1	Y0
0	0	0
0	1	0
1	0	0
1	1	1

Example 4. The basic OR statement


```
IF   X0 | X1                                    THEN   (Y0)
```

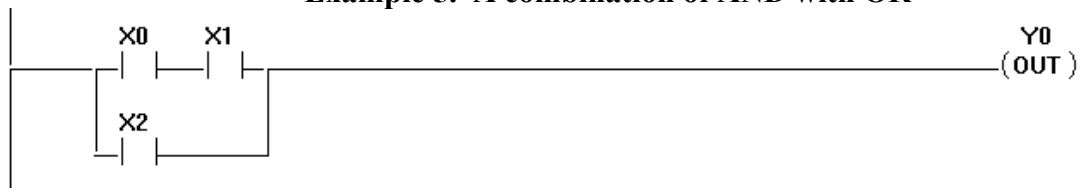
This is the basic OR operation. What it means is that if X0 **OR** X1 is on, then Y0 will be on. Otherwise, Y0 will be off.

Again, there are four different combinations that are possible. The truth table is in Table 4.

Table 4. Truth table for example 4 (OR).

X0	X1	Y0
0	0	0
0	1	1
1	0	1
1	1	1

Example 5. A combination of AND with OR




```
IF   X0 &  X1 | X2                                THEN   (Y0)
```

This example shows a simple combination of logic. When using combinations of AND, OR, XOR, and NOT, there are rules that determine the order in which operations occur. This is similar to the way there are rules when combining numbers and mathematical operations. It is easy to understand when it is realized that:

AND is like multiplication, OR is like addition, and like math, multiplication comes before addition. One could also say that multiplication has precedence over addition.

For example, imagine X0 being 2, X1 being 3, and X2 being 4. In this scenario, the line above would be written as 2 x 3 + 4 (= 10). This is helpful when trying to understand and determine what the result will be.

A program statement composed of three different contacts has eight different combinations. The table below, often referred to as a "truth" table, shows the outcomes of all these combinations.

Table 5. Truth table for example 5.

X0	X1	X2	Y0
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Example 6. Another combination of OR with AND

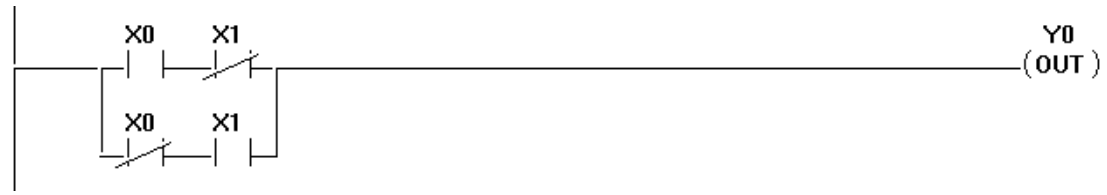

```
IF   X0 & (X1 | X2)                              THEN   (Y0)
```

Using the analogy to math, the way this logic is calculated is similar to the way that 2 * (3 + 4) is calculated. Note that in both the XPLC program and in the math example, parenthesis are needed to override the normal rules. Just like in math, the result is usually not the same, see Table 6 and compare it to Table 5.

Table 6. Truth table for example 6.

X0	X1	X2	Y0
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Example 7. The basic XOR statement




```
IF      X0 ^ X1                                    THEN    (Y0)
```

This example shows the XOR operation. What it means is that if one of X0 or X1 is on (1) , but not both, then Y0 will be turned on (set to 1). Otherwise, Y0 will be off (reset to 0). The truth table is below. Another way of stating this is that if both of the inputs do not have the same value, then the result is true.

Table 7. Truth table for example 7.

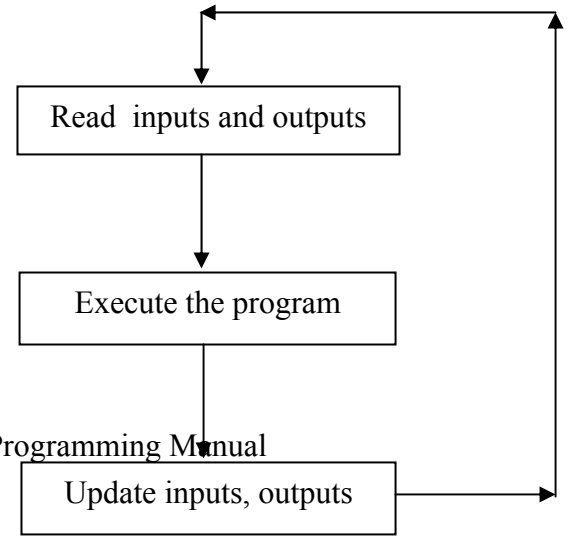
X0	X1	Y0
0	0	0
0	1	1
1	0	1
1	1	0

Program execution

Before continuing with examples of some of the more advanced contact types, it will be helpful to review and understand how a program is executed. Programs are composed of a series of lines that are executed from top to bottom continuously. The XPLC execution occurs 256 times a second, that is, the entire program is executed from top to bottom 256 times a second. Thus, the first concept to understand is that the program is constantly being executed. One execution of the program from top to bottom is referred to as a **pass**. Further explanations will refer to passes, such as the first pass of the program, or the second pass of the program, etc.

The second concept to understand is how the inputs, outputs, memory, and other bits are updated **as the program executes**. As a review from the standard ??plc manual??, the complete plc program execution follows this cycle:

When the inputs and outputs are read, a
of the their current state is made and
into a **copy buffer**. The copy buffer
a copy of the input and output states as
at the start of that pass of program execution.



snapshot
placed
contains
they were

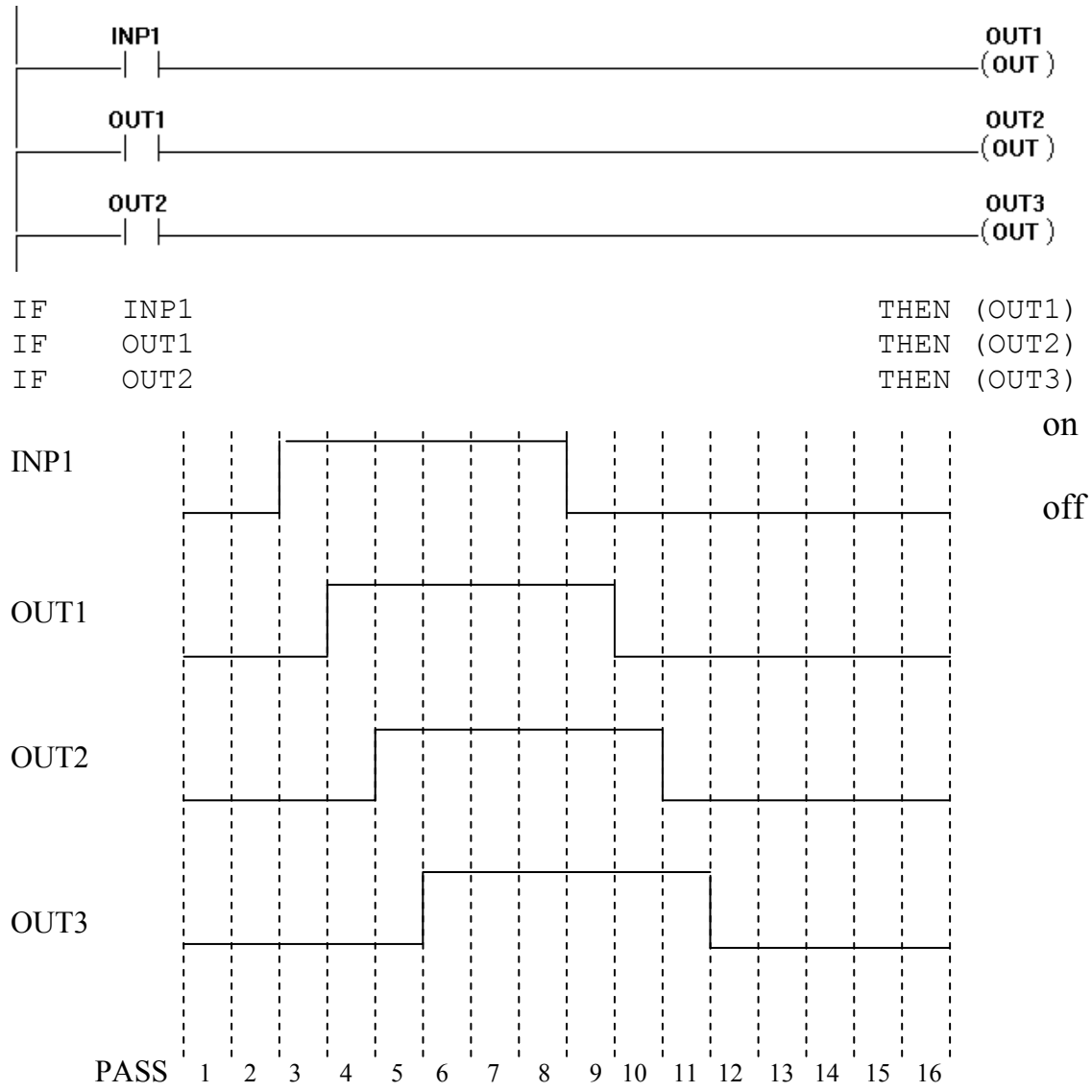
As the program is executing, <*boolean_expression*> that reference the inputs and outputs (INPnn and OUTnn) use the value from the copy buffer to determine what the state of the contact is.

There is another buffer used during execution, which is referred to as the **image buffer**. The image buffer contains what is going to be the new value of the input or output after the program has finished execution. This is the place where, during execution of the program, the new values of the inputs and outputs will be stored.

Most of the problems and errors found when developing programs are directly related to these principles, namely that program execution is continuous from top-to-bottom and that input and output states are read from the copy buffer and written using the image buffer.

For other types, such as memory bits, stage bits, and word values, there is no buffer. Program statements that write these types will **immediately** change the value. To better understand this concept, some timing diagrams are presented below. The first example shows when outputs are updated. This would be the same if the OUTs were INPs. The second example shows the timing and updating when using MEM bits.

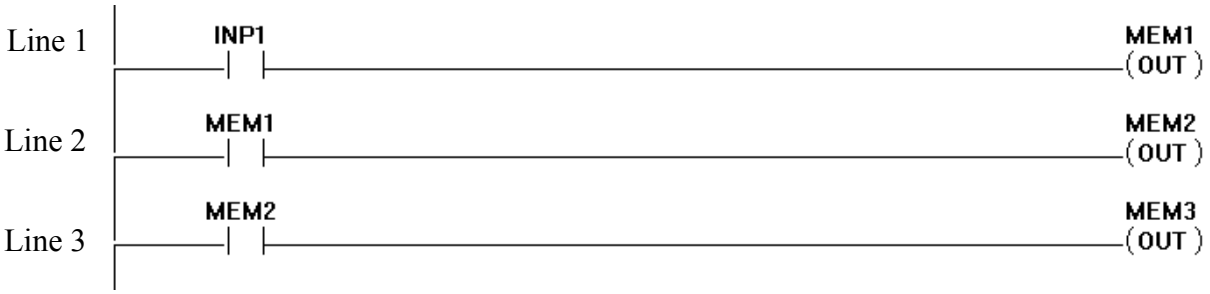
Example ??. Timing diagram of writing of outputs.



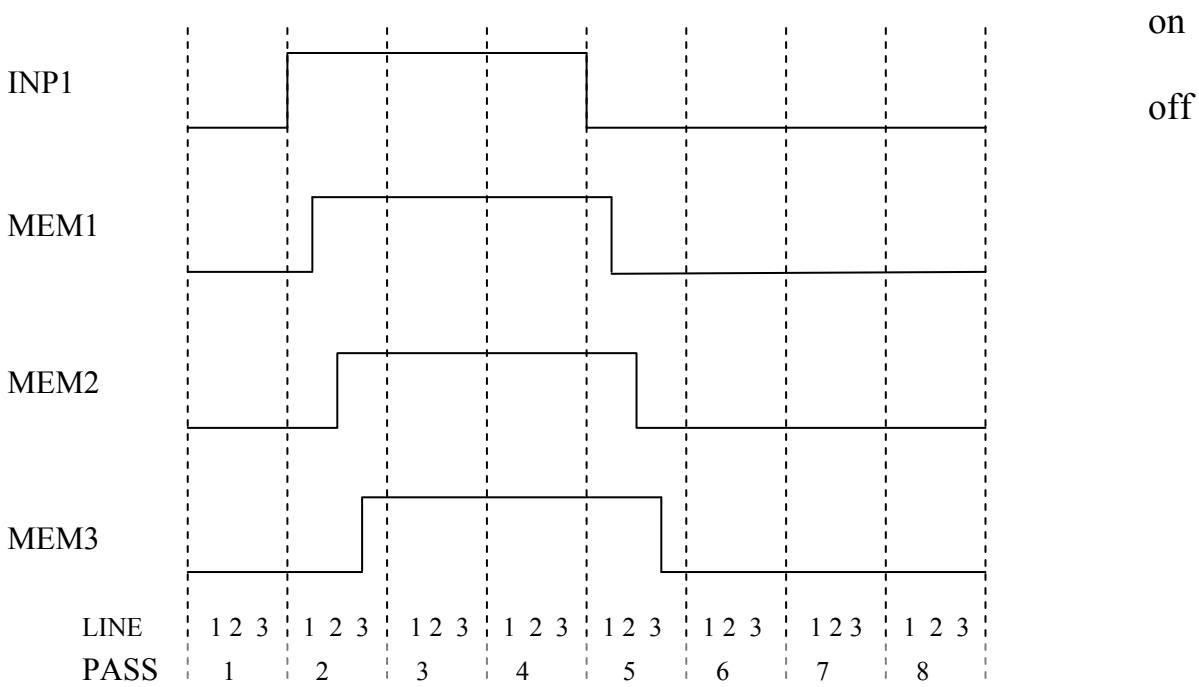
Hopefully this example has demonstrated how the inputs and outputs are read and written, and the effects this has on when they are updated.

