XPLCCOMP

USER

MANUAL

v1.00

REV 030214

Introduction

XPLCCOMP is a PLC compiler that accepts a text file as input and outputs a PLC program (a text file) that can be executed by the PC-side PLC executor program PCPLC.

Text file structure

The input text file that XPLCCOMP accepts is known as a plc source code file. The plc source code file is composed of lines of text. Lines of text can be up to 1024 characters in length. The source code lines of text contain sequences of tokens, keywords, and operators, which are described in more detail below. Semicolons `';'` and all characters after them on a line are treated as comments. The compiler is case-insensitive and allows free form input, i.e., blank lines may be inserted freely and spaces are only required between tokens, keywords, and operators when necessary for correct interpretation.

Grammar

The grammar helps to explain what constitutes a valid plc program but it does not offer much to explain the semantics of the language, or what that valid program will accomplish. The meaning of the various constructs will be explained in more depth after the explanation of the grammar. The following typeface conventions are used:

non-terminals	Italicized lowercase typeface for non-terminal symbols
TERMINAL	Bold uppercase typeface for terminal symbols
`source_code`	Courier typeface for program fragments
`'ccc'`	Courier typeface enclosed in quotes for literal character constants or a series of literal character constants

In describing the XPLCCOMP language, the grammar will be augmented with written descriptions and examples, and will mostly flow in a manner from high level constructs down to the lower level constructs and terminal symbols.

plc_program: *label_statements rungs*

There are two main sections in a plc source code program. The first part of a program contains definition statements, which are used to label the various terminal tokens, which makes for better readability and maintainability of plc programs. The second part contains the program code, or rungs, in the traditional plc vernacular.

label_statements: *label_line label_statements*
EPSILON

Label statements are composed of a label line followed by more label statements. The terminal symbol EPSILON is used to signify an empty set, or nothing at all. Thus, there doesn't need to be any label statements in a source code program in order for it to be compiled successfully, but there can be one or more if desired.

label_line: *identifier* **IS** *plc_token*

The definition of *identifier* is that it is a sequence of one or more characters, up to a length of 32 characters, that begins with a letter (A-Z) and may be followed by one or more letters, underscores `'_'`, or digits (0-9). Identifiers that exceed 32 characters will be truncated.

plc_token: **INP**nnn
OUTnnn
MEMnnn
Wnnn
STGnnn
PDnnn
Tnnn
TMRnnn

Here, nnn denotes a number from 1-256. Therefore, INP1, INP2, INP3, INP4, INP5, all the way up to INP256 is a valid *plc_token*. The plc tokens represent the following:

INP represents a physical input in the electronics hardware. Note that the number of physical inputs is hardware specific. However, XPLCCOMP will compile a program successfully regardless of how many physical inputs may be available on the hardware platform. Specific hardware platforms are discussed in more depth later in the manual.

OUT represents a physical output in the electronics hardware. Again, the number of physical inputs is hardware specific, and XPLCCOMP will compile a program successfully regardless of how many physical inputs are actually available.

MEM represents a one bit memory location.

W represents a word memory location. A word is a 32-bit signed integer, which has values in the range -2147483648 to +2147483647.

STG represents a type of subroutine or function marker that is used to provide structure to plc programs. It is used to section a plc program into different stages. It also represents a one bit memory location that determines whether the particular rungs in a stage are active or not. Whether a **STG** token represents a marker or a one bit status condition is determined by the context in which it is used. Stages are a shorthand way of coding structure into a plc program. The same functionality that the **STG** token provides can also be accomplished using other techniques. Stages are discussed in more depth later in the manual.

PD represents a positive differential coil, or one-shot. A one-shot is a special type of plc memory bit that is active for only one pass of the plc program after a positive edge (a zero to one transition of the input logic).

T and **TMR** represent a reference to a timer. A timer has two values and an input associated with it. The values are the preset value and the current value. The input to the timer is what causes the timer to time. When the input to the timer is true, the timer updates its current value. In a plc program, a **T** token represents the status of a timer, i.e., a one bit internal memory location signifying whether the timer has expired, which is when the current value is greater than or equal to the preset value, provided the current value is not zero. A **TMR** token represents the current value of a timer, which is the number of time units that the timer has been on. Time units are integer values in 0.01 second increments. Thus, a value of 100 represents one second, a value of 1234 represents 12.34 seconds. The maximum time that a timer can time to is close to twenty days. A **T** or **TMR** token can also represent an assignment to a timer preset value or a connection to the timer input, depending upon the context in which it is used. Note that although the resolution of the timer is in 0.01 second increments, the hardware inputs and outputs may not react this fast.

