

**A Programming Language for
A Process Control Computer**

**Wayne D. Woodruff
Villanova University
Electrical Engineering Department
E.E 9030 Independent Study, Fall 1986
Dr. Richard Perry, Advisor**

Table of Contents

Abstract	3
PSC market.....	3
PSC Environment	3
An overview of the PSC.....	3
PSC Executive program	4
PSC Logic Program	4
PSC Batch Sequencer Program	5
PSC I/O Program	7
PSC Communications Program	7
PSC Language, Detailed examples	7
PSC Language, Backus-Naur specification	13
Configuration Description.....	13
Input/Output Description.....	14
Uses Description	15
Recipe Description.....	17
Device Description.....	17
Logic Device Description.....	19
Sequence Device Description	27
Development of the Compiler	32
Conclusions	34
References	36
Appendix A Backus-Naur Specification.....	37

Table of Figures

Figure 1 Programmable Sequence Hierarchy.....	4
Figure 2 Example of a Pump Starter Network	5
Figure 3 Example of the Batch Sequencer Language.....	7
Figure 4 Generic Device	12

A Formal Computer Language for a Process Control Computer

Abstract

Typically, Process Control Computers do not use a formal programming language to describe the customer's control program. More often, they use a fill-in-the-blanks approach, which does not translate into a formal grammar. In the Moore Products Co. Programmable Sequence Controller (PSC), a formal computer programming language was developed to allow users to create more flexible and more powerful control programs.

PSC market

The PSC was designed to fill a void in the small batch control market. Traditionally, these processes are controlled using conventional Programmable Logic Controllers (PLC's). Programming PLC's to control batches using ladder logic has proven cumbersome and ineffective. To overcome this obstacle, MYSL (MYcro Sequence Language) was developed. MYSL is a dedicated language optimized for batch process control. MYSL can communicate directly with the PSC ladder logic program to create a tightly connected control program.

PSC Environment

The PSC was designed to run on the Moore Products Co. Local Instrument Link (LIL). The LIL is a token passing, logical ring communications link, similar to IEEE 802. The LIL accommodates other Moore Products Co. instruments as well as foreign devices (e.g. Allen Bradley Programmable Controllers, IBM Personal Computers, etc.). The PSC was designed to operate in conjunction with the Moore Products Co. Single Loop Digital Controller (SLDC) for batch control applications. The PSC acts as a "master" computer performing logic and sequencing (discrete) control while the SLDC performs regulatory (continuous) control.

An overview of the PSC

The PSC consists of five major blocks or programs (see Fig. 1):

1. Executive
2. Logic
3. Batch Sequencer
4. Input/Output (I/O)
5. Communications

The executive program acts as an operating system. Logic and Batch Sequence devices execute the user's process control algorithms. The local I/O program reads inputs and generates outputs, which are local to the PSC. The Communications program is responsible for gathering information from and providing data to other stations on the LIL.

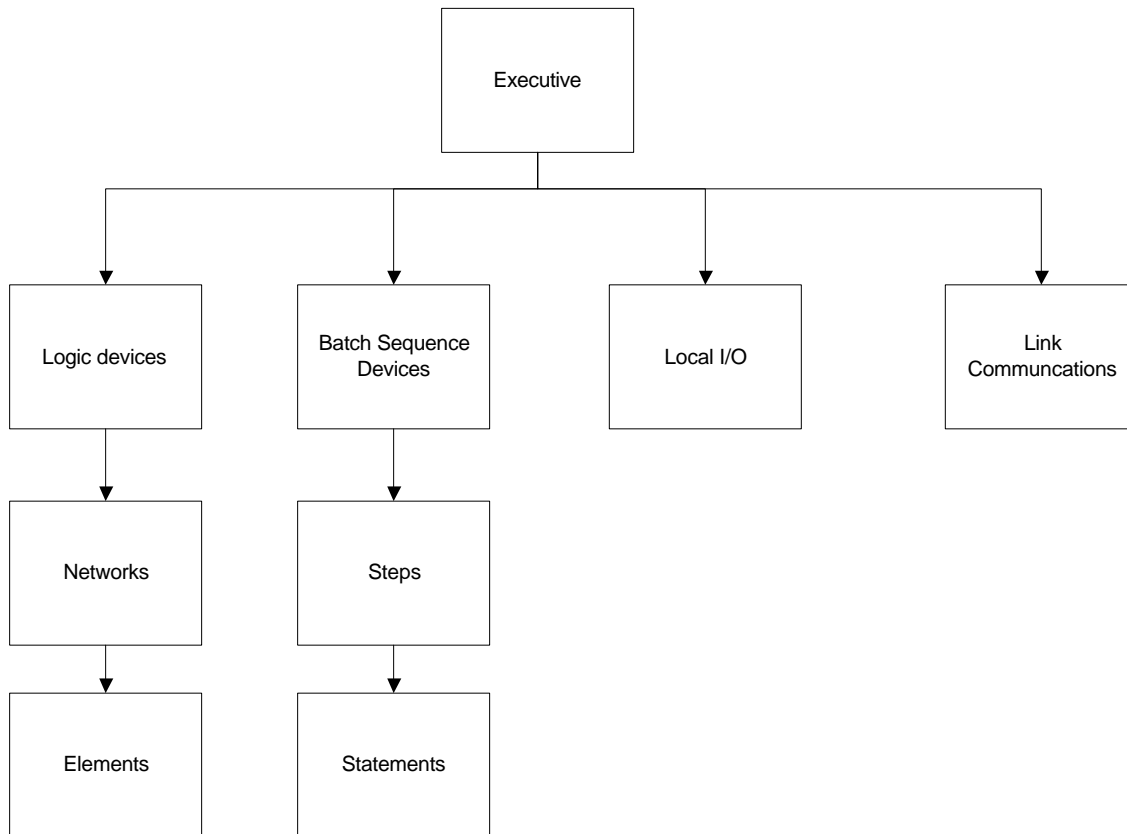


Figure 1 Programmable Sequence Hierarchy

PSC Executive program

The executive is responsible for running diagnostics, performing on-line error checking, and starting the other tasks. It first executes the communications program, then the local I/O program, the Logic program, the Batch Sequencer program, and finally the on-line diagnostics program.