Consider now an example of writing MEM bits. The same example would hold true if the MEM bits were replaced by STG or PD bits or for word memory assignments.

Example ??. Timing and states when writing MEM bits.

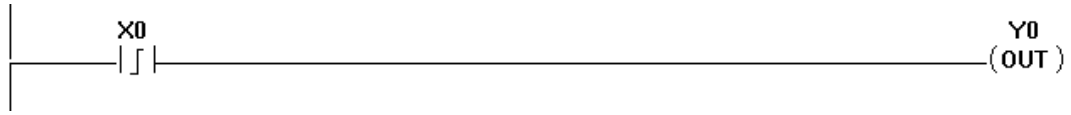



```
IF     INP1                                  THEN (MEM1)
IF     MEM1                                  THEN (MEM2)
IF     MEM2                                  THEN (MEM3)
```



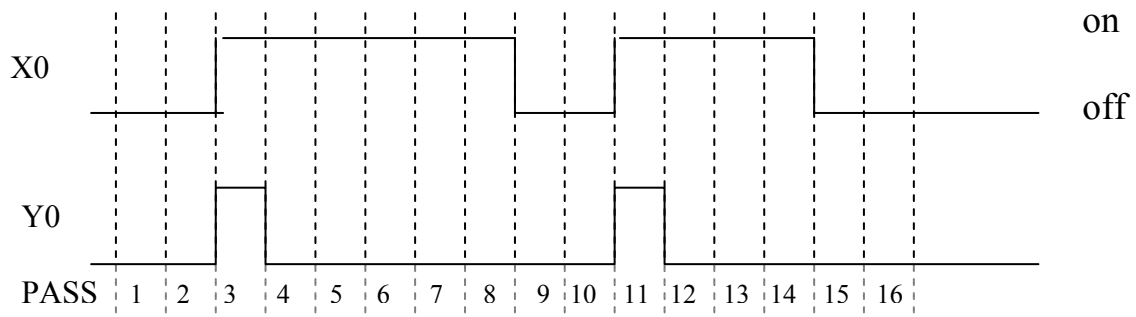
Note that the reason the MEM bit rising and falling edges are slightly skewed throughout pass #3 and pass #5 is to show that they don't actually get changed until that particular line of logic is executed.

Example ??. The basic rising edge one-shot or positive differential PD

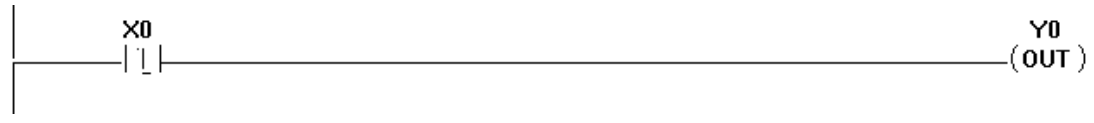



```
IF      X0                                   THEN (PD1)
IF     PD1                                   THEN (Y0)
```

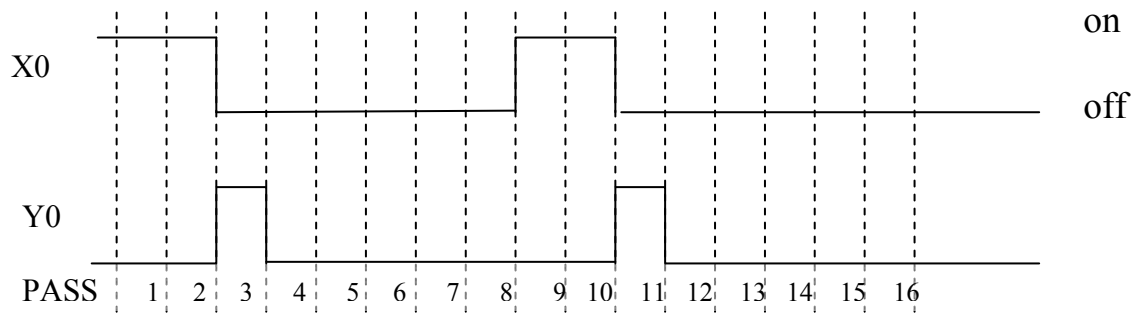
This example demonstrates the use of a positive differential, or rising edge one-shot type of contact. What it means is that if X0 is off one pass and then on the next pass, then Y0 will be turned on for one pass. Perhaps a timing diagram can better show this.



Example ??. The basic falling edge one-shot or negative differential PD




```
IF   !X0                                THEN (PD1)
IF   PD1                                THEN  (Y0)
```



One-shots are often used to toggle states without the side effect of causing rapid flickering of outputs. Specific applications of one-shots and when to use them will be covered in a later section of manual. One-shots are really nothing more than a convenience when writing programs. The same result can be obtained using just MEM bits and having the program keep track of the last state. Example ?? above can be written using MEM bits as such:

```
IF X0 & !MEM1 THEN (Y0)
IF X0         THEN (MEM1)
```

Note when using one-shots it takes another line of XPLC program to achieve the same effect as would be required in RLL programming. The format used for XPLC programming was used to

cut down the number of different token types that are used. Note that an XPLC program allows an entire rung of logic to control the one-shot. A similar RLL program in this case would also require two rungs of logic.

Example ??. Basic comparison statements.


```
IF     W1 == W2                                          THEN (Y0)
```


```
IF     W1 != W2                                          THEN (Y0)
```


```
IF      W1 < W2                                          THEN   (Y0)
```


```
IF      W1 >= W2                                         THEN    (Y0)
```

The above examples demonstrate comparisons between integer type operands. What they mean is that if the comparison is true, then turn on Y0. Otherwise, turn it off. In the first program line above, if W1 = 3 and W1 = 4, then Y0 would be turned off.

Example ??. Basic calculations.

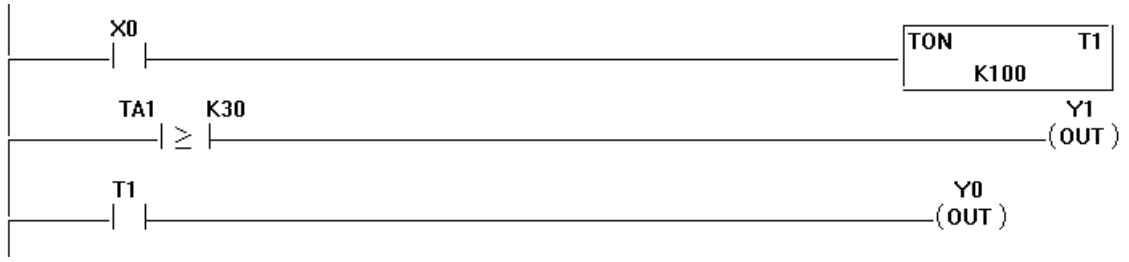
XPLC programming of mathematical operations is easier to accomplish than using RLL as it does not require the use of stacks, accumulators, and various math boxes. See below.


```
IF     X0                                    THEN  W1 = 3 * 4
```


```
IF   X0                                      THEN  W1 = W1 + 1
```

When programming XPLC mathematical expressions, simply write the expression as it would be written on paper.

Example ??. Timers




```
IF    X0                                     THEN T1 = 100,  (T1)
IF    TMR1 >= 30                             THEN            (Y1)
IF    T1                                     THEN    (Y0)
```

The programs above will turn on Y1 after X0 has been on continuously for at least 300 ms (0.3 seconds) in addition to turning on Y0 after 1.0 second.

Timers have four components associated with them, which are:

(1) A **preset value**. In the example above, the preset value is 100, or 1 second. The value is specified using the syntax Tnn = <*numerical_expression*> in an action statement. The preset value only needs to be set once. In a typical program, the preset value is set once in the beginning of the program.

(2) A **timer input** which, when switched on, causes the timer to start keeping track of elapsed time. If the timer input is off, the timer is reset to 0.

(3) A **current value** that returns how long the timer has been on. This is where the elapsed time is stored. The syntax for accessing the timer current value is to use a TMRnn reference in an integer comparison.

(4) A **timer contact** that is on when the current value >= preset value. The syntax for referencing a timer contact is to use a Tnn reference in a <*boolean_expression*>.

With XPLC programs there are no accumulating timers or up/down counters. However, the functionality of these typical RLL elements can be coded using a combination of word memory and timers with a bit of skill. Examples of up/down counters and accumulating timers are provided later in the manual.

Example ??. Executing multiple actions for the same boolean expression.


```
IF      X0                                      THEN (Y0),
                                                     (Y1),
                                                     (Y3)
```

In this example, when X0 is on, Y0, Y1, and Y3 are on. Otherwise, they are all off.

Understanding Stages

Stages are a feature of XPLC programming that helps write structured programs. Stages also aid in program development and maintenance. As used in XPLC programming, they are closely related to the RLL programming of master control relays (MCS and MCR coils).

Any IF statement of an XPLC program may be preceded by a STGnn. This marks all program lines after the STG, until another STG or the end of the program occurs, as belonging to that stage.

Example ??.

```
1:  STG1
2:
3:  IF INP1 THEN (Y0)
4:  IF INP2 THEN (Y1)
5:
6:  STG2
7:
8:  IF INP3 THEN (Y2)
9:  IF INP4 THEN (Y3)
```

In the example above, lines 2-5 are part of STG1 and lines 7-9 are part of STG2.

A stage can be ON or OFF. It has an associated internal memory bit that determines whether it is ON (1) or whether it is OFF (0). This memory bit can be written (turned on and off) by using certain action statements and can be read by referencing it in a <*boolean_expression*>. For example:

```
IF INP1 THEN (STG1)
```

This is an example of how the STG status is turned on or off. If INP1 is on, then STG1 would be turned on. If INP1 is off, STG1 would be turned off.

```
IF STG1 THEN (Y0)
```

This is an example of reading a stage status. Here, if STG1 is on, then Y0 is turned on. If STG1 is off, then Y0 would be turned off.

How stages work in program execution

When the XPLC program is being executed and it encounters a STGnn, the executor marks that stage as the **active stage.** If the active stage is ON, then the execution of the program continues normally until the next stage or end of program. If the stage is OFF, the effect is that **all the <*boolean_expression*> are considered false for that stage**.

```
STG1
IF INP1 THEN (Y0)
```

In this example, if STG1 is OFF, then Y0 would be turned off **regardless of whether INP1 was on or off**. If STG1 is ON, then Y0 would be on if INP1 is on and off if INP1 is off.

STG1 is ON, by default, when the control system is initialized. STG2-STG256 are OFF at system initialization. This allows a starting point for initialization without having to explicitly turn it ON.

Note that like PD contacts, stages are a feature that can also be implemented in another way using MEM bits and additional program lines. There is no requirement that stages are even used in an XPLC program, but experienced PLC programmers use them to make their programs easier write and easier to understand and maintain.

Specific uses and application of stages are covered more in-depth later in the manual.

<Actions>

<*Actions*> are the various commands that can be executed depending upon the value of a <*boolean_expression*>. So far, most of the examples in this manual have used one specific type of action- the output coil. The ones that have not were the basic calculations and the timer example. All of the possible actions are explained below.

Output coil ( )

The output coil action has several forms. The most common form is used to turn on or off a specified bit and is used with **INP**, **OUT**, **MEM**, or **STG** bits.

```
IF INP1 THEN (OUT1)
```

What it means is that if INP1 is ON, then turn on OUT1. If INP1 is off, then turn off OUT1.

```
IF INP1 & !INP3 | MEM3 | W1 <= 3 THEN (OUT1)
```

Here, if the entire <*boolean_expression*> "INP1 & !INP3 | MEM3 | W1 <= 3" is true, then OUT1 is on. Otherwise, it is off.

When an output coil is used with a **PD** bit, as in

```
IF INP1 THEN (PD1)
```

then the meaning is:

if <*boolean_expression*> is true and on the previous pass it was false, then turn on PD. Otherwise, turn off PD.

When an output coil is used with a **T** or **TMR** bit, it means to connect the value of <*boolean_expression*> to the **timer input**. Thus, if <*boolean_expression*> is true, then the timer

updates the **current value** with the elapsed time. If <*boolean_expression*> is false, **the current value is set to 0**..

Other actions

All the remaining types of action statements work in this way:

if <*boolean_expression*> is true, then **execute** the command.
if <*boolean_expression*> is false, then **do not execute** the command.

SET, RST

These actions turn on and turn off **INP**, **OUT**, **MEM**, and **STG** bits.

```
IF INP1 THEN SET OUT1
```
If <*boolean_expression*> is true, then the bit is turned on.

```
IF INP1 THEN RST OUT1
```
If <*boolean_expression*> is true, then the bit is turned off.

*Note that in these two examples that if <boolean_expression> is false**, nothing happens**. It is a mistake to think that if <boolean_expression> is false, the SET command is turned into a RST command, or vice versa. The same is true for any action that is not an output coil.*

The line:

```
IF INP1 THEN (OUT1)
```

is the same as:

```
IF  INP1 THEN SET OUT1
IF !INP1 THEN RST OUT1
```

= (Assignment)

The assignment command is used to assign a <*numerical_expression*> to a word memory location **W** or to set a timer **preset value**, provided <*boolean_expression*> is true. If <*boolean_expression*> is false, no assignment is made.

```
IF INP1 THEN W1 = 60 * 10
IF INP1 THEN T1 = 500
```

In the first example, if INP1 is on, then W1 is assigned the value 600 (60 * 10).
In the second, if INP1 is on, then T1 **preset value** is assigned 500 units, or 5 seconds.

JMP STGnn

The JMP command resets the **active** stage and sets STGnn.

```
STG1

IF INP1 THEN JMP STG2
```

Here, if INP1 is on, then the JMP STG2 command will reset STG1 and set STG2. Note that if the active stage is reset execution continues normally for lines in that STG.