Some examples of valid label statements are:

```
Y_LIMIT           IS INP23
LIMIT_TRIPPED     IS MEM134
MAINSTAGE         IS STG15
```

```
TOOLNUMBER          IS W178
BRAKE_MODE_1SHOT    IS PD234
```

It is not an error for the same *plc_token* to be defined more than once. For example, the program containing the following:

```
X_LIMIT             IS INP1
Y_LIMIT             IS INP1
```

will compile correctly given these label statements. When either X_LIMIT or Y_LIMIT is referenced in the rungs, INP1 will be referenced in the compiled program. While defining the same *plc_token* more than once is not an error, it is considered a bad programming practice.

It is an error for the same *identifier* to be used more than once. For example,

```
X_LIMIT             IS INP1
X_LIMIT             IS INP2
```

will generate an error during compilation.

rungs: *rung rungs*
EPSILON

Rungs is nothing, a single rung, or more than one rung. Note that since *label_statements* can also be empty, or nothing, that a text file with nothing in it, or a text file with comments only, is a valid *plc_program*. In this case, the program that is generated will do nothing and will contain only the end-of-program instruction.

rung: **STG IF** *boolean_expr* **THEN** *actions*
IF *boolean_expr* **THEN** *actions*

A rung may have an optional **STG** token followed by the keyword **IF**, a *boolean_expr*, the keyword **THEN**, and finally a set of *actions*. If the rung starts with the optional **STG** token, then it signifies the start of a stage, or section, of the plc program. All *rungs* from this point on until the next **STG** token or end of the file are considered part of the designated stage.

boolean_expr: *single_bit_ref*
boolean_expr **OR** *boolean_expr*
boolean_expr **AND** *boolean_expr*
boolean_expr **XOR** *boolean_expr*
'(' *boolean_expr* **')'**
NOT *boolean_expr*
relational_expr

A *boolean_expr* is an expression that resolves to a boolean value, i.e., either true (1) or false (0). A *single_bit_ref* is one of **INP**nnn, **OUT**nnn, **MEM**nnn, **STG**nnn, **T**nnn, or **PD**nnn. When the **STG** token is used in this context, as part of a *boolean_expr*, it represents a single bit internal memory location associated with the active status of the **STG**. When the **PD** token is used in this context, it represents the status of the internal single bit token associated with the **PD** token, which status is whether or not the one-shot is on, or triggered (1) or off, or not triggered (0).

The **OR** token can also be replaced with the `'|'` character. The **AND** token can be replaced by the `'&'` character, the **XOR** token can be replaced by the `'^'` character, and **NOT** can be replaced with the `'!'` character. The left parenthesis `'('` character and right parenthesis `')'` character, respectively, and can be used to force operator precedence and associatively. Some examples of valid boolean expressions:

```
NOT INP1
INP1  AND  INP2
INP1 OR INP2 OR INP3
STG1 AND STG2 AND PD13
MEM123 XOR MEM124
(INP1 OR INP2) AND INP3
STG2 & STG3 | ( OUT1 ^ (MEM76 AND ! INP17))
T1 & OUT2
```

relational_expr: *numerical_expr relational_op numerical_expr*

relational_op: '**<**'
'**<=**'
'**>**'
'**>=**'
'==**'**
'**!=**'

numerical_expr: *integer_ref*
numerical_expr '-' *numerical_expr*
numerical_expr '+' *numerical_expr*
numerical_expr '*' *numerical_expr*
numerical_expr '/' *numerical_expr*
'(' *numerical_expr* ')'
'-' *numerical_expr*

integer_ref: **W**nnn
TMRnnn
FLT
integer_constant

An *integer_const* is defined as a sequence of one or more digits (0-9). The compiler will generate warnings for constants that are too large or that overflow. See the section on errors for more information. **FLT** is a special word memory location that, when it has a non-zero value, holds information about internal executor errors. The **FLT** internal memory location 'bit maps' the individual faults so that if more than one fault occurs, it can be determined if desired. In practice, any non-zero value is a cause for concern.