PSC Logic Program

The Logic program consists of up to sixty-four Logic Devices. Each logic device consists of one or more ladder diagrams. A Ladder Diagram consisting of eight rows (or rungs) by ten columns is called a Network. All networks are executed every scan of the executive loop. Networks are primarily used for time-critical applications such as motor starters and safety interlocking. Figure 2 is an example of an abbreviated Network that controls a pump. It works as follows: The left most vertical rail is considered to be at some potential above ground, usually 120 VAC. The rightmost vertical rail is considered to be at ground potential. A series of elements, terminated by a coil, are connected in a path. When all the elements in a given path "conduct", the coil is energized, thereby performing a control action. In examining Fig. 2, rung 1 (row 1), column 1 - stop1 is a Normally Closed Contact (NCC) referenced to a Local Discrete Input Channel. Typically this input channel would be connected to a stop pushbutton located on a control panel. As soon as the button is

statements. Statement 1 indicates the user wishes to tron (turn on) start1. By turning on start1, the sequencer will write a "true" condition to the internal coil that was shown in the network, which will then let the ladder logic program start the pump. This is one method of communicating between sequencer and logic. Statement 2 indicates that this Batch Sequencer should relinquish control back to the executive. This is to allow the Ladder Logic to execute before the sequence executes statement 3. If the release statement was not placed here, statement 3 would be true and an alarm would occur (statement 4). Statements 3 and 4 are interpreted as "if contrl (a local output that controls the motor) ne (not equal) runing (an user defined constant that is initialized to TRUE) then alarm tron (turn on an alarm) *a1 (alarm 1). Statement 6 and 7 are interpreted similarly. While the tank is not full, do nothing i.e. wait until the tank is full. Statements 9 - 11 indicate some of the power built in to the PSC Batch Language. Statement 9 is interpreted as send_com (send a command over the LIL) *sta01 (to station 1) sldc (indicates that station 1 is an SLDC) loop1 (a command to loop1) ramp_time (set the ramp time in loop 1) rmptim (a user defined variable initialized to the value the user wants the ramp time to be set to). Statement 10 is interpreted similarly except that target means set the target setpoint to the value contained in tsp (a user variable). In statement 11, ramp_on means set loop 1 into the ramping mode. These three statements will tell the SLDC to ramp its setpoint from its current value to the new value over time rmptim. This will cause the temperature in the tank to rise at an even rate. Statement 12 is simply a delay loop to wait for the tank to heat up.

Statements

1 tron start1 ;start pump to fill tank

2 release ;release control until next scan

3 if contrl ne running then ;if pump not started, turn on an alarm

4 alarm tron *a1 ;set 1st alarm

5 else ; else run normally

6 while level ne full do ;while tank not full, do nothing

7 endwhile

8 troff start1 ; stop pump

9 send_com sta01 sldc loop1 ramp_time rmptim ; set ramp time

10 send_com sta01 sldc loop1 target tsp ; set target setpoint

11 send_com sta01 sldc loop1 ramp_on ; put sldc in ramp mode

12 delay prst until acc ; wait until tank is heated

```
13 endif ;
```

```
14 end ;tank is now full and heated
```

Figure 3 Example of the Batch Sequencer Language

PSC I/O Program

The PSC I/O program reads the state of the input channels and writes them to the proper table. It also reads the output tables and writes the correct values to the proper output channels. It also performs some diagnostics and error checking on the physical I/O hardware.

PSC Communications Program

The PSC Communications program performs many tasks. Each communications computer on the LIL maintains a copy of the "Global Database". The Global Database consists of information supplied by each computer on the link that is available to all other computers on the link. All information in the Global Database is updated every 1/2 second by a second processor, which is dedicated to this task. The PSC processor sends and receives commands and data through the LIL processor. The PSC processor also informs the LIL processor of which information it wishes to contribute to the Global Database.

PSC Language, Detailed examples

The initial grammar for this language was developed from the functional description of MYSL and was expanded to include ladder logic. Because of this, the PSC Language is a large language in comparison to general-purpose high level languages such as Pascal and C. Since it is a language dedicated for process control, it contains rules which do not exist in conventional programming languages. To allow the language to be mastered more easily, an integrated programming environment was developed that includes a syntax directed editor, compiler, user program documentation, and other utilities. The following is an example of a minimal user program created by the syntax directed editor.

```
config New psc 324 author Anon revision 0 timestamp 08/20/86 14:56 io endio uses enduses  
endconfig
```

A minimal source file was developed to facilitate development of parsing algorithms. Perhaps a more interesting example would be a "real" program that performs some useful work. This source file was created using the syntax directed editor.

```
config paper  
psc 324  
author Wayne  
revision 2  
timestamp 09/13/86 13:22  
;This configuration will be used as examples for a paper ;written on the PSC Language
```

io

discrete in 1 1
discrete out 1 2
bcd in 1 3
bcd out 1 4
freq in 1 5
wdt 1 6
clock 1 7

endio

uses

*dr001 alias pre1 init 10 dec
*dr002 alias acc1 init 0 dec
*dr003 alias pre2 init 10 dec
*dr004 alias acc2 init 0 dec
*dr005 alias level init 0 dec
*dr006 alias full init 0 dec
*dr007 alias rmptim init 200 rt1
*dr008 alias acc init 0 dec
*dr009 alias goo init 0 dec
*dr010 alias prst init 0 dec
*dr011 alias tsp init 70.0 pct
*co001 alias runing init 0f80 hex
*ic001 alias start1
*ic002 alias stop1
*li001
*li002
*li003 alias psw
*li004 alias ovrl
*lo001 alias motor
*lo002 alias surge
*lo003 alias motoff
*lo004 alias bckspn
*lo005 alias contrl
*lo100 alias quark

enduses

recipe redpaint

pre1 init 10 dec
acc1 init 0 dec
pre2 init 10 dec
acc2 init 0 dec
level init 0 dec
full init 0 dec
rmptim init 200 rt1
acc init 0 dec
goo init 0 dec
prst init 0 dec

```

    tsp init 70.0 pct
endrecipe
device Motor logic
    type 1
    unit 1
    ;An example of a logic device
    ;
    channel 10
    update acc1
    update acc2
    destination pre1
    destination pre2
    network network1 ;an example of a complex motor starter
        1-1 ncc stop1 ;stop switch
        2-1 trc pos start1 ;start contact
        3-1 ncc psw ;pressure switch
        4-1 ncc bckspn ;backspin protection
        5-1 ncc ovrlld ;motor overload switch
        6-1 hs 4
        10-1 coil motor nonretentive ;see rung 8
        2-2 noc motor ;holding contact
        1-3 ncc surge ;actually, this is NOT surge
        2-3 ncc stop1
        3-3 noc motor
        4-3 hs 1
        5-3 timer one pre1 acc1 ;surge timer
        6-3 hs 4
        10-3 coil surge nonretentive
        1-5 trc neg motor ;senses stop pulse
        2-5 coil motoff ;motor off pulse generator
        1-6 hs 1
        2-6 ncc motoff ;starts backspin timer
        3-6 hs 1 4-6 timer one pre2 acc2 ;backspin protection
        5-7 hs 5
        10-7 coil bckspn nonretentive
        1-8 noc motor
        2-8 hs 8
        10-8 coil contrl nonretentive ;controls the motor !
        1-1 vs
        2-1 vs
        4-1 vs
        4-2 vs
        4-3 vs
        3-6 vs
    endnetwork
enddevice