WTB Wnn MEMnn
WTB Wnn OUTnn

The WTB commands write the lower eight bits of the Wnn word to a series of MEM or OUT bits. The least significant bit is written to nn and the most significant is written to nn+7

```
IF 1==1 THEN W1 = 170, WTB W1 OUT41
```

After the above line is executed,

OUT48	OUT47	OUT46	OUT45	OUT44	OUT43	OUT42	OUT41
1	0	1	0	1	0	1	0

170 (Decimal) = AA (Hexadecimal) = 10101010 (Binary)

BCD Wnn
BIN Wnn

The BCD Wnn command converts the value in Wnn to binary coded decimal (BCD) format. The BIN Wnn command converts the value in Wnn to binary, assuming that it was in BCD format.
LDT Wnn
LTS Wnn
LMT Wnn
LCP Wnn

All of the above commands are for support of automatic tool changers.

LDT Wnn will load the value of the last tool number into Wnn. Tool numbers are sent to the XPLC system when an M107 command is executed in an M&G code program.

LTS Wnn will load the value of the tool in the spindle into Wnn. This value comes from the CNC10.JOB file at CNC10 startup.

LMT Wnn will load the value of the maximum number of tools into Wnn. This value comes from CNC10 Machine Parameter 161 at startup.

LCP Wnn will load the value of the tool carousel position into Wnn. This value comes from the CNC10.JOB file at startup.

LSR Wnn

This command will load into Wnn a value that can be checked to see if the CNC10 software is currently running. Its main use is to disable carousel indexing when CNC10 is not running since if the carousel position changed, CNC10 would not be able to monitor the new position and save it in the CNC10.JOB file.

```
IF 1==1     THEN LSR W1
IF W1 == 17 THEN (Disable_Tool_Indexing)
```

In this example, Disable_Tool_Indexing is a memory bit that would be used later in the XPLC program to prevent tool indexing via Aux keys.

LP0 Wnn – **LP9** Wnn

These ten commands are used to load the value of CNC10 Machine parameters into Wnn.

LP0 loads CNC10 Machine Parameter 170 into Wnn.
LP1 loads CNC10 Machine Parameter 171 into Wnn.
…
LP9 loads CNC10 Machine Parameter 179 into Wnn.

Machine Parameters 170-179 can have values between 0 – 65535.

These commands have many applications, including changing XPLC program behavior based upon certain values and toggling the logic value of an input.

Standard and XPLC programs.

Both the standard and XPLC programs may differ according to the physical hardware that constitutes the system. Such hardware considerations are mainly:

(1) PLC hardware (PLC3/3, PLC 15/15, RTK2, SERVO3IO, PLCIO2, RTK3, etc.)
(2) Jog Panel hardware: None, keyboard, M39 pendant, Uniconsole-2, etc.

A basic set of standard and XPLC source programs are presented below. These programs are targeted to a system composed of a SERVO3IO used with a Uniconsole-2 jog panel.

When listing programs and in order to aid readability, the `Courier typeface` is used for program lines and Arial typeface is used for the more meaningful comments.