The **FLT** memory word is mapped as follows:

1 = Stack fault
2 = Division by Zero fault
8 = Illegal instruction

Some further examples of boolean expressions and relational expressions are:

```
W1 <= W2
FLT != 0
TMR1  > 34  &  TMR1 <  56
```

```
W1 < 3*4+5 & W1 < 10*5/2 OR INP7
```

While it is not obvious from the grammar definitions, the XPLCCOMP compiler assumes (when parsing tokens between an **IF** and **THEN** token) that an expression that starts with a `'('` is a boolean expression. Therefore, a numerical expression cannot start with a `'('`. There are several ways that the same numerical result can be obtained for a numerical expression without being able to start it with a parenthesis, as in:

```
0+(W1+W2)*W3
1*(W1+W2)*W3
```

which both evaluate to the same result as if

```
(W1+W2)*W3
```

were able to be compiled without error. Note that in this case, the expression

```
W3*(W1+W2)
```

also produces the same numerical result and is not a compilation error.

actions: *action more_actions*

more_actions: `','` *actions*
EPSILON

Actions consist of a single action or multiple actions separated by commas. Note that at least one action must be present in order for the compiler to recognize a valid rung of plc program logic.

action: **W**nnn `'='` *numerical_expr*
timer_ref `'='` *numerical_expression*
SET *bit_token1*
RST *bit_token1*
LDT Wnnn
LSR Wnnn (CNC7 v8.10+)
LCP Wnnn (CNC7 v8.10+)
LTS Wnnn (CNC7 v8.10+)
LMT Wnnn (CNC7 v8.10+)
LP0 - LP9 Wnnn (CNC7 v8.11+)
BIN Wnnn
BCD Wnnn
WTB Wnnn **OUT**nnn
WTB Wnnn **MEM**nnn
JMP STGnnn
`'('` *bit_token2* `')'`

bit_token1: **INP**nnn
OUTnnn
MEMnnn
STGnnn

bit_token2: *bit_token1*
PDnnn
timer_ref

timer_ref: **T**nnn

General program execution

When PCPLC executes the compiled source code program, it does so in the same sequence as the program has been entered into the source file, proceeding from top to bottom.

PCPLC evaluates the boolean expression between the **IF** and **THEN** tokens, which will result in either a 1 (true) or 0 (false). The result of this boolean expression is remembered and is used to determine what effects the actions will have. As a simple example, consider the following program:

```
IF MEM1 THEN SET OUT1
```

In this program, the boolean expression is a simple one bit location MEM1. The executor will get the status of this memory bit. In considering the action SET OUT1, the executor will do one of the following:

(1) If MEM1 is true (1), then OUT1 will be set (turned on).

(2) If MEM1 is false (0), then OUT1 will not be set (turned on).

Note that in this example of the SET action, that if MEM1 were false, OUT1 is not reset (turned off), but merely that there will be no explicit action taken to turn on the output. Further, this explanation does not apply to every type of action.

Evaluation of boolean expressions

Evaluation of boolean expressions, just like mathematical or numerical expressions, is done according to certain rules. For single bit token expressions, as above, the executor simply queries the state of the corresponding bit. For **INP**, **OUT**, and **MEM** bits, this meaning is straight forward. For **T** references, the result returned is true (1) if the associated timer current value >= preset value, provided that the current value is not zero, otherwise the result is false (0). For **STG** types, the result is true (1) if the internal one bit active status is set (1), otherwise the result is false (0). The internal active status of a **STG** token is controlled through certain action statements discussed later. For **PD** types, the result returned is true (1) if the internal status associated with the **PD** coil is on, otherwise the result is (0). Control of the internal status of the **PD** coil is by way of certain action constructs described in more detail later.

For boolean expressions containing more than a single bit token, i.e., combinations of single bit tokens using the **AND**, **OR**, **XOR**, and **NOT** operators, or for those containing relational or numerical expressions, the results are determined by a set of rules known as the associatively and precedence rules.

Operator precedence and associatively

When forming expressions composed of multiple tokens and operators, the following table lists the operators with highest precedence starting at the top. Operators of the same operator category have equal precedence. Parenthesis, `'('` and `')'`, can be used to force operator precedence or associatively.