```

```

device tankex sequence
  type 1
  unit 1
  step tankfiller ;fill tank and heat solution
    1 tron start1 ;start pump to fill tank
    2 release ;release control, wait for pump to start
    3 if contrl ne runing then ;if not started, alarm
    4 alarm tron *a1 ;set 1st alarm
    5 else ;else run normally
    6 while level ne full do ;while not full,wait
    7 endwhile
    8 troff start1 ;stop pump
    9 send_com *sta01 sldc loop1 ramp_time rmpetim
    10 send_com *sta01 sldc loop1 target tsp
    11 send_com *sta01 sldc loop1 ramp_on
    12 delay prst until acc ;wait until tank is heated
    13 endif
    14 end ;tank is now full and heated
  endstep
enddevice
endconfig

```

The first four lines are known as the configuration identification section. The name of the configuration (or program) is called paper. Any name up to eight characters may be used. The second line identifies the name and model number of the target machine. Currently, this product is the only one that is programmed via this method. The third line identifies the author of the program. The name is limited to twelve characters. The third line identifies the revision number. It can be any number from 0 to 65535. Timestamp is the date and time that this configuration was last edited using the syntax directed editor.

The next section (io ... endio) identifies the type of physical I/O the user has installed in the various I/O channels. The PSC accommodates several different types of physical I/O. There are discrete input and output cards (0-5 volts, 0-120 volts), analog (BCD) input and output cards (0-9999 counts), a real time clock card, a frequency input card (used primarily for flow measurement), and a watchdog timer card. The numbers following the card type identify their physical location on the PSC I/O bus.

The next section (uses ... enduses) identifies the memory locations the user has used. The PSC language allows users to identify specific members of the internal data tables with any character string up to six characters or with "absolute" addresses (identified by a leading *). All absolute references are of the form "*xxnnn" where "xx" is a table descriptor and "nnn" is the index within the table. For instance, *dr001 means the data register table, the first data register. There are ten user accessible tables within the PSC. The tables are:

Data Register table - 512, 16 bit locations initialized at compile time, modifiable at run time.

Constant table - 110, 16 bit locations initialized at compile time.

Local Input table - 256, 16 bit locations initialized and modifiable at runtime

Local Output table - 192, 16 bit locations initialized and modifiable at runtime

Message table - 64, 20 character messages initialized at compile time

Analog Input, Analog Output tables - 128, 16 bit locations, initialized and modifiable at run time

Internal Coil table - 256, 16 bit locations initialized and modifiable at runtime

Global Input table - 256, 16 bit locations initialized and modifiable at run time

The next section (recipe ... endrecipes) identifies a group of initial values for data registers. These values can overwrite the existing data registers without changing the program. This allows users to create several different products using the same program. There can be up to twenty recipes per source file.

The next two sections (device ... enddevice) identify the control algorithms that the user programs. The first device is a logic device and it is called Motor. A logic device consists of one or more ladder diagrams. The "type 1" is used to identify a library type. For instance, the user may wish to save this motor starter in the library. If the user adds another motor of this type to the process, the logic to control the motor is already written and debugged. The user copies the device from the library and modifies the addresses within the device. The user may have up to 65,535 different devices in a library. Unit number is used to identify the process unit this device is used to control. The unit number and the library type have no bearing on the execution of the device, they are provided as a tool for the user. The next two lines are a comment field. The user may have up to 20 lines of 60 characters as a comment.

The "channel" number is used for communications purposes. Each station on the link has "devices" that reside within the station. Devices can be thought of as black boxes, which perform a desired control task (see Fig. 4). A device can receive inputs locally (physical I/O) or globally (from the communications link). Each device can also generate outputs locally or globally. Devices also have status information associated with them and they can generate alarms. Status and alarm information is always global. The user may add information to the Global Database by using "update" syntax. By specifying an address after the word update, the PSC will make the contents of that address global. Likewise, if the user wishes to have inputs to a device from the link, the "destination" syntax is used. By specifying an address after the word destination, the PSC will allow commands from the link to modify the contents of that address.

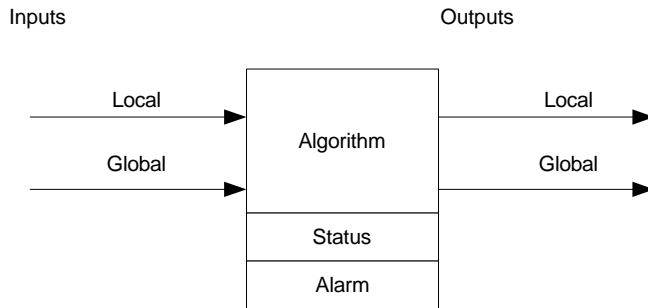


Figure 4 Generic Device

Each station on the communications link has a maximum of 256 channels by 256 parameters of communications. Some stations have devices that can send and receive commands (the PSC) while other stations have devices that only receive commands (the SLDC). Devices consume a minimum of 2 channels and a maximum of 11 channels.

If there is no entry on this line, the compiler will assign the next available channel to this device.

The remaining syntax will be explored in greater detail later on.

The last line identifies the end of the computer program. PSC Language, Meta-syntax PSC Language, Meta-syntax PSC Language, Meta-syntax

The following section describes the method of interpreting the language specification. The rules for the language are written using a pseudo Backus-Naur Form (BNF). The Meta Syntax is as follows:

- 1) Any string between '/*' and '*/' is a comment /* this is a comment */
- 2) '::=' means "defined as" /* a ::= integer: a is an integer */
- 3) '|' means alternate choices. /* a | b means a or b */
- 4) '< >' is used to group things /* <1-65535> means a number between 1 and 65535 */
- 5) '[a.b]' means repeated anywhere from a to b times /* a[1.5] means "a" or "aa" or "aaa" or "aaaa" or "aaaaa" */
- 6) '[0.*]' means an arbitrary number of times /* b[0.*] means nothing or "b" or "bb" or "bbb" etc... */
- 7) '!' means new line /* */
- 8) '{string}' means a string is optional /* a{b} means "a" or "ab" */
- 9) "..." means a regular expression /* "foo" means the string foo */

10) When a symbol name is defined in capitals, it is a rudimentary type that cannot be decomposed or subspecified into lower parts.

PSC Language, Backus-Naur specification

The PSC language is a context free grammar. A context free grammar has no ambiguity in the interpretation of a given program statement. This frees the user from worrying about how a particular program statement will be interpreted by the compiler.

The BNF specification for the entire PSC language is contained in Appendix A. Listed below is a detailed explanation of the language components.