The standard program.

```
; * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
; *
; *  File:    BASECPU1.SRC
; *  Purpose: STANDARD PLC program for SERVO3IO/UNICONSOLE-2
; *
; *          Works in conjunction with XPLC program BASEXPC1.SRC
; *
; *          CNC Configuration Settings
; *              Jog Panel Type:   Uniconsole-2
; *              PLC Type:         Normal
; *
; *          CNC Parameter Settings
; *              P31   = 1 or 2 (for COM1 or COM2 spindle control)
; *
; *              P177  = 0.0  Normal behavior
; *                    = 1.0  Fault Override in effect
; *
; *              P178  Bit flags
; *                    LubeNONC         ; P178 Bit 0  (1)
; *                    SpindleNONC      ; P178 Bit 1  (2)
; *                    NoReverseSpindle ; P178 Bit 7  (128)
; *                    SpinRangeNONC    ; P178 Bit 9  (512)
; *
; *              P179  Lube timer settings  (see below for full explanation)
; *                    = 0  On when running a job
; *                    MMMSS   Off for MMM minutes, On for SS seconds
; *                            if SS = 0, On for at least MMM minutes
; *
; *
; *   () Support for probing and digitizing.
; *   () Handling of both NO/NC spindle drive faults and automatic reset.
; *   () Handling of both NO/NC low lube faults.
; *   () Handling of drive faults.
; *   () Support of two limits switches each for all three axes
; *   () Support for NO/NC spindle high/low range that can
; *      optionally reverse direction.
; *   () Support for drawbar unclamping.
; *   () Control of flood coolant in both automatic (M8) and manual modes.
; *   () Control of mist coolant in both automatuc (M7) and manual modes.
; *   () Automatic/manual control of a spindle brake.
; *   () Lube pump timing and control.
; *   () Fault override logic via parameter setting.
; *
; * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
;
;  By way of standard practice, the definitions for all the INP bits are
;  grouped together as well as those for OUT, and MEM bits.
;
;  If a bit is unused in the program, it is prefixed with a u_.
;
; ************************************************************************
;                             INPUT DEFINITIONS
```

```
; ************************************************************************
;
;  Remember that for the SERVO3IO that an input that is electrically
;  closed is a 0.  An input that is electrically open is a 1.
;
E_stop               IS INP1    ; 0 = Normal    1 = E_STOP ** HARD CODED **
Probe_input          IS INP2    ;
Spindle_range_in     IS INP3    ;
Spindle_ok           IS INP4    ; 0 = fault     1 = ok
Probe_not_detected   IS INP5    ; 0 = probe     1 = no probe
Lube_Fault_In        IS INP6    ;
X_minus              IS INP7    ; 0 = ok        1 = Tripped
X_plus               IS INP8    ; 0 = ok        1 = Tripped
Y_minus              IS INP9    ; 0 = ok        1 = Tripped
Y_plus               IS INP10   ; 0 = ok        1 = Tripped
Z_minus              IS INP11   ; 0 = ok        1 = Tripped
Z_plus               IS INP12   ; 0 = ok        1 = Tripped
Servo_fault          IS INP13   ; 0 = Drive ok  1 = Drv Fault ** HARD CODED **
Tool_Release         IS INP14   ; 0 = pressed   1 = not pressed
Zero_Speed           IS INP15   ; 0 = at zero speed
PLC_ok               IS INP16   ; 0 = fault     1 = ok
;
; The lines above define the physical inputs that one would wire to
; on the SERVO3IO. The reason that PLC bits are defined in this way
; is because it makes the program easier to understand.
;
; The ** HARD CODED ** comment refers to the fact that on the SERVO3IO,
; INP1 must be the emergency stop input because the SERVO3IO performs special
; processing when it sees the E_stop input.  The other ** HARD CODED ** input
; is INP13.  There is no physical connection to this input as it is internal
; to the hardware.   The SERVO3IO will, however, cause INP13 to change
; to a 1 when there is a fault detected, such a sif the SYNC or DATA fibers
; are not connected.
;
;
; INP16 is reseved for PLC line check. It is an internal signal that has no
; physical comnnection.  If the PLC fiber connections CLK, RXS, or TXS
; are removed then the bit is turned off.
;
;
; INP17 - INP32 have no physical mapping for SERVO3IO.
;
u_inp17              IS INP17   ;
u_inp18              IS INP18   ;
u_inp19              IS INP19   ;
u_inp20              IS INP20   ;
u_inp21              IS INP21   ;
u_inp22              IS INP22   ;
u_inp23              IS INP23   ;
u_inp24              IS INP24   ;
u_inp25              IS INP25   ;
u_inp26              IS INP26   ;
u_inp27              IS INP27   ;
u_inp28              IS INP28   ;
u_inp29              IS INP29   ;
u_inp30              IS INP30   ;
u_inp31              IS INP31   ;
u_inp32              IS INP32   ;
;
; M94/M95 Mappings
;
;
; The M94/M95 mappings are interfaces to M&G code programming.
; Turning on and off these bits is how certain M-functions work.
;
; In M&G code programming, the command
;
;    M94/1 turns on  INP33
;    M95/1 turns off INP33
;
;    M94/2 turns on  INP34
```

```
;    M95/2 turns off INP34
;    ...
;    ... and so on
;    ...
;    M94/16 turns on  INP48
;    M95/16 turns off INP48
;
;  These pre-defined M-functions are:
;
;  M3 (Spindle CW)
;    M95/2
;    M94/1
;
;  M4 (Spindle CCW)
;    M95/1
;    M94/2
;
;  M5 (Spindle Off)
;    M95/1/2
;
;  M7 (Flood Coolant Off, Mist Coolant On)
;    M95/3
;    M94/5
;
;  M8 (Mist Coolant Off, Flood Coolant On)
;    M95/5
;    M94/3
;
;  M9 (Flood Coolant Off, Mist Coolant Off)
;    M95/3/5
;
;  M10 (Clamp On)
;    M94/4
;
;  M11 (Clamp Off)
;    M95/4
;
;  This program does not use a clamp but its definition
;  is included here because it has a predefined meaning.
;
;  ** Note that M6 (Tool Change) and M39 (Air Drill) commands
;  ** also have pre-defined meanings which affect one or more of these
;  ** bits.  See the M-series Operators Manual for more details.
;
M3                    IS INP33   ; Map to M94/1  M95/1 (Spindle CW)
M4                    IS INP34   ; Map to M94/2  M95/2 (Spindle CCW)
M8                    IS INP35   ; Map to M94/3  M95/3 (Flood On)
M10                   IS INP36   ; Map to M94/4  M95/4 (Rotary Clamp)
M7                    IS INP37   ; Map to M94/5  M95/5 (Mist)
M94_M95_6             IS INP38   ; Map to M94/6  M95/6
M94_M95_7             IS INP39   ; Map to M94/7  M95/7
M94_M95_8             IS INP40   ; Map to M94/8  M95/8
M94_M95_9             IS INP41   ; Map to M94/9  M95/9
M94_M95_10            IS INP42   ; Map to M94/10 M95/10
M94_M95_11            IS INP43   ; Map to M94/11 M95/11
M94_M95_12            IS INP44   ; Map to M94/12 M95/12
M94_M95_13            IS INP45   ; Map to M94/13 M95/13
M94_M95_14            IS INP46   ; Map to M94/14 M95/14
M94_M95_15            IS INP47   ; Map to M94/15 M95/15
M94_M95_16            IS INP48   ; Map to M94/16 M95/16
;
;  Jog panel AUX keys
;
Brake_key             IS INP49   ; 1 = pressed
Aux_2_key             IS INP50   ; 1 = pressed
Aux_3_key             IS INP51   ; 1 = pressed
Aux_4_key             IS INP52   ; 1 = pressed
Aux_5_key             IS INP53   ; 1 = pressed
Aux_6_key             IS INP54   ; 1 = pressed
Aux_7_Key             IS INP55   ; 1 = pressed
```

```
Aux_8_key            IS INP56   ; 1 = pressed
Aux_9_key            IS INP57   ; 1 = pressed
Mist_Key             IS INP58   ; 1 = pressed
;
; INP59 -INP62 are physical inpus located on the CPU.
; They should not be used.
;
u_inp59              IS INP59   ;
u_inp60              IS INP60   ;
u_inp61              IS INP61   ;
u_inp62              IS INP62   ;
;
;                                  High     Med-High  Med-Low  Low
;
Mid_range            IS INP63   ;    0          1        1      0
High_Low_Range       IS INP64   ;    0          0        1      1
;
; INP65 is set by CNC software when running a job or in MDI mode.
;
CNC_program_running  IS INP65   ; 0 = stopped   1 = running
;
; Spindle control related keys
;
Spindle_mode_switch  IS INP66   ; 0 = manual    1 = auto
Spindle_Dir_Key      IS INP67   ; 0 = CCW       1 = CW
Spindle_start_key    IS INP68   ; momentary     1 = start spindle
Spindle_stop_key     IS INP69   ; momentary     1 = stop spindle
;
Aux_11_key           IS INP70   ; Aux_11_Key
Aux_12_key           IS INP71   ; Aux_12_key
;
; Coolant related keys
;
Coolant_mode_switch  IS INP72   ; 0 = manual    1 = auto
Flood_Key            IS INP73   ; Flood_Key
;
; Misc keys
;
Cycle_cancel         IS INP74   ; 1 = pressed
Cycle_start          IS INP75   ; 1 = pressed
Tool_check_key       IS INP76   ; 1 = pressed
Pause                IS INP77   ; 1 = Pause
Block_mode_key       IS INP78   ; 0 = auto      1 = block mode
Rapid_over_key       IS INP79   ; Not present on a Uniconsole-2
;
;*****************************************************************************
;                           OUTPUT DEFINITIONS
;*****************************************************************************
Mist                 IS OUT1    ; 0 = off       1 = on
Lube                 IS OUT2    ; 0 = off       1 = on ** Hard Coded **
Flood                IS OUT3    ; 0 = off       1 = on
Brake                IS OUT4    ;
VFD_reset            IS OUT5    ; 0 = Normal    1 = Reset
u_out6               IS OUT6    ;
Drawbar_Sol          IS OUT7    ;
;
; OUT8-OUT15 have no physical mapping for SERVO3IO
;
u_out8               IS OUT8    ;
u_out9               IS OUT9    ;
u_out10              IS OUT10   ;
u_out11              IS OUT11   ;
u_out12              IS OUT12   ;
u_out13              IS OUT13   ;
;
; OUT14 and OUT15 are used here as memory bits.
; On other PLC hardaware, such as PLC 15/15, RTK2, PLCIO2
; They are the physical ouputs that control spindle
; direction and enable.
;
Spindle_Enable       IS OUT14   ;
```

```
Spindle_Dir_Out      IS OUT15   ;
;
;  OUT16 is an internal bit that must be set
;  when INP16 (PLC fault INP) is set.
;
Plc_fault_out        IS OUT16   ; 0 = normal    1 = fault
;
Spindle_speed0       IS OUT17   ; Reserved for 8/12 bit spindle speed
Spindle_speed1       IS OUT18   ; Reserved     "
Spindle_speed2       IS OUT19   ; Reserved     "
Spindle_speed3       IS OUT20   ; Reserved     "
Spindle_speed4       IS OUT21   ; Reserved     "
Spindle_speed5       IS OUT22   ; Reserved     "
Spindle_speed6       IS OUT23   ; Reserved     "
Spindle_speed7       IS OUT24   ; Reserved     "
Spindle_speed8       IS OUT25   ; Reserved for 12 bit spindle speed
Spindle_speed9       IS OUT26   ; Reserved for 12 bit spindle speed
Spindle_speed10      IS OUT27   ; Reserved for 12 bit spindle speed
Spindle_speed11      IS OUT28   ; Reserved for 12 bit spindle speed
;
; OUT29-OUT40 have no physical mapping for SERVO3IO
;
u_out29              IS OUT29   ;
u_out30              IS OUT30   ;
u_out31              IS OUT31   ;
u_out32              IS OUT32   ;
u_out33              IS OUT33   ;
u_out34              IS OUT34   ;
u_out35              IS OUT35   ;
u_out36              IS OUT36   ;
u_out37              IS OUT37   ;
u_out38              IS OUT38   ;
u_out39              IS OUT39   ;
u_out40              IS OUT40   ;
;
;  IF MEM49 = 1 THEN OUT41-OUT48 ARE XPLC PROGRAMMABLE
;  IF MEM49 = 0 THEN OUT41-OUT48 ARE 2 DIGIT BCD TOOL NUMBER
;
u_out41              IS OUT41   ;
u_out42              IS OUT42   ;
u_out43              IS OUT43   ;
u_out44              IS OUT44   ;
u_out45              IS OUT45   ;
u_out46              IS OUT46   ;
u_out47              IS OUT47   ;
u_out48              IS OUT48   ;
;
;  Jog panel LEDs
;
Brake_LED            IS OUT49   ;
Aux_2_LED            IS OUT50   ;
Aux_3_LED            IS OUT51   ;
Aux_4_LED            IS OUT52   ;
Aux_5_LED            IS OUT53   ;
Aux_6_LED            IS OUT54   ;
Aux_7_LED            IS OUT55   ;
Aux_8_LED            IS OUT56   ;
Aux_9_LED            IS OUT57   ;
Mist_LED             IS OUT58   ;
;
; OUT59 - OUT62 are on the CPU
; They should not be used.
;
u_out59              IS OUT59   ;
u_out60              IS OUT60   ;
u_out61              IS OUT61   ;
u_out62              IS OUT62   ;
;
Lubricant_low        IS OUT63   ; 0 = normal    1 = low lube
Drive_fault_out      IS OUT64   ; 0 = normal    1 = fault
Spindle_fault_out    IS OUT65   ; 0 = normal    1 = fault
```

```
Spindle_Auto_LED     IS OUT66   ; 0 = manual    1 = auto
Spindle_Dir_LED      IS OUT67   ; 1 = CCW       0 = CW
u_out68              IS OUT68   ;
u_out69              IS OUT69   ;
;
Aux_11_LED           IS OUT70   ;
Aux_12_LED           IS OUT71   ;
;
Coolant_Mode_LED     IS OUT72   ; 0 = manual    1 = auto
Flood_LED            IS OUT73   ; 0 = off       1 = on
Stop                 IS OUT75   ; 0 = run       1 = stop
PLC_op_signal        IS OUT76   ; 0 = run       1 = operation in progress
Feed_Hold_LED        IS OUT77   ;
Block_mode_LED       IS OUT78   ;
Rapid_override       IS OUT79   ;

;*****************************************************************************
;                            MEMORY DEFINITIONS
;*****************************************************************************
ZERO                 IS MEM1    ;  Used as a memory bit that is always 0
PC_Brake_LED         IS MEM2    ;
PC_Aux_2_LED         IS MEM3    ;
PC_Aux_3_LED         IS MEM4    ;
PC_Aux_4_LED         IS MEM5    ;
PC_Aux_5_LED         IS MEM6    ;
PC_Aux_6_LED         IS MEM7    ;
PC_Aux_7_LED         IS MEM8    ;
PC_Aux_8_LED         IS MEM9    ;
PC_Aux_9_LED         IS MEM10   ;
PC_Mist_LED          IS MEM11   ;
PC_Aux_11_LED        IS MEM12   ;
PC_Aux_12_LED        IS MEM13   ;
Range_reverse        IS MEM16   ;
Fault_Override       IS MEM17   ;
Lube_fault           IS MEM43   ;
Probe_Fault          IS MEM44   ;
Coolant_Fault        IS MEM47   ;
PC_PLC_RUNNING       IS MEM49   ;
Auto_Spin_Mode       IS MEM50   ;
PC_Lube_Fault        IS MEM51   ;
Brake_mode           IS MEM52   ;
Auto_Coolant_Mode    IS MEM53   ;
Man_spin_dir         IS MEM55   ;
Man_spin_mode        IS MEM56   ;
Spindle_dir          IS MEM57   ;
PC_Spindle_Fault     IS MEM58   ;
PC_OUT59             IS MEM59   ;
PC_OUT60             IS MEM60   ;
PC_OUT61             IS MEM61   ;
PC_OUT62             IS MEM62   ;
PC_Stop              IS MEM65   ;
PC_Block_mode        IS MEM66   ;
PC_Rapid_override    IS MEM67   ;
Already_run          IS MEM68   ;
Already_run2         IS MEM69
Real_Spin_Dir_Key    IS MEM73   ;
Last_Spin_Dir_Key    IS MEM74   ;
Spindle_Dir_Change   IS MEM75   ;
Last_Spin_Dir        IS MEM76   ;
Last_rapid_over_key  IS MEM77   ;
;
; MEM78 and MEM79 have pre-defined meanings and are
; used for communication between the PLC program
; and the CNC software to send the appropriate
; commands to a SPIN232, which uses either
; COM1 or COM2 port for communication.  When CNC Machine Parameter
; 31 = 1 or 2, the CNC software will look at MEM78 and MEM79
; to output the requested direction and enable information.
;
; Note that the SERVO3IO has a built-in SPIN232 interface.
;
```

```
Spin232_Enable       IS MEM78   ;
Spin232_Dir          IS MEM79   ;
;
;*************************************************************************
;*                              PROGRAM START
;*************************************************************************
;
; PLC_op_signal must be zero for motion control is to resume.
; Here, we halt processing if a spindle command
; is being executed (M3/M4) and Auto spindle mode is not selected or
; a coolant command (M7/M8) is being executed and Auto Coolant
; Mode is not selected.
;
; Setting PLC_op_signal is what causes the
; "Select Auto Spindle" or "Select Auto Coolant"
; messages to appear in the CNC message window and for
; program execution to pause until the signal is 0.
;
; It will be forced to zero if the Fault_Override bit is set.
; The XPLC program sets the Fault_Override bit which it
; determines from CNC Machine Parameter 177 settings.
;
PLC_op_signal        = ( ( ( M3 OR M4 ) AND Man_spin_mode ) OR
                       ( ( M7 OR M8 ) AND / Auto_Coolant_Mode ) ) AND
                       / Fault_Override
;
;/////////////////  UNICONSOLE-2 SPINDLE CONTROL //////////////
;
; A memory bit is used to simulate a "a real spindle direction key".
; Most of the jog panel inputs are on when the key is pressed and off
; when it is released.  On the Uniconsole-2 jog panel,
; the spindle direction is composed of two keys that use one input.
; The reason that the spindle direction logic is here in the
; standard program is because only it can control the spindle
; direction input (INP67).  Further, this bit cannot be set to a MEM
; bit by the XPLC prOgram and written by the standard program because
; of timing issues between the running of the XPLC and standard
; programs.
;
; The Uniconsole-2 jog panel firmware
; sends a message if the spindle direction key (CW or CCW) that is
; pressed is different than what was last sent. It initializes to CW.
;
;
Real_Spin_Dir_Key   = ( Spindle_Dir_Key & / Auto_Spin_Mode )
;
; This line determines if there is a spindle direction change.
; It does so on the following conditions:
;
;
; (1) The spindle direction key has changed and the mode is manual., i.e,
;     do not allow the spindle direction key to change the direction in
;     Automatic spindle mode since the direction in automatic spindle
;     mode is determined by an M3 or M4 command.
;
;                    OR
;
; (2) If in automatic spindle mode there has been a change between M3 and M4.
;
; If one of these conditions is true and the program is not already changing
; the direction, the Spindle_Dir_Change bit will flip the direction.
;
; The "OR / Already_run" is used to simulate a direction change at
; initial power up to get into a default CW state.
;
Spindle_Dir_Change  = ( ( ( ( Real_Spin_Dir_Key XOR Last_Spin_Dir_Key ) AND
                / Auto_Spin_Mode ) OR
                ( Auto_Spin_Mode & ( Spindle_Dir_Out XOR Last_Spin_Dir ) ) )
                AND / Spindle_Dir_Change ) OR / Already_run
Last_Spin_Dir_Key   = Spindle_Dir_Key
```

```
Spindle_Dir_Key     = Real_Spin_Dir_Key OR ( Auto_Spin_Mode AND M4 )
;
; Spindle_Dir is the commanded direction.
; Spindle_Dir_Out us the actual direction sent to the hardware.
; These two may be different if, for instance, the
; spindle is being reversed in low range.
;
Last_Spin_Dir       = Spindle_Dir_Out
Spin232_Dir         = Spindle_Dir_Out
;
; The Spindle_Dir_LED is set based upon the commanded direction, NOT
; the actual direction output to the hardware. As noted above,
; if Spindle_Dir_LED were set to Spinlde_Dir_Out then the LED would
; be incorrect if the spindle direction was being reversed because
; of being in low range.
;
Spindle_Dir_LED     = Spindle_Dir
;
; The XPLC program handles spindle Auto/Man changes and
; whether the spindle is enabled.
;
Spin232_Enable      = Spindle_Enable
Spindle_Auto_LED    = / Man_Spin_Mode
;
; Here the signal into CNC software that determines the spindle range
; is set. The CNC software uses this bit to determine the proper
; speed to display on the screen.
;
; Is is based upon the physical input and the
; Range_Reverse bit. The Range_Reverse bit is set by the XPLC
; program based upon CNC Machine Parameter 178 settings.  It is used to
; set the physical spindle range input to work as either a NO/NC input.
; More information is found in the XPLC program.
;
High_Low_Range       = Spindle_Range_In XOR Range_Reverse

;/////////////////////// UNICONSOLE-2 CPU7 COOLANT CONTROL  //////////
;
; The XPLC program is controlling the coolant functions.
; These lines just turn on the LEDs since XPLC cannot set them.
;
Flood_LED            = / Flood
Mist_LED             = Mist
Coolant_mode_LED     = Auto_Coolant_Mode
;
; The lines below will turn off the M-function
; requests if a programming is not running.
; For example, if while running a program with
; spindle CW on (M3) and/or Flood (M8),
; The spindle and coolant will be turned off
; when the program ends, or the CANCEL or ESC key is pressed.
;
M3                   = M3         AND CNC_Program_Running
M4                   = M4         AND CNC_Program_Running
M8                   = M8         AND CNC_Program_Running
M7                   = M7         AND CNC_Program_Running
;
; The logic below will signal to CNC software (via Drive_fault_out)
; a drive fault condition if the Servo_fault bit indicates a fault, or
; there already is a fault and the E_stop has not been pressed.
; This manner of logic is referred to as "latching" the fault.
; Once the fault has been "latched", the E_stop must be pressed AND
; the fault bit must indicate a no fault condition before the
; fault will be cleared.
;
; This fault can be disabled by setting of the Fault_override bit.
;
;
Drive_fault_out    =  ( Servo_fault OR Drive_fault_out AND / E_stop ) AND
                       / Fault_override
```

```
;
; The XPLC program will set PC_Lube_Fault and PC_Spindle_Fault
; to signal these faults.  Here, the standard program just
; reads the MEM bits and writes the actual OUT bits
; that signal the fault. These faults will be disabled
; if the Fault_Override bit is set.
;
Lubricant_low        = PC_Lube_Fault & / Fault_Override
Spindle_fault_out    = PC_Spindle_Fault & / Fault_Override
;
; And the same method of control noted above is used for controlling
; the Stop bit, Block_mode, and the AUX function key LEDs.
; In this way, there are no changes required to the standard
; program to control these states.
;
Stop                 = PC_Stop & / Fault_Override
Block_mode_LED       = PC_Block_Mode
Feed_Hold_LED        = Pause
Brake_LED            = Brake_mode
Aux_2_LED            = PC_Aux_2_LED
Aux_3_LED            = PC_Aux_3_LED
Aux_4_LED            = PC_Aux_4_LED
Aux_5_LED            = PC_Aux_5_LED
Aux_6_LED            = PC_Aux_6_LED
Aux_7_LED            = PC_Aux_7_LED
Aux_8_LED            = PC_Aux_8_LED
Aux_9_LED            = PC_Aux_9_LED
;
; Aux_11_LED, due to backward compatibility,
; is backwards. (1 = LED off, 0 = LED on)
;
Aux_11_LED           = / PC_Aux_11_LED
Aux_12_LED           = PC_Aux_12_LED
;*************************************************************************
;                              END OF PROGRAM
;*************************************************************************
```

The XPLC program:

```
; * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
; *
; *  File:    BASEXPC1.SRC
; *  Purpose: XPLC program for SERVO3IO/UNICONSOLE-2
; *
; *          Works in conjunction with STANDARD program BASECPU1.SRC
; *
; *          CNC Configuration Settings
; *              Jog Panel Type:   Uniconsole-2
; *              PLC Type:         Normal
; *
; *          CNC Parameter Settings
; *              P31   = 1 or 2 (for COM1 or COM2 spindle control)
; *
; *              P177  = 0.0  Normal behavior
; *                    = 1.0  Fault Override in effect
; *
; *              P178  Bit flags
; *                    LubeNONC         ; P178 Bit 0  (1)
; *                    SpindleNONC      ; P178 Bit 1  (2)
; *                    NoReverseSpindle ; P178 Bit 7  (128)
; *                    SpinRangeNONC    ; P178 Bit 9  (512)
; *
; *              P179  Lube timer settings  (see below for full explanation)
; *                    = 0  On when running a job
; *                    MMMSS   Off for MMM minutes, On for SS seconds
```

```
; *                            if SS = 0, On for at least MMM minutes
; *
; *
; *   () Support for probing and digitizing.
; *   () Handling of both NO/NC spindle drive faults and automatic reset.
; *   () Handling of both NO/NC low lube faults.
; *   () Handling of drive faults.
; *   () Support of two limits switches each for all three axes
; *   () Support for NO/NC spindle high/low range that can
; *      optionally reverse direction.
; *   () Support for drawbar unclamping.
; *   () Control of flood coolant in both automatic (M8) and manual modes.
; *   () Control of mist coolant in both automatuc (M7) and manual modes.
; *   () Automatic/manual control of a spindle brake.
; *   () Lube pump timing and control.
; *   () Fault override logic via parameter setting.
; *
; * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
;
;
```