Operator Category	**Token**	**Description**	**Associatively**
Unary	NOT (!)	Unary NOT	Right
	-	Unary Arithmetic Negation	Right
Multiplicative	*	Binary Multiply	Left

	/	Binary Divide	Left
Additive	+	Binary Addition	Left
	-	Binary Subtraction	Left
Relational	<	Less than	None
	<=	Less than or equal to	None
	>	Greater than	None
	>=	Greater than or equal to	None
	==	Equal to	None
	!=	Not equal to	None
XOR	XOR (^)	eXclusive OR	Left
AND	AND (&)	AND	Left
OR	OR (|)	OR	Left
Assignment	=	Assignment	None

The results of the **AND**, **OR**, **XOR**, and **NOT** operators are shown in the table below.

bit1	bit2	**AND**	**OR**	**XOR**	**NOT**
		`bit1 & bit2`	`bit1 | bit2`	`bit1 ^ bit2`	`! bit1`
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

An associatively of none implies that an operator cannot be used two times in a row, for example, the partial expression `W1 < W2 < W3` is invalid because the relational operators have no associatively. Since the assignment operator '=' has no associatively, an expression such as `W1 = W2 = W3` is also invalid.

The semantics of action statements

Wnnn `'='` *numerical_expr*

This action is used to assign a numerical value to a **W** memory reference. Its meaning should be straight forward. This action will only be completed if the *boolean_expr* for the rung evaluated to true (1).

timer_ref `'='` *numerical_expression*

This action statement is used to assign the preset value of a timer. Here, a timer reference can be either a **TMR** or **T** token. Remember that the preset value of a timer is specified in 0.01 second

increments. Timer presets default to 0 at plc executor initialization. The assignment will only take place if the *boolean_expr* for the rung evaluated to true (1).

Example:

```
T1 = 10 * 100      ; timer 1 preset set to 10 seconds
T2 = 10 * 100 * 60 ; timer 2 preset to 10 minutes
```

SET *bit_token1*
RST *bit_token1*

The SET/RST commands are used to turn on/off **INP**, **OUT**, **MEM**, and **STG** bits. For OUT and MEM bits, the meaning should be straight forward.

Setting or resetting an INP bit is allowed and will compile, but will not be able to have any effect on the actual inputs. There are specialized inputs (INP33-INP48) in the PLCIO2-CPU7 hardware combination which are treated as M function outputs but these inputs cannot be directly controlled by an XPLCCOMP plc program. More information is provided in the hardware platform section of the manual which discusses the plc hardware and software specifics of different platforms.

Setting or resetting a STG will set or reset an internal status bit associated with each STG. The internal status bit associated with each STG is an indicator of whether a STG is active or not. If a STG is active, then all rungs within that STG are executed. If a STG is inactive, then the rungs within that stage are not executed. What happens when an inactive stage is encountered is that any single output coil instructions in that stage are handled in a special manner. The specialized handling for a single output coil instruction encountered in an inactive stage is discussed in the section dealing with the single output coil instructions.

LDT Wnnn

The LDT command is used to load the tool number into the indicated word memory location. The tool number to be loaded will be that sent using the last M107 command. M107 is part of the default actions for an M6 command with auto tool changer installed (parameter 6 = 1) and it is most likely part of any customized M6 or other tool changing function. The tool number loaded is in straight binary format, as opposed to a BCD format. See the BCD and BIN command below.

LSR Wnnn (available in CNC7 v8.11+)

The LSR command is used to load the CPU7 stop reason into a word memory location. The primary use of this command is to detect when the CPU7 control software has been exited so that certain PLC functions (such as tool indexing) can be locked out. When CPU7 software has been exited, the stop reason = 17. This command is available in CNC7 software v8.10+.

LCP Wnnn (available in CNC7 v8.11+)

The LCP command is used to load the ATC carousel position into a word memory location. The primary use of this function is to recall the ATC carousel position upon power-up initialization. The machine parameters in the CNC7 software must also be set properly. Parameter 160 (Enhanced ATC) must be set to a 1 (nonrandom type) or 2 (random type) in order for the carousel position to be valid. If P160 = 0, the value loaded will be 0. This command is available in CNC7 software v8.10+.

LTS Wnnn (available in CNC7 v8.11+)

The LTS command is used to load the tool in the spindle into a word memory location. The primary use for this command is for nonrandom type ATC systems to recall the "putback" location for the tool in the spindle upon power-up initialization. The machine parameters in the CNC7 software must also be set properly. Parameter 160 (Enhanced ATC) must be set to a 1 (nonrandom type) so the value loaded

will be the "putback" value for the tool that is in the spindle. If P160 = 2, the value loaded will be the tool number of the tool in the spindle, but this information has no practical application as random type ATC systems place the tool in the spindle back into the location of where the next tool is picked up from.. If P160 = 0, the value loaded will be 0. This command is available in CNC7 software v8.10+.