Configuration Description

```
config_file ::= cfgid_blk io_blk uses_blk {recipe}[1.*] {device}[1.*] cfg_end
```

A config_file (configuration file) is defined as:

cfgid_blk - a configuration identification block

io_blk - a description of how to address physical I/O

uses_blk - a list of user defined variable names and their corresponding symbolic address in the PSC

recipe - an optional description of 1 or more user defined recipes. Recipes are essentially a list of variables to be re- initialized before a program is re-started. These can be used to create different products from the same process e.g. different color paints.

device - an optional description of 1 or more devices. These devices consist of collections of either ladder logic networks or batch sequence steps.

cfg_end - configuration end. The end of the current source configuration file.

Examining the cfgid_blk in greater detail,

```
cfg_blk ::= config_id config_dest author revision stamp {comment_blk}
```

The cfg_blk (configuration block) consists of the following:

config_id - This is the actual start of the program. It consists of the reserved word "configuration" then the name of the configuration, followed by a newline character.

config_dest - this consists of the reserved word "psc" followed by the model number of the product.

author - this consists of the reserved word "author" followed by a 12 characters string.

revision - this consists of the reserved word "revision" followed by the current revision number of the source code. This revision number is compiled and sent to the target machine. This permits compatibility checking between source and object databases.

stamp - this consists of the reserved word "timestamp" followed by the time and date.

comment_blk - this consists of up to 20 optional comment lines. Comment lines start with a semi-colon.

Input/Output Description

`io_blk ::= IO ! {iospec}[0.*] ENDIO !`

The `io_blk` consists of:

IO - the letters "io" followed by a carriage return

iospec - an optional description of the I/O configuration

ENDIO - the letters "endio" followed by a carriage return

going down a level, the `iospec` is defined as:

`iospec ::= io_type box_no slot_no`

where `io_type` is defined as:

`io_type ::= < DISCRETE OUT | DISCRETE IN | FREQ IN | BCD IN | BCD OUT | CLOCK | WDT >`

where `box_no` is:

`box_no ::= <1-8>`

and `slot_no` is:

`slot_no ::= <1-8>`

The PSC can accommodate eight local I/O boxes, each box containing eight slots. The user can place individual I/O cards in any order, in any slot. By including this in the language specification, it is possible for the compiler to create an array of I/O card types. This array is part of the database that is transferred to the PSC. The online executive can then verify that the user has installed the correct cards in the correct slots.

Uses Description

The uses section is used for variable declaration.

```
uses_blk ::= USES ! {data_spec}{*} ENDUSES !
```

USES - the letters "uses" followed by a carriage return

data_spec - an optional description of the users table usage

ENDUSES - the letters "enduses" followed by a carriage return data_spec is defines as:

```
data_spec ::=
```

```
< DR_ABS {ALIAS NAME} {INIT NUMBER datatype} !  
| AO_ABS {ALIAS NAME} !  
| AI_ABS {ALIAS NAME} !  
| CON_ABS {ALIAS NAME} {INIT NUMBER datatype} !  
| GI_ABS ORG {ALIAS NAME} !  
| IC_ABS {ALIAS NAME} !  
| LO_ABS {ALIAS NAME} !  
| LI_ABS {ALIAS NAME} !  
| MSG_ABS {ALIAS NAME} {INIT "20 CHAR"} !  
| ORIGIN_ABS {ALIAS NAME} ! >
```

All lines in the uses section begin with the absolute address of the memory location that that the program is using. Every element of every table that the user employs must be identified in this section, otherwise, the compiler generates an error message. The expression {ALIAS NAME} means the user can optionally define a six character identifier for this absolute address. The specific item can then be referenced by the six-character name or its absolute address. For Data Registers and Constants, the user can optionally specify a number and data type. If no entry is placed here, the initial value is assumed to be zero, with a decimal datatype. Messages can have an optional twenty character initializer.

```
DR_ABS ::= "*dr<1-512>"  
AO_ABS ::= "*ao<1-128>"  
AI_ABS ::= "*ai<1-128>"  
CON_ABS ::= "*co<1-110>"  
GI_ABS ::= "*gi<1-256>"  
IC_ABS ::= "*ic<1-256>"  
LO_ABS ::= "*lo<1-192>"  
LI_ABS ::= "*li<1-192>"  
MSG_ABS ::= "*msg<1-64>"  
ORIGIN_ABS ::= "*org<1-64>.<1-256>.<1-256>"  
STATION_ABS ::= "*sta<1-64>"
```

The above syntax contains two new items: ORIGIN_ABS and STATION_ABS. Both relate to link communications. To allow access to any station on the link, that station syntax was developed. This concept was expanded to include the "origin" of any piece of data. An origin completely describes the address of a piece of data. The origin definition above means use the numbers following the "*org" to identify the "station" <1-64>, "channel" <1-256>, and "parameter" <1-256>.

Each station has 256 by 256 matrix of communicatable information.

For the sake of convenience and consistency, portions of the data_spec section were grouped together to form logical groups.

A short source is a sixteen bit unsigned integer and can originate from a data register, analog input or output, global input, or a constant.

```
ss ::= DR|AI|AO|GI|CON
```

A coil is also sixteen bits, but it is used in a special format in the PSC. A coil can originate from an internal coil, local output, or local input.

```
coil ::= IC|LO|LI
```

The following is a list of the other groupings used in the PSC. The term long refers to a 32 bit unsigned number (two consecutive data registers or constants).

```
sd ::= DR|AO /* short dest */
cs ::= IC|LI|LO /* coil source */
cd ::= IC|LO /* coil dest */
ld ::= DR /* long destination */
ls ::= DR|CON /* long source */
source ::= ss|cs dest ::= sd|cd /* subset of source */
variable ::= <coil|ss> /* anything */
```

Sources are typically used as presets or inputs while destinations are typically used as accumulators or outputs. A variable is used to represent anything.

Constants and data registers have data types associated with them that allow the user to initialize these locations at compile time. Data types were developed primarily to allow users to easily communicate with the SLDC. The SLDC expects data to be formatted in a particular way. By using the PSC data types, the user can enter numbers in a fashion that is meaningful to the user and allow the compiler to be concerned with the specific conversion. For instance, if the user wants to send a proportional gain of 5 to a controller, a data register or constant is configured with the initial value 5 and the data type PG3. The specific details of the ranges are not important, the ability for the user to be able to easily configure the correct tuning value is. The data types, BIN, DEC, and HEX are used internally by the PSC in its calculations. PCT is a "universal" data type that is used by the link.

These SLDC specific data types are:

RA1 is used to send a Ratio tuning constant.