Note: The listing for the XPLC program omits the definitions for INP1-INP80, OUT1-OUT80, and MEM1-MEM80 since they are identical to the standard program.

```
Spin_start          IS MEM101  ;
Spin_stop           IS MEM102  ;
Autostart           IS MEM104  ;
Autostop            IS MEM105  ;
Limit_tripped       IS MEM106  ;
Fault               IS MEM107  ;
Halt                IS MEM108  ;
Spindle_pause       IS MEM109  ;
Block_mode_key_hit  IS MEM110  ;
Spin_dir_key_hit    IS MEM112  ;
Rapid_over_key_hit  IS MEM115  ;
Brake_key_hit       IS MEM117  ;
LubeTimeExpired     IS MEM140  ;
ManFlood            IS MEM152  ;
ManMist             IS MEM153  ;
Man_Cool_Type       IS MEM162  ;
Coolant_Type        IS MEM163  ;
LubeMethod1         IS MEM164  ;
LubeMethod2         IS MEM165  ;
LubeMethod3         IS MEM166  ;
;
; In this program, CNC Machine Parameter 178 has been
; pre-defined as a bit mapped parameter that controls
; various aspects of the PLC program behavior.
;
; The LubeNONC, SpindleNONC, and SpinRangeNONC bits can be used to flip
; the respective switch input behavior between NO and NC. For example,
; the lube fault is considered to be a NC input, i.e., if the switch is
; electrically closed, then there is no fault.  When the switch opens,
; then this signals a fault.  If the lube fault signal needs to be NO,
; then the LubeNONC bit can be set.
;
; The NoReverseSpindle bit is used to indicate whether the spindle
; direction outputs should be reversed between high and low range.
; The default behavior of this program is to reverse direction as this
; convention follows a majority of the spindle gearing systems in use.
;
LubeNONC            IS MEM200  ; P178 Bit 0  (1)
SpindleNONC         IS MEM201  ; P178 Bit 1  (2)
uP178Bit2           IS MEM202  ; P178 Bit 2  (4)
uP178Bit3           IS MEM203  ; P178 Bit 3  (8)
uP178Bit4           IS MEM204  ; P178 Bit 4  (16)
uP178Bit5           IS MEM205  ; P178 Bit 5  (32)
uP178Bit6           IS MEM206  ; P178 Bit 6  (64)
```

```
NoReverseSpindle     IS MEM207  ; P178 Bit 7  (128)
uP178Bit8            IS MEM208  ; P178 Bit 8  (256)
SpinRangeNONC        IS MEM209  ; P178 Bit 9  (512)
uP177Bit10           IS MEM210  ; P178 Bit 10 (1024)
uP178Bit11           IS MEM211  ; P178 Bit 11 (2048)
uP178Bit12           IS MEM212  ; P178 Bit 12 (4096)
uP178Bit13           IS MEM213  ; P178 Bit 13 (8192)
uP178Bit14           IS MEM214  ; P178 Bit 14 (16384)
uP178Bit15           IS MEM215  ; P178 Bit 15 (32768)
;
;*************************************************************************
;                            STAGE DEFINITIONS
;*************************************************************************

InitialStage         IS STG1
MainStage            IS STG2
LoadCNC10Parameters   IS STG3
Lubemonitor          IS STG6

;*************************************************************************
;                             WORD DEFINITIONS
;*************************************************************************

LubeWord             IS W12
LubeOnTime           IS W13
LubeOffTime          IS W14
;LubeOffX100         IS W15
P178Bits             IS W16
P177Bits             IS W17
TotalTime            IS W18
PresetTime           IS W19

;*************************************************************************
;                            TIMER DEFINITIONS
;*************************************************************************

LubeOffT             IS T2
AccumulatingTMR      IS TMR3
LubeOnT              IS T4

;*************************************************************************
;                          PD (One-Shot) DEFINITIONS
;*************************************************************************
CM_1Shot             IS PD1      ; Coolant Auto/Manual key pressed
MK_1Shot             IS PD2      ; Mist coolant key pressed
SD_1Shot             IS PD3      ; Spindle Direction
BM_1Shot             IS PD4      ; Block Mode key pressed
ATM_1Shot            IS PD5      ; Coolant Auto To Manual change
ES_1Shot             IS PD6      ; E_stop release
M10_1Shot            IS PD7      ; M10
M11_1Shot            IS PD8      ; M11
FK_1Shot             IS PD9      ; Flood
BRK_1Shot            IS PD10     ; Aux1
;
;  Lube Timing Related One-Shots
;
PR_1Shot             IS PD11     ; started CNC program running
NPR_1Shot            IS PD12     ; stopped CNC program running
OffTimerExpired      IS PD13     ;
OnTimerExpired       IS PD14     ;

;*************************************************************************
;*                              PROGRAM START
;*************************************************************************

;----------------
 InitialStage
;----------------
;
; InitialStage (STG1) is on at PLC initialization.
; Here we set PC_PLC_Running (MEM49) to indicate to
```

```
; the standard program that XPLC is being used also.
;
IF !ZERO THEN  SET PC_PLC_Running,
               SET LubeMonitor,
               SET LoadCNC10Parameters,
               JMP MainStage

;---------------------
   LoadCNC10Parameters
;---------------------
;
; Here is where the processing of the pre-defined CNC
; Machine Parameters takes place.  Note that this STG
; remains on so that changes to the CNC Machine
; Parameters can take effect without rebooting.
;
; Read the value from P179 in MMMSS format
; and break it into separate MMM and SS parts
;
; Remember that mathematical operations work
; with integers.  When dividing, there are no remainders
; and no explicit rounding up.
;
; For example, 5 / 2 = 2, 100 / 99 = 1, 99 / 100 = 0
;
;  LubeWord      = HHHMM
;  HHHMM / 100   =   HHH (LubeOffTime)
;  HHH * 100     = HHH00 (LubeOffTime * 100)
;
;                  HHHMM (LubeWord)
;                - HHH00 (LubeOffTime * 100)
;                -------
;                =    MM (LubeOnTime)
;
IF !ZERO THEN  LP9 LubeWord,
               LubeOffTime = LubeWord / 100,
               LubeOnTime  = LubeWord - LubeOffTime * 100
;
; Convert LubeOffTime to minutes
; and    LubeOnTime  to seconds
; and set timer PRESET values
;
; TMR values are in 0.01 second increments, i.e.,
; there are 100 of them in every second.
;
IF !ZERO THEN  LubeOffTime = LubeOffTime * 100 * 60,
               LubeOnTime  = LubeOnTime * 100,
               LubeOffT    = LubeOffTime,
               LubeOnT     = LubeOnTime
;
;
; Initialize Bit reversals according to param 178
; Memory Bits 200-215 hold the values in parameter 178
;
;  P178Bits            P178Bits / 256
;  FEDCBA9876543210    00000000FEDCBA98
;
IF !ZERO THEN  LP8 P178Bits,
               WTB P178Bits MEM200,
               P178Bits = P178Bits / 256,
               WTB P178Bits MEM208,
               LP7 P177Bits

;
; Set MEM bit to communicate with standard PLC whether
; the spindle range input is NO/NC and whether to Override faults.
;
IF SpinRangeNONC THEN (Range_reverse)
IF P177Bits == 1 THEN (Fault_Override)
```

```
;------------------------------
     LubeMonitor
;------------------------------
;
; These Lube pumps are set by CNC10 Machine Parameter 179,
; where the value is between 0 - 65535
; and is formatted as MMMSS
; Where MMM is the Off Time in minutes
; and SS is the On Time in seconds.
;
;  If SS == 0 and MMM != 0, then Method 1 is used for control.
;  If SS != 0 and MMM != 0, then Method 2 is used for control.
;  If SS == 0 and MMM == 0, then Method 3 is used for control.
;
; METHOD 1
;
;  On the start of CNC_program_running,
;  the lube pump turns on.
;
;  The lube pump is turned off when E_stop or
;  a low lube condition exists or when
;  a program has NOT been running for MMM minutes.
;
;  This type of control is intended
;  for lube pumps that have internal timers that either
;  lube immediately when power is applied and then start timing, or
;  for pumps that wait until power has been on for the set time
;  before pumping.  The former type of pump will run out of lube oil quickly
;  on short jobs if lube is only applied while CNC_program_running.
;  The latter type of pump will never lube on short job runs if lube power is
;  is only applied while CNC_program_running. A short job is defined
;  as one which completes in less time than the internal lube timer
;  is set to.
;
;  Example 1.
;   The lube pump has a timer and it is set to lube every 30 minutes.
;   Set Machine Parameter 179 = 3500.  Note that this time (MMM) should
;   be longer than the setting of the lube timer.
;
; METHOD 2
;
;  The lube turns on for SS seconds every MMM minutes.
;
;  Example 2.
;   To set the lube pump power to come on for 5 seconds
;   every 10 minutes, set P179 = 1005.
;
;  Example 3.
;   To set the lube pump power to come on for 30 seconds
;   every 2 hours, set P179 = 12030
;
; METHOD 3
;
;  The lube comes on whenever a program is being run
;  (CNC_program_running is on).  This includes MDI mode.
;
;;
;
;  Set MEM bits to indicate the method used
;
IF LubeOnTime == 0 & LubeOffTime != 0 THEN (LubeMethod1)
IF LubeOnTime != 0 & LubeOffTime != 0 THEN (LubeMethod2)
IF LubeOnTime == 0 & LubeOffTime == 0 THEN (LubeMethod3)

IF  Cnc_program_running THEN (PR_1Shot)
IF !CNC_program_running THEN (NPR_1Shot)
;
; Here the accumulated time that a CNC program has been
; running is programmed.  When the Accumulated time
; has reached the OffTime (MMM), then start the On timer.
```

```
; The Accumulated time is reset when the On timer expires.
;
IF NPR_1Shot THEN TotalTime = TotalTime + AccumulatingTMR
IF CNC_program_running & !LubeOnT THEN (AccumulatingTMR)
IF TotalTime + AccumulatingTMR > LubeOffTime THEN (LubeOnT), (LubeTimeExpired)
IF LubeOnT THEN TotalTime = 0
IF  LubeTimeExpired THEN (OffTimerExpired)
IF !LubeTimeExpired THEN (OnTimerExpired)
;
;
; Turn the lube pump ON under one the following conditions:
;
; (1) the CNC program has started running and using Method 1 or Method3
; (2) the OffTime is up and using Method2.
;
IF (PR_1Shot & (LubeMethod1 | LubeMethod3)) |
   (OffTimerExpired & LubeMethod2) THEN SET Lube
;
;
;  When lube is on, start keeping track of the off time
;  for use in Method 2.
;
IF Lube THEN (LubeOffT)
;
;  Turn the lube pump OFF under one of these conditions:
;
;  (1) Method1 and the Off timer has expired and a
;     CNC program is not running.
;  (2) Method2 and the On timer expired.
;  (3) Method3 and a CNC program is no longer running.
;  (4) A low lube fault has occured.
;  (5) E_stop is pressed.
;
IF (LubeOffT & !CNC_Program_Running & LubeOnTime == 0 & LubeOffTime != 0) |
   (OnTimerExpired & LubeOnTime != 0 & LubeOffTime != 0) |
   (NPR_1Shot & LubeOffTime == 0) |
   (Lube_Fault_In XOR LUBENONC) | E_stop THEN RST Lube

;---------------
  MainStage
;---------------

;
;//////////////////// UNICONSOLE-2 XPLC SPINDLE CONTROL  ////////
IF !Man_Spin_Mode | !Already_run THEN (Auto_Spin_Mode)
IF M3 | M4    THEN (AutoStart)
IF !AutoStart THEN (AutoStop)

;-------------------------------------------------
;  Select between auto and manual spindle mode
;-------------------------------------------------
IF Spindle_Mode_Switch THEN (SD_1Shot)
IF (SD_1Shot XOR Man_Spin_Mode) THEN (Man_Spin_Mode)

;-------------------------
;     Start the spindle
;-------------------------
IF  (Man_spin_mode & Spindle_start_key) |
    (Auto_spin_mode & Autostart & !Spindle_pause) THEN (Spin_Start)

IF (Spin_Start & !Probe_not_detected) THEN (Probe_Fault)

;-------------------------
;     Pause the Spindle
;-------------------------

IF  Auto_spin_mode & ((Spindle_pause & !Spindle_start_key &
    CNC_program_running) | (!Spindle_pause & Spindle_stop_key
    & Spindle_Enable)) THEN (Spindle_pause)

;-------------------------
```

```
;     Stop the spindle
;-------------------------

IF  Spindle_Stop_key | Stop | Limit_tripped |
   (Auto_spin_mode & AutoStop) THEN (Spin_Stop)

IF (Man_spin_mode & Spindle_Dir_key) THEN (Man_spin_dir)

IF (Auto_spin_mode & M4) |
   (Man_spin_mode & Man_spin_dir) THEN (Spindle_dir)
;
;
; The direction of the physical ouput is reversed if
; the NoReverseSpindle Bit is off and the Spindle_range_in is:
;
;   open   (1) and SpinRangeNONC is RST (0)
;                OR
;   closed (0) and SpinRangeNONO is SET (1)
;
; NoReverseSpindle and SpinRangeNONC are SET/RST
; depending upon the value of P178 in CNC10 Machine Parameters.
;
; Note that even if the spindle is NOT to be reversed in low range
; that the Spindle_range_in is needed by CNC10 to determine
; the correct speed to display on the screen.
;
; Application Requirements:
;
; (1a)  The spindle must reverse in low range.
; (1b)  The spindle must NOT reverse in low range.
;
; (2a)  There is a NC HI-LO switch connected to Spindle_range_in
; (2b)  There is a NO HI-LO switch connected to Spindle_range_in
;
; (3a) The HI-LO switch is tripped when the gear change lever (or switch)
;      is in the LO position.
; (3b) The HI-LO switch is tripped when the gear change lever (or switch)
;      is in the HI position.
;
; Application P178 Settings:
;
; (1a)
;   NoReverseSpindle 0
; (1b)
;   NoReverseSpindle 1
;
; (2a & 3a) or (2b & 3b)
;   SpinRangeNONC   0
;
; (2a & 3b) or (2b & 3a)
;   SpinRangeNONC   1
;
IF Spindle_dir ^ (!NoReverseSpindle &
     (Spindle_range_in ^ SpinRangeNONC)) THEN (Spindle_Dir_Out)

IF (Spindle_Enable | Spin_start) & !Spin_Stop &
   Probe_not_detected THEN (Spindle_Enable)

IF !Spindle_Enable & !Tool_Release & !ZERO_speed THEN (Drawbar_Sol)

;//////////////  UNICONSOLE-2 XPLC COOLANT CONTROL  ////////////////////////

;-------------------------------------------
;     Toggle Auto Manual Coolant Mode
;-------------------------------------------

IF Coolant_mode_switch THEN (CM_1Shot)
IF Auto_Coolant_Mode ^ CM_1Shot OR ! Already_run THEN (Auto_Coolant_Mode)

;
```

```
; Flood coolant control
;   Toggle flood coolant on and off if coolant mode is manual
;   When switching from Auto to manual mode, turn off flood coolant
;
IF !Auto_Coolant_Mode THEN (ATM_1Shot)
IF Flood_key THEN (FK_1Shot)
IF (ManFlood ^ (!Auto_Coolant_Mode & FK_1Shot)) & !ATM_1Shot THEN (ManFlood)
IF !Stop & ((M8 & Auto_Coolant_Mode) | (ManFlood & !Auto_Coolant_Mode))
   THEN (Flood)
;
; Mist coolant control
;   Toggle mist coolant on and off if coolant mode is manual
;   When switching from Auto to manual mode, turn off mist coolant
;
IF Mist_key THEN (MK_1Shot)
IF (ManMist ^ (!Auto_Coolant_Mode & MK_1Shot)) & !ATM_1Shot THEN (ManMist)
IF !Stop & ((M7 & Auto_Coolant_Mode) | (ManMist & !Auto_Coolant_Mode))
   THEN (Mist)

;/////////////////////////////////////////
;
;       FAULT HANDLING
;
;/////////////////////////////////////////
;
; If FLT is not zero, there is an internal error
; in the execution of the XPLC program.
;
IF FLT != 0 THEN SET PLC_Fault_Out
;
; If the lube is low set the lube alarm
;
IF (Lube_Fault_In ^ LUBENONC) |
   (PC_Lube_Fault & !E_stop) THEN (PC_Lube_Fault)
;
; Check for tripped limits
;
IF (X_plus | X_minus | Y_plus | Y_minus | Z_plus | Z_minus)
   THEN (Limit_tripped)
;
;
; Set and clear spindle and servo faults
; The fault is latched and cleared when the fault
; signal is removed and E_stop is pressed.
;
; The  &Already_run2 is used to screen out the intialization passes
; which may initially indicate a fault.
;
IF ((!Spindle_ok ^ SpindleNONC) |
   (Spindle_fault_out & !E_stop)) & Already_run2
   THEN (PC_Spindle_Fault)
;
; Check for PLC fault
; The fault is latched and cleared when the fault signal
; is cleared and E-stop is pressed.
;
IF (!PLC_OK | (PLC_fault_out & !E_stop)) & !Fault_Override
   THEN (PLC_Fault_Out)
;
; If there's a fault condition then Stop
; A running program is not stopped because of
; low lube.  However, the error will occur as soon
; as the running job is stopped or completed.
;
IF PLC_fault_out | Spindle_fault_out | Drive_fault_out |
  (!CNC_program_running & Lubricant_low) THEN (Halt)

IF (Halt | Fault) & !E_stop THEN (Fault)

IF  Fault | E_stop | Coolant_Fault THEN (PC_Stop)
;
```

```
; Reset inverter (VFD) if there is a fault and the E_stop is pressed.
;
IF (!Spindle_ok XOR SpindleNONC) & E_stop THEN (VFD_reset)

;/////////////////////  SINGLE/BLOCK MODE CONTROL   //////////////
;
; Toggle between single/block mode
;
IF Block_mode_key THEN (BM_1Shot)
IF (!PC_Block_mode & BM_1Shot & !CNC_program_running) |
   (PC_Block_mode & !BM_1Shot)
  THEN (PC_Block_mode)

;/////////////////////// AUTO SPINDLE BRAKE CONTROL   /////////////
;
; Set auto spindle brake mode
; | !Already_run is used to turn on Auto Brake mode at initialization.
;
IF Brake_key THEN (BRK_1Shot)
IF BRK_1Shot ^ Brake_mode | !Already_run THEN (Brake_Mode)
IF Brake_mode & !Spindle_Enable THEN (Brake)
;
;  Keep these at the end of the Main Stage and in this order
;  These are used for power up conditions
;
IF Already_run THEN (Already_run2)
IF Already_run OR ! Already_run THEN (Already_run)
```