LMT Wnnn (available in CNC7 v8.10+)

The LMT command is used to load the maximum number of bin locations in a tool carousel into a word memory location. The primary use for this command is for ATC systems to load the maximum number of tools upon power-up initialization so that the PLC program does not need to be modified or hard-coded for ATC systems with varying numbers of bin locations. The machine parameters in the CNC7 software must also be set properly. Parameter 161 (ATC Maximum Tool Bins) should be set to the number of bins in the tool carousel. This command is available in CNC7 software v8.10+.

LP0 Wnnn **- LP9 W**nnn (available in CNC7 v8.11+)

The LP0 - LP9 commands are used to load CNC7 machine parameters 170-179. CNC7 machine parameters 170-179 are sent to the PLC executor (PCPLC) at the start of CNC7 and whenever the machine parameters are saved. The values are 16-bit unsigned integer values (0-65535).

BIN Wnnn

The BIN command is used to change the indicated word memory location to a binary value, assuming that the value presently stored is in BCD format. The result is undefined for invalid BCD values. Valid BCD values are from 0-99999999.

BCD Wnnn

The BCD command is used to change the indicated word memory location to a BCD value, assuming that the value presently stored is in a binary format. The result is undefined for values outside the range 0-99999999.

WTB Wnnn **OUT**nnn
WTB Wnnn MEMnnn

The WTB command writes the lower byte (eight bits) of the indicated **W** memory location to a sequence of outputs/memory starting with the indicated **OUT/MEM** bit reference. For example,

```
WTB W1 OUT41
WTB W1 MEM200
```

will write the lower byte of W1 into OUT41-OUT48. OUT41 will hold the least significant bit of the lower byte of W1 and OUT48 will hold the most significant bit of the lower byte of W1.

JMP STGnnn

The **JMP** command will reset the current executing stage and set the stage of the specified **STG** reference. Given the following program,

```
STG1

IF INP1 THEN JMP STG2
```

the action of JMP STG2 is the same as if the following program were executed:

```
STG1

IF INP1 THEN RST STG1, SET STG2
```

Single Output Coil Instructions

'(' *bit_token2* ')'

The single output coil instructions vary as to their meaning depending upon the type of *bit_token2* supplied as an argument and whether the program rung that includes them is part of a stage.

For **INP**, **OUT**, and **MEM** types that are executed as part of an active stage or as a rung not in a stage, the action is to set the bit if the *boolean_expr* evaluated to true and to reset the bit if *boolean_expr* evaluated to false. If the single output coil instruction is within a stage that is inactive, the **INP**, **OUT**, or **MEM** bit will be reset regardless of whether *boolean_expr* evaluated to true or false.

```
STG1
IF MEM1 THEN (OUT1)
```

The above rung of program will turn on OUT1 if MEM1 is on (1) and STG1 is active. Execution will turn off OUT1 if MEM1 is off (0) or STG1 is inactive. Note that when single output coil commands are used with **INP**, **OUT**, or **MEM** types, the following program is the same, considering that STG1 is active.

```
STG1
IF MEM1   THEN SET OUT1
IF ! MEM1 THEN RST OUT1
```

For **STG** types, the action is the same as for INP, OUT, or MEM bits, except that it is the internal stage status bit that is set (activated) or reset (deactivated).

For **PD** types, the action depends upon whether the current stage, if any, is active, and the result of *boolean_expr*. A **PD** type has associated with it two internal status bits. One of these status bits holds the result of the last *boolean_expr* result. The second internal status bit determines whether the **PD** coil is on (1) or off (0). A **PD** coil is on if the last *boolean_expr* evaluation was false (0) and now it is true (1), otherwise the **PD** coil is off. A **PD** coil is also turned off if it was on when the rung containing it is evaluated, since **PD** coils are known as one-shots, where the one means one pass of plc program execution. If a **PD** coil is referenced in a stage that is not active, the last status is set to true and the coil is turned off. The reason that the last status is set to true is because when the stage is active again, it won't set the **PD** coil if the *boolean_expr* is initially true also. Stated another way, if the last status were not set to true, it would be possible that the **PD** coil acts as a level triggered device instead of a leading edge triggered device, or positive differential device, at least for the initial coil on status.

```
TOOL_COUNTER    IS INP1
TOOL_NUMBER     IS W1
```

```
IF TOOL_COUNTER THEN (PD1)
IF PD1 THEN TOOL_NUMBER = TOOL_NUMBER+1
```

This program will only update the tool number on the rising edge of the tool counter input.