BIA is used for Bias tuning constant. LD1,LD2 are used for a Lead tuning constant. REL is used for a Setpoint Relative tuning constants. PG1,PG2,PG3,PG4 are used for Proportional Gain tuning constants. TI1,TI2,TI3 are used for Integral Time tuning constants. DG2 is used for Derivative Gain tuning constant. TD1,TD2 are used for Derivative Time tuning constants. RT1 is used for Ramp Time. LNG is used for Long (32 bit) data values.

datatype ::=

< HEX | BIN | DEC | PCT | RA1 | BIA | REL | RT1 | LD1 | TI1 | TD1 | DG2 | LD2 | TD2 | TI2 | TI3 | PG1 | PG2 | PG3 | PG4 | LNG >

Recipe Description

Recipes are re-initialization of data registers. When the PSC has finished processing a batch, the data registers values can be changed to allow processing of a new product.

recipe ::= RECIPE LIDENT comment ! init_spec[0.*] ENDRECIPE !

A recipe block starts with contains the reserved word "recipe" followed by a name of up to 12 characters and an optional comment. This is followed by a list of user variables and what initial values are to be placed in them. Any variable not listed in a recipe reverts back to its initial value as defined in the "uses" section. The compiler creates a new recipe by making a copy of the uses section and then overwriting the existing values with any new ones the user defined. The syntax for a recipe line is strikingly similar to the data register section of the "uses" section.

init_spec ::= DR INIT NUMBER datatype

All recipes terminate with the reserved word "endrecipe"

Device Description

Following the recipes are devices. Devices are relatively complicated, and they contain the main area of interest in the language definition. Logic and Sequence devices consist of the same basic building blocks: a description; a device type; a unit number; optional comments; optional link information; and the code section.

device ::= seq_dev | lgc_dev

lgc_dev ::= lgc_desc dev_type unit_desc {comment_blk} {lil_blk} lgc_code device_end

seq_dev ::= seq_desc dev_type unit_desc {comment_blk} {lil_blk}

seq_code device_end

Devices contain a description. This description consists of the reserved word "device" followed by a user defined name, followed by the reserved word "logic" or "sequence".

lgc_desc ::= DEVICE NAME LOGIC ! seq_desc ::= DEVICE NAME SEQUENCE !

Following the description is the library type and unit number. The library number is defined as the reserved word "type" followed by a number and a newline. The unit description is similar.

def_type ::= TYPE NUMBER ! unit_desc ::= UNIT NUMBER !

The user is allowed to specify a block of up to twenty lines each line being up to sixty characters as a device description or comment. A comment is delimited by a ";" and a newline.

comment_blk ::= {comment}[1.20] comment ::= ';' {CHAR}[1.59] !

Following the comment is the link communications information. A lil_blk consists of: a starting channel number, a list of globally updating variables, a list of command destinations (for logic devices only), and an emergency step to goto (for sequence devices only). All information in the lil block is optional.

lil_blk ::= {channel_no} {update_set}[0.*] {parameter_set}[0.*] {emerg}

channel_no ::= CHANNEL <5-242> !

update_set ::= UPDATE variable !

parameter_set ::= DESTINATION <DR | cs> ! emerg ::= EMERGENCY STRING !

In the PSC, the user is allowed to specify the channel number that this particular device will be seen at in the global database. If this information is not entered, the compiler automatically assigns the next available channel to this device. This method has some drawbacks. For instance, a user configured some devices and allows the compiler to assign the starting channels. The channels are assigned as follows:

device a channels 4 -10 device b channels 11-14 device c channels 15-20

The user then decides to add one piece of global information to device "a". Device "b" would then start at channel 12 and device "c" would start at channel 16. Any other device on the link configured to talk to devices "b" or "c" would have to be modified and given the new addresses. If the user had configured these devices with explicit starting channels, extra channels could have been allocated to each device too allow for future expansion.

Update_set allows the user to specify up to eight pieces of data to be globally broadcast on the link. These broadcast values can be used to trigger actions in other controllers or be displayed and manipulated on an operators console.

Parameter_set is used in logic devices to allow users to change nonglobal variables via a command from another device on the link. These values come from an operators console or a customer's host computer. Typical applications are to allow the users to change the preset value for a counter or timer.

"Emerg" is only valid for sequence devices. It allows the user to specify which step this sequencer is to unconditionally branch to upon receiving an "emergency" command from some device on the link. Again, this typically comes from an operator's console.

The only remaining parts of the language are the logic code, sequence code, and end device. End device is just the reserved word "enddevice". Below is the syntax for logic code.

Logic Device Description

```
lgc_code ::= {network}[0.128] !
network ::= network_desc {element_line}[0.*] ENDNETWORK !
NETWORK_NAME ::= LIDENT
network_desc ::= NETWORK NETWORK_NAME {comment} !
```

Logic code, "lgc_code", consists of a group of 0 to 128 networks. Each network consists of a network descriptor, element lines, and the reserved word "endnetwork". A network descriptor consists of the reserved word "network" and a user defined name, followed by an optional comment and a newline.

Element lines consist of a column number, a dash, a row number, a logic element and an optional comment.

```
element_line ::= col_num '-' row_num element {comment} ! row_num ::= <1-8> col_num ::= <1-10>
```

The column-row numbering was derived from the way in which the processor executes the logic code i.e. ladder logic executes in a column by column fashion. Elements are the actual function blocks that the user configures to control the process. Elements are grouped in the BNF according to similar functionality.

The first group is used to perform basic logical decisions such as "and", "or", and "not". These elements (except for the vertical shunt) consume one column and one rung in a ladder diagram. Starting with the first element in the BNF, VS stands for Vertical Shunt. A Vertical Shunt logically "or's" two rungs together. For example:

```
      x
--- | | ----+---
      y   |
--- | | ----+
```

The logical expression for this configuration is x or y.

The source code for this diagram is:

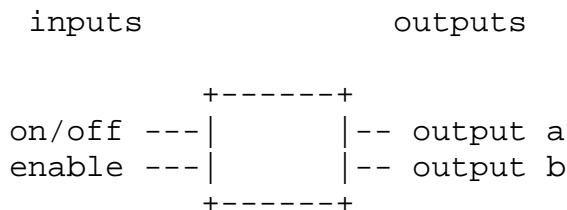
```
... 1-1 noc x 1-2 noc y 1-1 vs ...
```

The VS has the same column-row location as the NOC referenced to "x". This is the only element that can occupy the same location as another. It is best thought of as a modifier.