Application Examples

All the examples in the standard manual can be converted to an XPLC program by identifying which bits the XPLC has control of and converting these lines of logic to XPLC format. The definition of plc bits should be the same in both the standard and XPLC program.

Remember the standard programs follow this pattern:

<plc_bit> = <boolean_expression>

To change it into an equivalent XPLC program statement, rewrite is as

IF *<boolean_expression>* THEN (*<plc_bit>*)

and convert any '/' (NOT) symbols to '!'.

As an example, the standard program

```
Brake_key_hit       = Brake_key AND / Last_brake_key
Last_brake_key      = Brake_key
Brake_mode          = ( Brake_mode XOR Brake_key_hit ) OR / Already_run
Brake               = Brake_mode AND / SpindleRelay
```

would be converted to the XPLC program:

```
IF Brake_key AND ! Last_brake_key                   THEN (Brake_key_hit)
IF Brake_key                                        THEN (Last_brake_key)
IF ( Brake_mode XOR Brake_key_hit ) OR ! Already_run THEN (Brake_mode)
IF Brake_mode AND ! SpindleRelay                    THEN (Brake)
```

Using an Aux key to toggle an output.

Assume that for the base programs listed above that we wish to modify the programming so that the AUX2 key on the jog panel toggles a light that is connected to OUT6. We desire the AUX2 LED on the jog panel to be on when the light is on. In this case, we can make all the changes solely to the XPLC program as such:

Definitions

```
LIGHT             IS OUT6
AUX2_KEY_PRSSED   IS PD50
```

Program

```
IF AUX_2_KEY THEN (AUX2_KEY_PRESSED)
IF LIGHT ^ AUX2_KEY_PRESSED THEN (LIGHT), (PC_AUX_2_LED)
```

Using an Aux key to momentarily turn on an ouput.

Assume that in the example above, we wish the Light and LED to be on only while the AUX2 key is being pressed. Here is how it could be done:

Definitions

```
LIGHT             IS OUT6
```

Program

```
IF AUX_2_KEY THEN (LIGHT), (PC_AUX_2_LED)
```

Implementing a Hass Indexer

A Haas Indexer can be can interfaced to a CNC system using one input, one output, and a custom M-code. The example will use an M12 custom M-function to perform the indexing operation. When the M12 command is executed in an M&G code program, OUT6 will turn on until the Haas Indexer finished signal (wired to INP6) closes

For this example, there is a need to change the standard PLC program so that the M12 command will be cancelled if for some reason the job stops running.

Definitions

```
Index_finished   IS INP6
Indexer_Out      IS OUT6
M12              IS INP38
```

Program

```
IF M12 THEN (Indexer_Out)
```

Custom M12 (CNC10.M12)

```
M94/6         ; turn on INP38
M101/6        ; wait for INP6 to close
M95/6         ; turn off INP38
```

Standard PLC program changes

```
M12 = M12 AND CNC_Program_running
```

Adding an AUX key to the Haas Indexer example.

This example will add to the above Haas Indexer example by allowing an AUX key to work the indexer. The AUX key indexing will work provided that a CNC job is not currently running.

In addition to the above XPLC program we need:

```
AUX1_HIT            IS PD1
AUX1_KEY            IS INP49
AUX_Index           IS MEM20


IF AUX1_KEY THEN (AUX1_HIT)
IF AUX1_HIT & !CNC_Program_running THEN SET Aux_Index

IF M12 | Aux_index THEN (Indexer_Out)
IF !Index_finish | Stop THEN RST Aux_index
```

A Flashing Light

The following example can be used to turn an output on and off at a certain frequency. We will assume a light is attached to this output. A memory bit will be used to turn on the flashing light.

```
LIGHT       IS OUT1
DO_FLASH    IS MEM10
FLASH_LIGHT IS STG2

IF DO_FLASH THEN (FLASH_LIGHT)

;---------------------
    FLASH_LIGHT
;---------------------
IF !T1 THEN T1=200, (T1)
IF TMR1 < 100 THEN (LIGHT)
```

In this example, the LIGHT will be on for one second and off for one second.

Programming UP/DOWN counters

While the XPLC language has no direct support for UP or DOWN counters, the same functionality can be achieved using the following programming. The basic technique is to use a one shot PD coil to increment or decrement a word memory location.

```
COUNTER_INPUT     IS INP1
COUNTER_RESET     IS MEM2
COUNT_REACHED     IS MEM3
COUNTER_1SHOT     IS PD4
COUNTER_VALUE     IS W5
COUNTER_LIMIT     IS W6
COUNTER_SETUP     IS STG7
MONITOR_COUNT     IS STG8

;--------------------------------------
         COUNTER_SETUP
;--------------------------------------
IF 1==1 THEN COUNTER_LIMIT = 10, JMP MONITOR_COUNT
;--------------------------------------
         MONITOR_COUNT
;--------------------------------------
IF COUNTER_INPUT THEN (COUNTER_1SHOT)
IF COUNTER_1SHOT THEN COUNTER_VALUE = COUNTER_VALUE + 1
IF COUNTER_VALUE >= 10 THEN (COUNT_REACHED)
IF COUNTER_RESET THEN COUNTER_VALUE = 0
```

This example omits the need to turn on the COUNTER_SETUP stage during program initialization. It also does not show how other parts of the program would respond to the COUNT_REACHED.

Turning a truth table into a boolean expression.

INP1	INP2	INP3	OUT1
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

```
IF

(!INP1 & !INP2 &  INP3) |
(!INP1 &  INP2 & !INP3) |


( INP1 & !INP2 &  INP3) |
( INP1 &  INP2 & !INP3)

THEN (OUT1)
```

The example above shows a truth table and the boolean expression that would be used in a program to implement it. By close examination, one can see the pattern that emerges. Every row in the truth table where OUT1 is 1 becomes a line of logic that is OR'ed with the other rows where OUT1 is 1. Each row is the ANDing of each INP. If the truth table was a 0 for an INP, then in the boolean expression it would have a ! (NOT) preceding it.

In RLL, the above program would be written as such:

Turning a truth table into a boolean expression as above will often result in an expression that is more complex than it needs to be. There are techniques that can be used to simplify these expressions but they are beyond the scope of this manual. Note that this particular expression can be written as:

```
IF INP2 XOR INP3 THEN (OUT1)
```

That's right. In this example INP1 has no effect on the resulting logic.

Converting a series of inputs to a number.

The conversion of a series of inputs into a useable number in a program is typically found when interfacing to an automatic tool changer.

For this example, suppose there are six inputs that represent a tool number and we wish to keep track of the current tool number. The inputs form a binary number.

```
TOOL_INPUT1   IS INP1
TOOL_INPUT2   IS INP2
TOOL_INPUT3   IS INP3
TOOL_INPUT4   IS INP4
TOOL_INPUT5   IS INP5
TOOL_INPUT6   IS INP6
TOOL_NUMBER   IS W2
TOOL_MONITOR  IS STG3


;-----------------------------------------
             TOOL_MONITOR
;-----------------------------------------

IF 1==1 THEN TOOL_NUMBER = 0
IF TOOL_INPUT1 THEN TOOL_NUMBER = TOOL_NUMBER + 1
IF TOOL_INPUT2 THEN TOOL_NUMBER = TOOL_NUMBER + 2
IF TOOL_INPUT3 THEN TOOL_NUMBER = TOOL_NUMBER + 4
IF TOOL_INPUT4 THEN TOOL_NUMBER = TOOL_NUMBER + 8
IF TOOL_INPUT5 THEN TOOL_NUMBER = TOOL_NUMBER + 16
IF TOOL_INPUT6 THEN TOOL_NUMBER = TOOL_NUMBER + 32
```

If the inputs were representing a BCD tool number then the last two lines in the TOOL_MONITOR stage would be

```
IF TOOL_INPUT5 THEN TOOL_NUMBER = TOOL_NUMBER + 10
```

```
IF TOOL_INPUT6 THEN TOOL_NUMBER = TOOL_NUMBER + 20
```

Another consideration to take into account when using this technique is that PLC inputs are not received into the copy buffer in parallel from the hardware. In other words, they are received one at a time or one after another. If the snapshot of the inputs is taken in the middle of this updating, the calculated tool number will be wrong. For example: when changing from a 7 (0111 binary) to 8 (1000 binary), there are four inputs that have changed. If a pass of program execution happens before all four bits have been updated then the number will be calculated incorrectly. Whether this will actually be a problem depends upon other factors of the interface.