For **T** or **TMR** types, the single output coil instruction specifies a connection to the timer input. When the *boolean_expr* for the associated rung evaluates to true, then the timer current value will be updated. If the *boolean_expr* for the associated rung evaluates to false, the timer current value will be set to zero, thereby disabling the timer. If the timer reference in a single output coil instruction is referenced in a stage that is not active, the result is that the timer current value is set to zero.

Example program using a timer:

```
STG1
IF MEM1 OR !MEM1 THEN T1 = 100,
                      JMP STG2
STG2
IF INP1 THEN (T1)
IF T1 THEN (OUT1)
```

In the above program, the first stage will set the timer 1 preset to 100 (one second) and will activate STG2. In STG2, if INP1 is 1, then timer 1 will start timing. If INP1 is 1 for at least one second, then OUT1 will be on. In fact, OUT1 will be on as long as INP1 has been on for one second or more. If INP1 goes off, then the timer is reset and then INP1 must be on continually for one second for OUT1 to come on again.

Understanding Stages

As mentioned before, stages are a way to section a plc program. At execution startup, STG1 is active and all other stages are inactive. Consider the scenario where what is wanted is to program a process to control a drilling operation which involves certain steps. On the rising edge of a start switch, start the following sequence of events:

(1) Turn on a solenoid to clamp the part. Wait for acknowledgment of a clamp signal.
(2) Turn on an output to start a drill motor and wait for an "at speed" signal.
(3) Engage a solenoid that will move the drill into the part and wait for the signal that the drill has reached the "at depth" position.
(4) Turn off the clamp, the drill motor, and drill in solenoid and wait for the unclamp sensor, the zero speed signal, and that there is no "at depth" signal, before starting over.

```
START_SWITCH      IS INP1
CLAMP_SENSOR      IS INP2
DRILL_AT_SPEED    IS INP3
AT_DEPTH          IS INP4
UNCLAMP_SENSOR    IS INP5
ZERO_SPEED        IS INP6
```

```
CLAMP_SOLENOID    IS OUT1
DRILL_ON_SIGNAL   IS OUT2
DRILL_IN_SOLENOID IS OUT3
START             IS STG1
WAIT_FOR_CLAMP    IS STG2
START_DRILL       IS STG3
MOVE_DRILL_IN     IS STG4
FINISH_CYCLE      IS STG5
START
  IF START_SWITCH THEN (PD1)
  IF (PD1) THEN SET CLAMP_SOLENOID,
                JMP WAIT_FOR_CLAMP
WAIT_FOR_CLAMP
  IF CLAMP_SENSOR THEN SET DRILL_ON_SIGNAL,
                       JMP START_DRILL
START_DRILL
  IF DRILL_AT_SPEED THEN SET DRILL_IN_SOLENOID,
                         JMP MOVE_DRILL_IN
MOVE_DRILL_IN
  IF AT_DEPTH THEN RST DRILL_ON_SIGNAL,
                   RST DRILL_IN_SOLENOID,
                   RST CLAMP_SOLENOID,
                   JMP FINISH_CYCLE
FINISH_CYCLE
  IF UNCLAMP_SENSOR & ! AT_DEPTH & ZERO_SPEED THEN JMP START
```

Using stages, the program is written in a manner that closely follows the steps listed in the procedure given for the task. Suppose the program were written without stages. It would probably look like the following program. Note that the input and output definitions remain the same and are not listed again.

```
START             IS MEM100
WAIT_FOR_CLAMP    IS MEM101
START_DRILL       IS MEM102
MOVE_DRILL_IN     IS MEM103
FINISH_CYCLE      IS MEM104
INIT              IS MEM105

IF ! INIT THEN SET INIT, SET START
IF START_SWITCH THEN (PD1)
IF PD1 & START_SWITCH & START THEN SET WAIT_FOR_CLAMP,
                                   SET CLAMP_SOLENOID,
                                   RST START
IF WAIT_FOR_CLAMP & CLAMP_SENSOR THEN SET START_DRILL,
                                      RST WAIT_FOR_CLAMP,
                                      SET DRILL_ON_SIGNAL
IF START_DRILL & DRILL_AT_SPEED THEN SET MOVE_DRILL_IN,
                                     RST START_DRILL,
                                     SET DRILL_IN_SOLENOID
IF MOVE_DRILL_IN & AT_DEPTH THEN SET FINISH_CYCLE,
                                 RST MOVE_DRILL_IN,
                                 RST DRILL_ON_SIGNAL,
                                 RST DRILL_IN_SOLENOID,
                                 RST CLAMP_SOLENOID
IF FINISH_CYCLE & UNCLAMP_SENSOR & ! AT_DEPTH & ZERO_SPEED
```

```
THEN SET START,
     RST FINISH_CYCLE
```