NOC stands for Normally Open Contact. A NOC is referenced to an internal coil, local input or output, or a global input. If the reference associated with the NOC is off, power is interrupted. In the example above, if "x" was a coil and it was turned off, current would not flow through the contact. The logical complement to the NOC is the NCC or Normally Closed contact and when the reference to a NCC is off, current would flow. TRC stands for TRansitional Contact or one shot. A TRC can detect a positive or negative transition on an internal coil or local input or output. A TRC will pass current for exactly one scan of the executive loop, if the desired transition has occurred. HS stands for Horizontal Shunt. A HS has a length associated with it and it is used to extend the current value of the rung (a no-op). The inverter inverts the current state of the rung. The BNF for the first group of elements is:

```
element ::= < VS | NOC <cs | GI> | NCC <cs | GI> | TRC {POS|NEG} cs | HSHUNT hs_length | INVERT
```

The second group of elements are the timers and counters. These elements consume one column and two rows in a ladder diagram. Timers and counters have a preset value and an accumulator value. The preset identifies the value to which the user wishes to time or count to. The accumulator is the current value. Timers may be configured with a 1/10th second or 1 second timebase. Counters may count up to a preset or from a preset down to zero. When the enable input is off (see the figure below), the accumulator set to zero in the up counter and timers. When the enable input is off for a down counter, the accumulator is set to preset. When the elements are enabled and off, the current accumulated value is maintained and when the elements are enabled and on, the element times or counts. The outputs indicate the relationship between the preset and accumulator. For timers and the up counter, the "a" output is "off" when the accumulator is less than the preset and "on" when the accumulator is greater than or equal to the preset. The "b" output is the converse of "a". For a down counter, the "a" output is zero for the accumulator greater than zero, and one when the accumulator is equal to zero. The accumulators for timers and counters are stored in non-volatile RAM. When power to the PSC is lost, then restored, the value will remain intact. If the user specifies that a timer/counter is NONRETENTIVE, the accumulator will be set to zero by the power up executive.



The BNF of the timers and counters is:

```
| TIMER <ONE|TENTH> ss sd {NONRETENTIVE}
| COUNTER <UP|DOWN> ss sd {NONRETENTIVE}
```

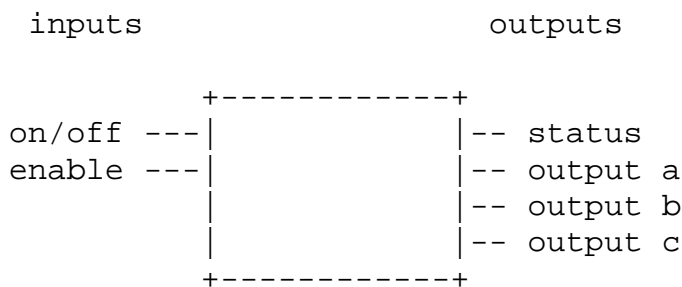
The next element encountered in the BNF is the totalizer. The totalizer represents a radical departure from traditional ladder logic functions and is considered to be in a group of its own. The BNF for it is complex and it contains several special cases. The totalizer consumes two columns and four rows.

The BNF is broken into five sections. TOTAL indicates that this element is a totalizer, "ss" is the address of the source value to be totalized. The next expression, contained within the "<>" indicates the type of presets the user wants to use for this totalizer - BCD input channels or long sources. The accumulated value is held in "ld" and the user may optionally specify that the total value be displayed on local BCD outputs.

The totalizer generates a thirty-two bit accumulated value therefore, the presets must also be thirty two bits. The only user configurable thirty two bit memory locations are data registers and constants. If the user wishes to use BCD inputs as presets, a routine to must convert the two sixteen bit values from the input channels to a thirty two bit value. The routine reads the two BCD input channels and stores the converted value in two consecutive data registers. The data registers are then used as the presets for the totalizer.

A totalizer accumulates a thirty two bit value by successively adding a 16 bit source to the accumulator every scan that it is executed. Its inputs work exactly the same as the timer/counter. It uses two different presets, a high preset and a low preset that are used to control the outputs. When the totalizer is totalizing, all outputs are on, except the "c" output. When the accumulated total reaches the low preset, the "b" output goes low and when the accumulated total reaches the high preset, the "a" output goes low while the "c" output goes high.

As an option, the totalizer can generate a second accumulated value, which is compatible with the BCD outputs.

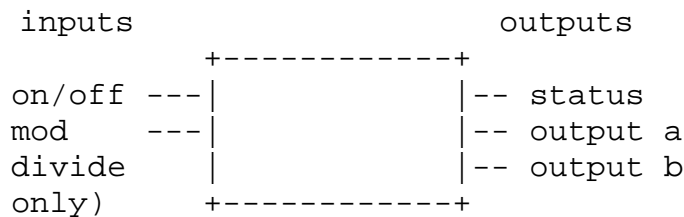


BNF for the totalizer:

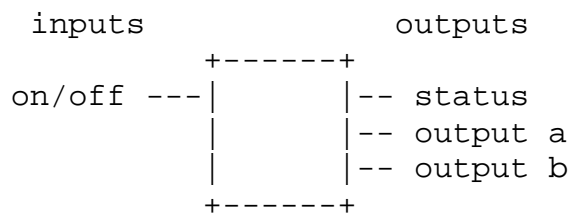
```
TOTAL ss <BCD LOW AI DR HI AI DR>|<BINARY LOW ls HI ls> ld {BCD AO}
```

The next group of elements are the math and comparison operators. The add, subtract, and comparison operators use one column and three rungs, while the multiply and divide use two

columns and three rungs. The reason the multiply and divide require extra columns is because they use thirty two bit numbers. The multiply generates a thirty-two bit quotient and the divide divides a thirty-two bit number by a sixteen bit number generating a sixteen bit quotient and a sixteen bit remainder. The divide has an additional input (mod) that determines whether the remainder is a whole number remainder or fractional remainder. The status output is on whenever the element is enabled. The "a" output is on whenever there is an overflow in multiplication or division. Output "b" indicates a divide by zero in the divide element. It is unused in the multiply.



The rest of the math and comparison elements operate on sixteen bit numbers exclusively. The add and subtract elements generate a short destination based on the result of the operation. The "status" output reflects the value of the on/off input. No operation is performed when this input is off. The "a" output is on whenever there is an underflow in subtraction or an overflow in addition. Output "b" is unused in the add and subtract. For the comparison elements, the "a" output is on whenever the comparison is false and off when the comparison is true. The "b" output is the complement of "a".



The BNF for the math and comparison operators is:

```

| ADD ss TO ss GIVING sd
| SUB ss FROM ss GIVING sd
| MUL ss BY ss GIVING ld
| DIV ls BY ss GIVING ld
| IF source math_op source DIDDLE sd

```

Where math_op is defined as:

```

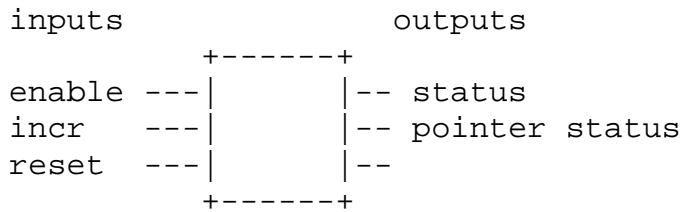
math_op ::= GT | LT | GE | LE | EQ | NE

```

The next group of elements are the transfers. Transfers consume one column and three rows. There are four types of transfers: Register to Table; Table to Register; Table to Table; and Block. Register to Table moves one sixteen bit piece of data into a table. The length of the table is determined by the user at configuration time. It can not be changed dynamically. The location at which the data is stored in the table is based on a pointer into the table.