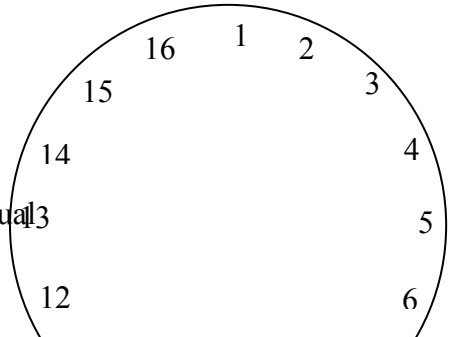
One method that can be used as a work-around to the above problem is to not allow the number to be updated unless it is within one of the last number calculated. For example, if the number is 3, do not change the number until it reaches 4, assuming the numbers were increasing.

Shortest Path Calculation for Circular Indexing

Automatic tool changers typically consist of a certain number of tools arranged in a circular carousel as shown in the figure below. Often, the carousel can be indexed forward and reverse. The following example shows some programming that can be used for calculating the shortest distance and the direction to index.

This example will assume a motor connected to reversing contactors. The motor control will be through a SPST relay that acts as a motor on/off switch that is wired to a SPDT relay that is wired to the FWD/REV contacts of the reversing contactors. The example below assumes PLCIO2 hardware is being used. It is also assumed that there is a custom M6 program to send the tool number (with M107) and turn on INP38 to start the process.

```
M6                IS INP38 ; (ToolChange) Map to M94/6  M95/6


MotorOnRly        IS OUT31 ;
MotorDirRly       IS OUT32 ; off = FWD, on = REV

M6_1SHOT          IS PD1   ;

ToolNumber        IS W1    ; the current carousel position
WantedTool        IS W2    ; the carousel position to index to
NumbeOFTools      IS W3    ; the number of carousel positions
MotorDir          IS W4    ; the motor direction (1 = forward, -1 = backward)
Distance          IS W5    ; the number of indexes required

InitialStage      IS STG1  ;
CalcDistAndDir    IS STG2  ;
Calc_B1           IS STG3  ;
Calc_B2           IS STG4  ;
IndexMotor        IS STG5  ;
IndexFinished     IS STG6  ;


;-----------------------------
     InitialStage
;-----------------------------
IF 1==1 THEN NumberOfTools = 16,
             ToolNumber = 1,
```



```
            JMP MainStage

;-----------------------------
     MainStage
;-----------------------------
IF M6 THEN (M6_1SHOT)
IF M6_1SHOT THEN LDT WantedTool
                 SET CalcDistAndDir

;-----------------------------
     CalcDistAndDir
;-----------------------------
IF WantedTool == ToolNumber
  THEN JMP IndexFinished

IF WantedTool > ToolNumber
  THEN Distance = WantedTool - ToolNumber,
  JMP Calc_B1

IF ToolNumber > WantedTool
  THEN Distance = ToolNumber - WantedTool,
  JMP Calc_B2


;-----------------------------
    Calc_B1
;-----------------------------
IF Distance <= NumberOfTools/2
  THEN MotorDir = 1

IF Distance > NumberOfTools/2
  THEN MotorDir = -1,
       Distance = NumberOfTools - Distance

IF 1==1 THEN JMP IndexMotor

;-----------------------------
    Calc_B2
;-----------------------------
IF Distance <= NumberOfTools/2
  THEN MotorDir = -1

IF Distance > NumberOfTools/2
  THEN MotorDir = 1,
  Distance = NumberOfTools - Distance

IF 1==1 THEN JMP IndexMotor

;-----------------------------
    IndexMotor
;-----------------------------
;
; It is assumed here that another stage is monitoring and updating ToolNumber
;
IF 1==1 THEN (MotorOnRly)
IF MotorDir < 0 THEN (MotorDirRly)
IF ToolNumber == WantedTool THEN JMP IndexFinished

;-----------------------------
    IndexFinished
;-----------------------------
;
;  At this point, the tool carousel has been indexed to the new location.
;  There should be some handshaking with the custom M6 to signal the end of the process
;
IF 1==1 THEN RST IndexFinished
```

Accumulating Timers

Like counters, XPLC programs do not directly support accumulating timers. Accumulating timers are timers whose current value does not reset to zero if the timer input is false. Accumulating timers have a separate input that is used to reset the current value.

```
ATIMER_INPUT    IS INP1
ATIMER_RESET    IS INP2
ATIMER_UP       IS MEM20
INPUT_OFF       IS PD1
ACCUMTMR        IS TMR1
ATIMER_PRESET   IS W1
TOTAL_TIME      IS W2
SUBTOTAL        IS W3
ATIMER_SETUP    IS STG1
ATIMER_MONITOR  IS STG2

;--------------------------------------
              ATIMER_SETUP
;--------------------------------------
IF 1==1 THEN ATIMER_PRESET = 100 * 60 * 15, JMP ATIMER_MONITOR



;--------------------------------------
              ATIMER_MONITOR
;--------------------------------------
IF !ATIMER_INPUT THEN (INPUT_OFF)
IF INPUT_OFF THEN SUBTOTAL = SUBTOTAL + ACCUMTMR
IF ATIMER_INPUT & !ATIMER_RESET  THEN (ACCUMTMR)
IF ATIMER_RESET  THEN SUBTOTAL = 0
IF 1==1 THEN TOTAL_TIME = SUBTOTAL + ACCUMTMR
IF TOTAL_TIME >= ATIMER_PRESET THEN (ATIMER_UP)
```

The example above programs a 15 minute accumulating timer. Whenever ATIMER_INPUT is on, the timer will start timing. When ATIMER_INPUT is off, the timing stops timing but the TOTAL_TIME contains the accumulated time. If the TIMER_RESET is on, the TOTAL_TIME is reset. When the accumulating timer reaches the preset value (programmed above as 15 minutes), the ATIMER_UP bit will be on.

Communication via CNC Machine Parameters

CNC Machine Parameters 170-179 are passed to the PLC program when they are changed using G10 codes or when the parameters are saved. The value of these parameters can be read into XPLC word memory using the LP0 – LP9 commands.

```
IF 1==1    THEN LP0 W1
IF W1 == 1 THEN (OUT1)
```

In the above program, if CNC Machine Parameter 170 was set to 1.0, then OUT1 would turn on.

In a G-code program, P170 could be turned on/off as such:

```
G10 P170 R1   ; turn on OUT1
G10 P170 R0   ; turn off OUT1
```

Note that when a G-code program is being run that the program is actually being parsed ahead of what is currently executing. Thus, if a line such as G10 P170 R1 were located near the end of the program, it could happen that the parameter change could take effect almost immediately after starting a job. The way to halt processing until a certain point is reached is to proceed it with a read of a PLC bit variable as such:

```
if #6001      ;  do not parse the program any further than this line.
G10 P170 R1   ;  turn on OUT1
```

In this way, the parameter change will not occur until the G-code execution reaches this point. #6001 is a reference to INP1. It is possible to use other values as well.

A CNC Machine Parameter can also be used to bitmap values. Since the range of valid values in the Machine Parameters is 0-65535, a single parameter can be used to turn on and off up to sixteen different bits. Refer to the program BASEXPC1.SRC which demonstrates this technique and allows a CNC Machine Parameter to configure certain inputs to work with either normally open (NO) or normally closed (NC) switch inputs without having to rewrite the program.

Using more than 16 custom M-codes

Custom M-codes usually involve using a combination of M94/M95 commands to turn on/off INP33-INP48. The PLC program will then look at INP33-INP48 to turn on/off an output. There is a problem, however, when more than 16 custom M-codes are required. Remember also that at least five are predefined for spindle, coolant, and clamp control. Presented below is a technique used to get more than 16. The basic idea is to use so many lines to form a binary M-code number and another to act as a strobe. Then the individual M-codes setup the binary number and then turn on a strobe. The following example outlines everything needed for 32 custom M-codes, using just six of the INP33-INP48 bits.

```
MFUN_STROBE      IS INP38  ; Set with M94/6
MFUN_BIT0        IS INP39  ; Set with M94/7
MFUN_BIT1        IS INP40  ; Set with M94/8
MFUN_BIT2        IS INP41  ; Set with M94/9
MFUN_BIT3        IS INP42  ; Set with M94/10
MFUN_BIT4        IS INP43  ; Set with M94/11


MFUN_1SHOT       IS PD1

MFUN_VALUE       IS W1

DO_M_FUCNTION    IS STG2
DO_MFUN_0        IS STG100
DO_MFUN_1        IS STG101
…
…
…
DO_MFUN_31       IS STG131


;
;  Locate this code somewhere in the main loop
```

```
;

IF MFUN_STROBE THEN (MFUN_1SHOT)
IF MFUN_1SHOT THEN SET DO_M_FUCNTION

;-----------------------
     DO_M_FUNCTION
;-----------------------

IF 1==1 THEN MFUN_VALUE = 0
IF MFUN_BIT0 THEN MFUN_VALUE = MFUN_VALUE + 1
IF MFUN_BIT1 THEN MFUN_VALUE = MFUN_VALUE + 2
IF MFUN_BIT2 THEN MFUN_VALUE = MFUN_VALUE + 4
IF MFUN_BIT3 THEN MFUN_VALUE = MFUN_VALUE + 8
IF MFUN_BIT4 THEN MFUN_VALUE = MFUN_VALUE + 16

;
;  At this point MFUN_VALUE will be 0-31
;
IF MFUN_VALUE == 0  THEN JMP DO_MFUN_0
IF MFUN_VALUE == 1  THEN JMP DO_MFUN_1
IF MFUN_VALUE == 2  THEN JMP DO_MFUN_2

…
…
…

IF MFUN_VALUE == 30 THEN JMP DO_MFUN_30
IF MFUN_VALUE == 31 THEN JMP DO_MFUN_31

;--------------------
     DO_MFUN_0
;--------------------
;
; Place actions in here
;

;--------------------
     DO_MFUN_1
;--------------------
;
; Place actions in here
;

…
…
…

;--------------------
     DO_MFUN_31
;--------------------
;
; Place actions in here
;
```

The preceding program would be used with the following CNC10.M?? codes.

CNC10.M00

```
M95/7    ;\
M95/8    ;-\
M95/9    ;-->  Set up binary pattern 00000
M95/10   ;-/
M95/11   ;/

M94/6    ;
G4 P0.2  ;     Trigger MFUN_1SHOT
M95/6    ;
```

CNC10.M01

```
M94/7    ;\
M95/8    ;-\
M95/9    ;-->  Set up binary pattern 00001
M95/10   ;-/
M95/11   ;/

M94/6    ;
G4 P0.2  ;     Trigger MFUN_1SHOT
M95/6    ;

…
…
…
```

CNC10.M31

```
M94/7    ;\
M94/8    ;-\
M94/9    ;-->  Set up binary pattern 11111
M94/10   ;-/
M94/11   ;/

M94/6    ;
G4 P0.2  ;     Trigger MFUN_1SHOT
M95/6    ;
```

Note that the actual M-codes used would typically be ones that do not already have a pre-defined meaning. Some M functions that do not have pre-defined meanings in milling software are M12-M24, M27-M29, M31-M38, and M40-M90.

Another Way to Control Outputs in M&G code Programs

Another technique that can be used to control PLC outputs from within an M&G code program is to combine the use of a bitmapped CNC Machine Parameter with appropriate subprograms.

Assume that CNC Machine Parameter 170 is used to hold the states of 16 individual bits. The XPLC program fragment below maps P170 into MEM100-MEM115 and then MEM100-MEM115 are used to control OUT1-OUT15, and OUT29. OUT16 was skipped because it is a PLC fault indicator and OUT17-OUT28 are reserved for 12-bit spindle speed.

```
IF 1==1 THEN LP0 W1,         ; Load P170 into W1
             WTB W1 MEM100,  ; Write the lower 8 bits to MEM100-MEM107
             W1 = W1 / 256,  ; Shift the upper 8 bits to the lower 8 bits
             WTB W1 MEM108   ; Write the 8 bits to MEM108-MEM115

IF MEM100 THEN (OUT1)
IF MEM101 THEN (OUT2)
IF MEM102 THEN (OUT3)
…
…
…
IF MEM114 THEN (OUT15)
IF MEM115 THEN (OUT29)
```

And in the NCFILES directory we have the following programs:

SETBIT

```
if [#4201 || #4202] goto 1      ; skip if doing search or backplot
if #6001                        ; wait until program execution reaches this point
G10 P170 R[#9170 or (2 ^ #B)]   ; set the bit in P170
N1 ; end
```

CLRBIT

```
if [#4201 || #4202] goto 1                    ; skip if doing search or backplot
if #6001                                      ; wait until execution reaches this point
G10 P170 R[(#9170 and ~(2 ^ #B)) and 65535]   ; clear the bit in P170
N1 ; end
```

Then the individual outputs can be turned on in M&G codes like this

```
G65 "SETBIT" B1  ; turn on  OUT2
```

and turned off with

```
G65 "CLRBIT" B1  ; turn off OUT2
```

Here, B is a value from 0-15.

VOID Program Templates

There are times when it is nice to experiment with PLC programming without having to worry about fault messages appearing on the screen. The standard PLC program below, VOID.SRC, is a program that can be used to eliminate all faults.

```
; * * * * * * * * * * * * * * * * * * * * * * * * * * * *
; * File:    VOID.SRC
; * Purpose: blank program for testing
; *
; * Notes:   Forces all faults and stop bit off.
;            PLC_op_signal is forced to zero.
;
; * * * * * * * * * * * * * * * * * * * * * * * * * * * *

PLC_fault_out       IS OUT16   ;
Lubricant_low       IS OUT63   ;
Drive_fault_out     IS OUT64   ;
Spindle_fault_out   IS OUT65   ;
Stop                IS OUT75   ;
PLC_op_signal       IS OUT76   ;
PCPLC_running       IS MEM49   ;
Zero                IS MEM73   ;

;*****************************
;*       Program Start       *
;*****************************
;
```

```
;  Ensure the Zero MEM bit is forced to zero
;  In case it had previously been set and the
;  system was not powered off.
;
Zero                = Zero & / Zero
;
; Disable PLC, Lube, Spindle, and Axis Drive faults
;
PLC_fault_out       = Zero
Lubricant_low       = Zero
Drive_fault_out     = Zero
Spindle_fault_out   = Zero
;
; Prevent the stop bit from being set.  Normally,
; the Stop bit would be set by some other fault.
; If a different plc program was running, it may have
; set the fault.  We clear the Stop bit here so that this
; program can be compiled and installed without rebooting
; being needed to clear the fault.
;
Stop                = Zero
;
; Clear the PLC_op_signal so at startup CNC software does not "hangup"
; with a "Waiting for PLC operation" in the message window
; and then waiting for an <ESC> or <CANCEL>.
; If PLC_op_signal is 1, then trying to enter MDI
; mode in CNC software will prevent the Block? prompt
; from appearing and require the <ESC> or <CANCEL>
; to be pressed a couple of times to return to normal display.
;
PLC_op_signal       = Zero

;******************************
;  End Program
;******************************
```

Troubleshooting and Debugging

It is common during the development of programs to make errors. Errors can be viewed as either syntax errors or logic errors. Syntax errors are those errors that prevent a program from being compiled and are generally errors that are fixed easily. Logic errors are those errors that occur when a program behaves differently than expected.