Hopefully, the concept of stages is more clear. What is not seen in these simple examples is how stages are helpful in writing and debugging real world applications. Also, a program written using stages executes slightly faster, which may be an advantage for larger programs. Execution is faster not only because the compiled programs using stages are shorter, but because the executor handles inactive stages by quickly skipping through tokens. The first program using stages compiles to 46 words of memory whereas the second program written without stages compiles to 70 words of memory.

Compiling programs

Programs are compiled by the following syntax:

XPLCCOMP *inputfile*[.*ext*] [*outputfile*[.plc]]

where *inputfile* is the name of the file that is being compiled. If the input file is not supplied with an extension, the default extension of .src is used to find the file.

outputfile is the name of the file that the compiled plc program will be written to. If no extension is supplied, a default extension of .plc will be used. If no output file is specified, the output file is assumed to be the name of the input file with a .plc extension.

If there are errors that prevent successful compilation, they will be displayed. If compilation was successful, the following is written to the screen:

```
XPLCCOMP v. x.xx  - Centroid XPLC compiler
Copyright 2001 Centroid Corp.

Compilation successful
Program size: nnnn
```

General errors

There are a couple of errors which are not errors related to compiling a plc program file. When these errors are generated, the error is written to the screen after the two line program information is displayed. The general errors that can be displayed are:

"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 seen, 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 excessive nesting within expressions.

"Error opening file *filename*"

This error is generated when the system cannot open a file named *filename*. It is most likely caused by specifying an input file which does not exist, but can also be caused by trying to open the output 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.

Compilation 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, using 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 plc program source code in which the error occurred, *token_string* is the sequence of characters that the compiler was looking at, and *message* is one messages described below. Most of the errors are self-explanatory.
"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 rungs, 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 cases, 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 MEM50
Error Line (1): Expected OUT token #MEM50#
```

"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 a single 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#
```

Understanding plc program execution

Programs that are compiled by XPLCCOMP are executed by another program, PCPLC, which is a DOS command line terminate-and-stay-ready (TSR) program. When PCPLC is started for the first time, the screen will appear as below:

```
PCPLC v. 1.00 Beta - Centroid XPLC Executor
Copyright 2001-2002 Centroid Corp.
Plc Memory Real Mode Segment:Offset Address 0bef:9fde Size 6420
Running Frequency: 256 Hz
```

This is assuming that the plc program was found and loaded successfully. The plc program that is loaded when PCPLC is first started is a file called PC.PLC and that resides in the current directory.

If the PC.PLC file cannot be found, the output of PCPLC will display

```
Error opening file PC.PLC
```

After PCPLC has been loaded successfully it resides in memory and the only way to stop it, or to start it running with a different plc program, is to reboot. If an attempt is made to start PCPLC again, the following additional information is displayed.

```
PCPLC already loaded.
Status: Deactivated.
```

After CNC7 software (v7.50+) starts and initializes the hardware, a command is sent to activate PCPLC. Once activated, PCPLC will start executing the plc program at the frequency displayed upon startup. Executing PCPLC again will result in the following display:

```
PCPLC already loaded.
Status: Activated.
```

There are two more errors that can occur when trying to load PCPLC. The first is a file checksum error, which means that PCPLC has determined that the file has been corrupted. The second is that the compiled plc program is too long In this case, the display will include one of these messages:

```
File checksum error.
File too large.
```

Initialization

When PCPLC is loaded, the following initialization occurs:

(1) All STGs are set to inactive, except for STG1, which is active from the start.
(2) All MEM bits are set to 0.
(3) All timers are set to a preset value of 0 and a current value of 0.
(4) All OUT types are off (0).
(5) All PD type have the last status set to 1 and are off (0).
(6) All W types are initialized to 0.