Upon examination of the BNF below, "tsize" is the size of the table; "source" is the address of where the input data is to be gotten from; "dest" is the address of where the output data is to be stored to; and "ss" is the offset or pointer into the source or destination table.

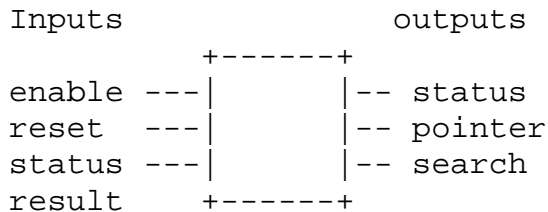
The Register to Table moves data from one source to a table. The Table to Register works in the opposite manner. The Table to Table Transfer and the Block Transfer move a group of registers within one table to another table. The Table to Table moves one element each scan while the Block moves all elements in one scan. The Block Transfer does not require a pointer. The pointer increment input, when on, inhibits the built in automatic pointer increment function. The pointer reset input sets the pointer or offset value to zero, indicating the beginning of the table. The status output reflects the value of the enable input and the pointer status indicates that the pointer has exceeded the table length.



The BNF for the transfer elements is as follows:

- | TRANSFER REG TABLE tsize source dest ss
- | TRANSFER TABLE REG tsize source dest ss
- | TRANSFER TABLE TABLE tsize source dest ss
- | TRANSFER BLK tsize source dest

The next element is the Table Search. The Table Search consumes one column and three rows and it uses the same types of comparisons as the math comparison element. It compares sixteen bit values in a table against some other value in the PSC. The comparison starts at the the current pointer value and continues until a match is found or the table pointer exceeds the table size. It is similar BNF-wise to the transfer elements. The reset input resets the pointer prior to scanning the table. The status output reflects the enable input. The pointer status output turns on when the pointer has exceeded the table size. The search result output turns on if the comparison was true.

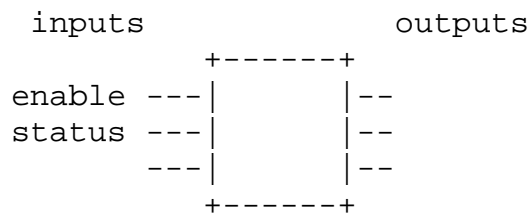


The BNF for the search function is:

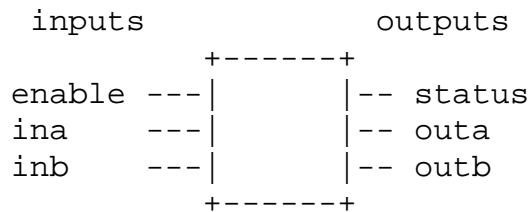
- | SEARCH tsize source math_op source ss

The next group of elements are the Matrix operators. They consume one column and three rungs. The PSC defines a matrix as a table of data registers that are accessed on a bit by bit basis. The matrix operators allow the user to AND, OR, EXCLUSIVE-OR, and COMPLEMENT matrices as well as modify or sense a given bit in a matrix. Two matrices of equal size may also be compared. Upon examination of the BNF, the matrix AND, OR, XOR, and COMPLEMENT specify the source and destination matrices and the size. The result of the operation is stored in the destination matrix. The matrix modify, sense, and compare must have access to a given bit, so a pointer is required.

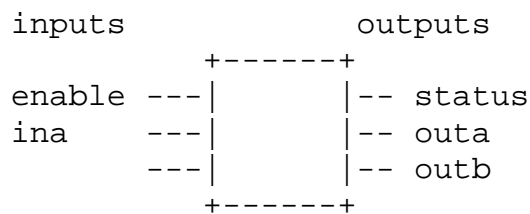
The matrix AND, OR, XOR, and COMPLEMENT have one input and one output. If the enable line is on, the operation is performed and the status output turns on.



The Matrix modify, when enabled, will look at the ina input and inb inputs. If ina is on, the bit pointed to by the pointer will be set, otherwise, it will be reset. If the inb input is off the matrix pointer will be incremented, otherwise, it will be left alone. The outa output reflects the ina value. The outb output will turn on when the pointer exceeds the matrix boundary.



The Matrix sense element, when enabled, will sense the value of the bit at the pointer. The value of outa will reflect the value of the bit. The value of ina will then be looked at and if it is off, the matrix pointer will be incremented. If the matrix boundary is exceeded, outb will turn on.



The BNF for the Matrix elements is :

```

| MATRIX matrix_op tsize source dest
| MATRIX MODIFY tsize dest ss
| MATRIX SENSE tsize source ss

```

| MATRIX COMPARE tsize source source ss

The next group of elements are the scalers. Scalers consume one column and two rungs. They convert PSC data into link compatible data (output scaler) and link data into PSC data (input scaler). The link data is in the form of 80 hex to F80 hex (128 - 3968 decimal). PSC data, for the most part, is 0 hex to 270F hex (0 - 9999 decimal). The link data range was determined from design of previous products. The PSC was required to conform to it. The PSC data range was chosen from the maximum value of a four digit BCD input. The scaler reads the source value, converts it to the appropriate value and stores it in the destination. The status output reflects the enable input. The input status output is only used in the input scaler and indicates that the input value is out of range.

```
inputs                outputs
                    +-----+
enable  ---|          |--
status  ---|          |-- input
status   +-----+ (input scaler only)
```

The BNF for the scalers is:

| SCALE IN ss sd
| SCALE OUT ss sd

The next group of elements are the Fans. The Fans consume one column and two rungs. The Fan In packs twelve consecutive source words into one sixteen bit word (the upper four bits are unused). If the source word is greater than or equal to 800 hex (mid value of the 80 - F80 hex range), the corresponding bit is set. The Fan Out unpacks a twelve bit word into twelve consecutive destinations. If a bit is set, F80 hex is written to the destination, otherwise, 80 hex is written to the destination. The status output reflects the enable input. When the input is on, the translation takes place.