Handling Syntax Errors

Syntax errors are best handled by fixing them in the order that they are output when the program is compiled. While the compiler will output the line number in the program where it encountered an error, sometimes the error is actually caused by something that preceded the displayed line number. The point here is do not assume that there is always something wrong with the line the compiler displays.

Handling Logic Errors

Logic errors are typically harder to find. It is possible that these types of errors are not discovered for months or even years.

There is a PLC bit watch display in the CNC software that can be toggled on and off using <ALT-I>. This will display the states of the bit numbers for INP1-INP80, OUT1-OUT80, MEM1-MEM80, and, if there is an XPLC program running, STG1-STG80. These bits can be watched to help determine the problem.

For bits that are outside the 1-80 range in the PLC debug display, it is usually best to set one of the lower MEM bits equal to that bit. For example, to monitor MEM200, write in the program:

```
IF MEM200 THEN (MEM35)
```

so that MEM200 can be monitored by viewing MEM35 in the watch display. The same techniques can also be used for monitoring timers, one-shots, or a complete boolean expression.

Common logic errors

There are several common mistakes made that cause logic errors. One of these mistakes is to reference a bit in an output coil action twice in a program. For example,

```
LUBE_FAULT   IS INP1
LEVEL_LOW    IS INP2

RED_LIGHT    IS OUT1

IF LUBE_FAULT THEN (RED_LIGHT)
…
…
…
IF LEVEL_LOW THEN (RED_LIGHT)
```

In the above example program, it is desired to turn on a RED_LIGHT whenever there is a LUBE_FAULT or if the LEVEL_LOW input was on. What happens when the program is executed is that the RED_LIGHT is on only when LEVEL_LOW is on. The RED_LIGHT never comes on when the LUBE_FAULT occurs. The reason is because the last statement that referenced RED_LIGHT will overwrite any previous changes.

The way to find this error is to search through the program for all occurrences of the particular bit. In the example above, the search would be for all occurrences of RED_LIGHT and OUT1. The reason to search for OUT1 is that it is possible it was referenced in the program without using the defined name, RED_LIGHT. Note there is no rule that plc bits must be referenced by their previously defined names.

There is a couple of ways to solve the problem with the most obvious being by replacing the two lines that reference RED_LIGHT with the following:

```
IF LUBE_FAULT | LEVEL_LOW THEN (RED_LIGHT)
```

Now, the program will work as expected.

Another common mistake is forgetting that a program is continuously being executed. Consider a short example where we desire that when the AUX1 key is pressed, then an internal counter will be incremented so as to keep track of the number of times the AUX1 key was pressed.

```
AUX1_KEY    IS INP49
COUNTER     IS W1

IF AUX1_KEY THEN COUNTER = COUNTER + 1
```

This program fragment would appear to work correctly but it does not. What happens is that COUNTER is incremented 256 times for every second the AUX1_KEY is held down. The solution to this problem is to use a one-shot to trigger the incrementing of the counter, as such:

```
AUX1_KEY    IS INP49
AUX1_1SHOT  IS PD1
COUNTER     IS W1

IF AUX1_KEY   THEN (AUX1_1SHOT)
IF AUX1_1SHOT THEN COUNTER = COUNTER + 1
```

Compilation errors

Many different errors can be displayed when trying to compile an XPLC program. The potential errors are listed below along with some brief source code examples that can cause them

General errors

"Memory error."
This error is generated if there is not enough memory available to compile the program. It is not likely that this error will be displayed, but it can be purposely caused by generating a program that has many label statements in it, to exceed the limit that would be reached by defining every PLC token once.

"Stack overflow!"
Like the memory error, it is unlikely that this error will ever be displayed. However, it can be caused by very excessive nesting within expressions.

"Error opening file *filename*"
This error is generated when XPLCCOMP cannot open a file named *filename*. It is most likely caused by specifying an input file that does not exist, but can also be caused by trying to open the .PLC file if it already exists and is a read-only file.

"Too many errors"
This message is displayed after 19 errors have been generated and displayed on the screen. This error causes compilation to be stopped.

"Malformed command line"
XPLCCOMP source_file[.ext] [output_file[.ext]]
This error is generated when there are too many or too few arguments supplied when calling the program.

Syntax errors

When errors related to the compilation process are encountered, they are displayed on the screen. The error logic used in the compiler reports only one error per line and does make a limited attempt to recover from errors, usually by discarding tokens until the next IF, THEN, or STG token is encountered. Errors are displayed in the following format:

`Error Line (line_number): message #token_string#`

`line_number` is the line number of the XPLC source program in which the error occurred

`token_string` is the sequence of characters that the compiler was looking at when the error occurred.

`message` is one of the messages described below.

"IF expected"
This error is generated when the compiler expects to see the IF token, but it does not. A program such as:

```
    STG1
    STG2
Error Line (2): IF expected #STG2#
```

"End of file expected"
Despite the name of this message, this error is usually generated when the compiler does not find the start of a valid rung (an IF or STG token) and then assumes that the next token is the end of the file.

"*symbol* already defined."
This error is generated by defining a symbol more than once, such as in the program:

```
    X_LIMIT  IS INP1
    X_LIMIT  IS INP2
Error Line (2): X_LIMIT already defined. #INP1#
```

"Invalid label statement"
This error is typically seen when a label statement does not define a valid PLC bit token, such as:

```
    X_LIMIT  IS  WHATEVER
Error Line (1): Invalid label statement #WHATEVER#
```

"Undefined label *identifier*"
This error happens when, during the compilation of the program, that an identifier has been found that has not been previously defined.

```
    IF LUBE_LOW THEN (OUT1)
Error Line (1): Undefined label LUBE_LOW #LUBE_LOW#
```

"STG expected"
There are two cases in which this error will be displayed, both of which are demonstrated in the program below:

```
    StageOne IS INP1
    StageOne
    IF INP1 THEN (OUT1)
    IF INP2 THEN JMP OUT2
Error Line (3): STG expected #IF#
Error Line (4): STG expected #OUT2#
```

Note that in the first case, the error is actually on line 2 but not recognized until line 3.

"THEN expected"
This error is generated when the THEN token is omitted or when there is an error trying to parse a valid <*boolean_expression*>. The program below demonstrates both cases.

```
    IF INP2 == INP2 THEN (OUT1)
    IF INP1 & INP2 JMP STG
Error Line (1): THEN expected #==#
Error Line (2): THEN expected #JMP#
```

"= expected"

```
    IF INP1 THEN W1
Error Line (1): = expected ##
```

"W expected"

```
    IF INP1 THEN BCD OUT1
Error Line (1): W expected #OUT1#
```

") expected"

```
    IF INP1 THEN (OUT1
Error Line (1): ) expected ##
    IF (INP1 THEN (OUT1)
Error Line (1): ) expected #THEN#
```

"Expected OUT token"

```
    IF INP1 THEN WTB W1 INP50
Error Line (1): Expected OUT or MEM token #INP50#
```

"Invalid action statement"

```
    IF INP1 THEN OUT1
Error Line (1): Invalid action statement #OUT1#
```

"Invalid numerical expression"

```
    IF INP1 THEN W1 = W1 +
Error Line (1): Invalid numerical expression ##
```

"Relational operator expected"

```
    IF W1 AND INP1 THEN (OUT1)
Error Line (1): Relational operator expected #AND#
```

"One of INPn OUTn MEMn STGn expected"

This error occurs if one of the expected tokens does not appear after the SET or RST command is parsed.

```
    IF INP1 THEN SET TMR1
Error Line (1): One of INPn OUTn MEMn STGn expected #TMR1#
```

"One of INPn OUTn MEMn STGn PDn Tn TMRn expected"

This error occurs after an initial `'('` is parsed as part of an action to denote an output coil instruction but none of the expected tokens are found.

```
    IF INP1 THEN (W1)
Error Line (1): One of INPn OUTn MEMn STGn PDn Tn TMRn expected #W1#
```

"Line too long"

This error occurs when a line exceeds 1024 characters in length.

"Integer const too large"

```
    IF INP1 THEN W1 = 2147483647
    IF INP2 THEN W2 = 2147483648
Error Line (2): Integer const too large #2147483648#
```

"Integer constant overflow"

This error indicates that not only is an integer constant too large but also that it overflows what can be stored in a 32-bit unsigned integer.

```
    IF INP1 THEN W1 = 4294967295
    IF INP2 THEN W2 = 4294967296
Error Line (1): Integer const too large #4294967295#
Error Line (2): Integer constant overflow #4294967296#
```

"Token out of range"

There are only 256 of each type of PLC token and they are numbered 1-256. Any value not inside this range causes this error.

```
    IF INP0   THEN (OUT1)
    IF INP256 THEN (OUT1)
    IF INP257 THEN (OUT1)
Error Line (1): Token out of range #INP0#
Error Line (3): Token out of range #INP257#
```

"Invalid identifier"

This error is generated when an otherwise valid identifier ends in a character that cannot be part of a valid identifier.

```
    X_LIMIT@ IS INP1
Error Line (1): Invalid identifier #X_LIMIT@#
```

"Invalid character"

Whenever a character is found that is not a part of the XPLCOMP language, this error message is generated.

```
    @HOME IS OUT2
Error Line (1): Invalid character #@HOME#
```

PLC Frequently Asked Questions

These are some common questions regarding the interaction and use of the two PLC programs that are being used in CNC10 software versions 7.50 and above.

Q. Does this FAQ's have anything to do with KOYO PLCs?

A. No. It has nothing to do with the PLC Direct by Koyo or any software having to do with it.

Q. Do I need to set the PLC type to dual in the configuration screen?

A. No. That setting is for third party PLCs only. Generally this should be set to normal.

Q. What are the two PLC programs and where are they located?

A. One is CNC10.PLC which is located in the CNC10 directory or CNC10t directory for lathes, the other is PC.PLC and is located in the PLC directory. The CNC10.PLC file is the same file that has been used on our controls for all previous versions of CNC10 software. The PC.PLC file is a new PLC file with expanded capabilities.

Q. What's the difference between the two?

A. CNC10.PLC – This PLC file runs on the CPU7 / CPU9 motion control card. It is limited in size to 765 tokens (approximately 2,800 bytes). This file can control inputs 1-80, outputs 1-80, and memory bits 1-80. It is the only PLC file that is needed for the control to function.

PC.PLC – This PLC file runs on the PC. Its file size can be much larger than the CNC10 PLC file. It has features such as stages, times, counters, and one-shots. It also has access to 255 inputs, outputs, and memory bits although there are only 80 physical inputs and outputs. With versions 8.10 and above, it can also have parameters passed to it such as number of tools in a tool changer. To use this PLC program memory bit 49 needs to be set to a 1. This tells the motion control card that the PC.PLC program will be controlling outputs 1-48, 81-255 and memory bits 1-255. Note that even though the PLC program has access to all outputs and inputs it can only write to outputs 1-48 and 81-255.

Q. How do I compile the PLC files?

A. The PLC programs use different formats and therefore need different compilers in order to work. The CNC10 PLC program uses PLCCOMP and the PC PLC program uses XPLCCOMP. Typical usage is as follows:

PLCCOMP [source file] [destination file] destination file = c:\CNC10\CNC10.plc

XPLCCOMP [source file] [destination file] destination file = c:\plc\pc.plc

Q. Why is there a need for two PLC programs?

A. The answer is the features in the new PLC program are useful in tool changer and custom applications, but the new PLC program doesn't have access to all of the I/O so the CNC10 PLC program needs to be there to control those I/O. A minimal PLC program for the CNC10 can be made by echoing memory bits to the outputs you need to control. Then you can control those memory bits in the PC PLC program thus making it possible to control virtually everything needed in the PC PLC program.

Q. Do I have to use both PLC programs?

A. No. You only need to use the CNC10 PLC program. Just make sure that you don't have a file called PC.PLC in your PLC directory.

Q. Do I need to have a PLCIO2 to use the new PLC program?

A. No. It can be used with any of our current PLCs: RTK2, PLC15/15, PLC3/3, Servo3IO, PLCIO2. It's just that the PLC programs will be limited to the physical I/O available on that PLC.

Q. What would happen if I set MEM49 and didn't use the PC.PLC program?

A. The motion control card would give up control of the outputs and memory locations and since there wouldn't be a PC PLC program running no outputs would ever turn on.

Q. Can I just use the PC side PLC program?

A. No. At this time there are still compatibility issues that make it necessary to have the CNC10 PLC program.

Q. How do I find out if my system is running both PLC programs?

A. To find this out all you need to do is look in the PLC directory. If there is a file called PC.PLC then your system is running both.

Q. How do I find out what source code was used to compile the PLC programs?

A. To do this you need to edit both the CNC10.PLC file in the CNC10 directory and the PC.PLC file in the PLC directory. At the top of each, there is a header that lists the name of the source code that each one was compiled from.

Q. Is there more information available about the PLC programs?

A. Yes. There is documentation on how to use both types of PLC programs within this chapter.