After the CNC7 software (v7.50+) starts it activates PCPLC. From this point on, execution of the plc program will occur at the frequency shown at startup, which is typically 256 times a second. The steps that occur on each pass of program execution are:

(1) Read the status of all INPuts.
(2) Execute the PLC program.

(3) Update the OUTputs.

During execution of the plc program, there are different rules used when plc types are referenced. OUT types (and INP types that can be modified) are not updated until after execution of the current pass is complete, whereas STG, MEM, and other types are updated during execution. An example serves to understand this. Consider this program:

```
IF INP1 THEN (OUT1)
IF OUT1 THEN (OUT2)
IF OUT2 THEN (OUT3)
IF OUT3 THEN (OUT4)
```

When this program is executed and INP1 is (1), OUT1 comes on after the end of program execution, OUT2 comes on after the next pass, OUT3 after the next pass after that, and so on. Stated another way, references to OUT types always return the value that existed at the start of that pass of program execution. Now, consider what happens when MEM types are used:

```
IF INP1 THEN (MEM1)
IF MEM1 THEN (MEM2)
IF MEM2 THEN (MEM3)
IF MEM3 THEN (MEM4)
```

When this program is executed, as soon as INP1 is (1), then at the end of that pass, MEM1-MEM4 are all on. W types are updated in a similar manner as MEM types. For example,

```
IF INP1 THEN W1 = W1 + 1,
             W2 = W1 + 1
```

At the start of program execution, W1 = 0. When INP1 is (1), then after that pass, W1 = 1 and W2 = 2. STG types are also treated like MEM or W types.

```
STG1
IF INP1 THEN SET STG2
STG2
IF INP2 THEN SET STG3
STG3
IF INP3 THEN JMP STG1
```

In the above program, when INP1 is (1), STG2 is treated as active on that very same pass.

For timer types in which the current value will be updated, the actual time passed is considered a constant through that entire pass of plc program execution. Because timer resolution is in 0.01 second increments and considering that a plc program executes at 256 times a second, then every pass will update timer current values by 1/256 of a second, subject to rounding of integer types. The maximum error is always <10ms and for times that are multiples of 250ms, (such as 1/4 sec, 1/2 sec, 3/4 sec, 1 sec, etc.) the time error is virtually zero.

Understanding the PLCIO2-CPU7 platform

The specific platform for which the executor, PCPLC, executes compiled plc programs, effects several areas that are important to understand when using XPLCCOMP to write plc programs. The most widely used platform will be discussed first, which is referred to as the **PLCIO2-CPU7** platform. In this platform, the PLCIO2 I/O hardware is used and is connected to a CPU7 (or CPU9) motion control card which has a special PIC chip installed suitable for communicating with the PLCIO2 I/O hardware. Control software versions v7.50+ are supported.

The PLCIO2 I/O board limits the number of physical inputs and outputs available to be programmed. There are 35 physical inputs, which are mapped to INP1-INP15, INP17-INP32, and INP59-INP62. There are 39 physical outputs, mapped to OUT1-OUT15, OUT29-OUT48, and OUT59-OUT62. INP26, OUT40, and OUT31 have special meaning and are not considered general purpose I/O points.

The CPU7/CPU9 motion control card also runs a plc program. The plc program that is run by the motion control card is compiled by the PLCCOMP compiler according to its language and form of input. Under this platform, certain rules apply as to what plc program can control what I/O, and so forth.

There are two methods of running plc programs that the CPU7 will use. The determination of which method is used is indicated by the value of **MEM49**. It is helpful to know that the CPU7 plc platform recognizes 80 INP, 80 OUT, and 80 MEM types. No other types are supported and it is impossible for a CPU7 plc program to read or reference any INP, OUT, or MEM types that are beyond 80.

If MEM49 = 0, then CPU7 controls (writes) everything except MEM1-MEM72 and it does not matter what program PCPLC is executing. In v8.10+, the CPU7 controls everything except MEM49-MEM56.

IF MEM49 = 1, then CPU7 controls the writing of:

INP1-INP80
OUT17-OUT28, OUT49-OUT80
MEM73-MEM80 (MEM57-MEM80 in v8.10+)

and PCPLC will control the writing of:

OUT1-OUT16,
OUT29-OUT48
MEM1-MEM72

INP81-INP256 and OUT81-OUT256 are programmable but have no effect other than to be like a MEM bit, except they are updated like any other output (or input) at the end of a pass. MEM81-MEM256 act just like one of the MEM1-MEM72 bits.