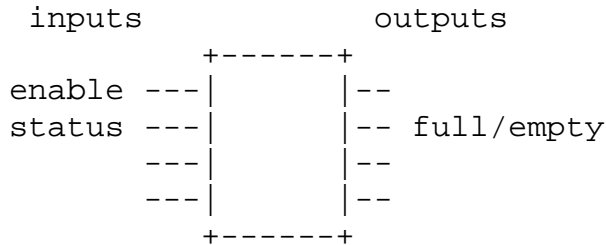
```
inputs                outputs
                    +-----+
enable  ---|          |-- status
        ---|          |--
        +-----+
```

The BNF for the Fans is:

| FAN IN source dest
| FAN OUT source dest

The PSC has some basic queuing elements, the FIFOs (first in - first out). Items can be added and removed at anytime, the fifo algorithm ensures that the first item pushed into the queue will be the first item pulled from the queue. The FIFOs consume one column and four rungs. There is a FIFO Input and FIFO Output element. The size of the FIFO is determined at configuration time and can not be dynamically changed. It can only hold n-1 items when full. Upon examination the BNF for the FIFOs, the source is the address of the input data for the queue. The "que" consists of a head,

size, tail, and the starting address of the queue. The head is a pointer to the next available location in the queue, while the tail is a pointer to the next element to be removed from the queue. The starting address is at which data register the queue is to start. Both FIFOs have only one input, the enable. The status output reflects the enable input value. The full output is used on the FIFO Input to indicate that there is no more room in the queue. The empty output on the FIFO Output indicates that there are no more elements available to remove from the queue.



The BNF of the FIFO is:

```

| FIFO IN source que
| FIFO OUT dest que

```

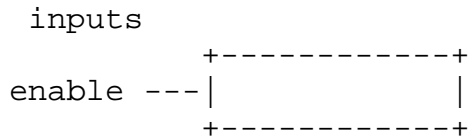
Where que is defined as:

```

que ::= DR tsize DR DR /* HEAD tsize TAIL START */

```

The next element is called the Skip. The Skip is a goto that only allows the program to jump forward. The Skip consumes two columns and one rung. The double width is necessary to allow the user to specify a network name to skip to. Network names can be up to twelve characters. When the skip is enabled, there is an unconditional branch to the network specified.



The BNF for the skip is:

```

| SKIP LIDENT

```

The next element is the alarm. The user can configure up to four alarms per logic device. If the input to the alarm is on, the alarm and the nak (not acknowledge) bits are set, if the input is off, the alarm bit is reset. By using this method, unacknowledged alarms can be detected; e.g. an alarm condition occurs, then goes away, the operator will still be notified that an alarm occurred because the nak bit will still be set. In logic alarms, only an external device can reset the NAK bit. Alarm words are broadcast on the LIL and are available to all stations on the link. alarm

```

---<A>---

```

The BNF for the alarm is:

```
| ALARM ALARMNO
```

The last logic element is the coil. The coil is used to turn local discrete outputs on and off. The coil element can also write to the internal coil table, which is a table identical to the local output table, but has no physical connection to the outside world. A coil can be nonretentive such that the power up executive will turn the coil off if power is lost, then restored.

coil

```
---( )--- non-retentive  
---(R)--- retentive
```

The BNF for the coil is:

```
| COIL cd {NONRETENTIVE}
```

```
> /* end of element list */
```

Sequence Device Description

The sequence code consists of 0 to 256 steps. A step consists of a step descriptor, 0 to 20 statement lines, and the reserved word "endstep". A step descriptor is the reserved word "step", followed by a user defined 12 character name and a new line. A statement line consists of a statement number (1-20), a statement, an optional comment, and a new line.

```
seq_code ::= {step}[0.128] !  
step ::= step_desc {statement}[0.20] step_end  
STEP_NAME ::= LIDENT  
step_desc ::= STEP STEP_NAME {comment} !  
step_end ::= ENDSTEP !  
statement_line ::= statement_num statement {comment}!  
statement_num ::= <1-20>
```

Statements look like a marriage of a traditional high level language and specialized batch control statements. They are divided into three types: simple, multi-statement, and link communications. Simple statements are just that, simple. They perform the desired action and are relatively easy to understand. Multi-statement statements such as "if" and "while" are used to logically join simple statements into a collection that can control more complex tasks. The link statements perform all communications with other devices on the link.

The first statements are the set and reset. The set and reset are the sequence equivalent of the logic coil. Set turns a coil on, reset turns a coil off. There is no provision for nonretentiveness in the sequence language as there is in logic. These statements can control up to twelve coils at one time.

Set and reset are currently defined as the reserved words "tron" and "troff" respectively. The other representation for set and reset (open, close, etc.) have not been implemented.

```
statement ::=  
< set {cd}[1.12]  
| reset {cd}[1.12]
```

```
set ::= <OPEN|START|TRON>  
reset ::= <CLOSE|STOP|TROFF>
```

An example of a line of source code to represent a set statement might look like:

```
1 tron *lo001
```

Which means turn on local output number one.

The next statement, the alarm, is slightly more sophisticated than the logic alarm. The logic alarm only had the capability of setting and resetting the alarm bit. The sequence alarm has the additional capability of reset the nak bit.

```
| ALARM <set|reset> {NAK} ALARMNO
```

The move statement has the capability of moving one sixteen bit word from one type of source to a destination of the same type.

```
| MOVE <<cs TO cd> | <ss TO sd>>
```

The delay statement performs a delay loop.

```
| DELAY sd UNTIL ss
```

The next statement skips the remaining statements in the current step and transfers execution to the beginning of the next step.

```
| NEXT
```

The goto operator performs an unconditional branch to the designated step. The user fills in the name of the desired step, which is a long identifier (12 characters).

```
| GO_TO LIDENT
```

The emergency statement is the same as the goto except that this statement sets the emergency bit in the sequence status word. There can be multiple emergency statements in any sequencer, e.g. there can be different emergency steps for different levels of emergency processing.

```
| EMERGENCY LIDENT
```

The manual mode statement puts the sequencer executing this statement into the manual mode. No further processing will take place until the sequencer is returned to automatic by some external device.

| MANUAL MODE

Release terminates the execution of the current sequencer. It returns control the operating executive.

| RELEASE

The parallel operator initiates a parallel sequence. The PSC can have up to four parallel sequences running along with each of the five main sequences. When a parallel sequence is running, the time allotted by the executive to the main sequence is unchanged. The overall effect is that the parallel sequences will each be given a turn to run when the main sequence is called. This creates the appearance of the main sequence and the four parallel sequences running in parallel. The endparallel is used to terminate a parallel sequence

| PARALLEL LIDENT | ENDPARALLEL

The following statements are called multi-statement. They may cross step boundaries and consist of at least a pair of statements (if ... then , etc). The if ... then ... else, while ... do, and the repeat types of decisions: "source op source"; "NOT source"; and "source MEQ source MASK ss". The "source op source" decision is similar to other high level languages. Note, however, that this syntax can only allow one decision per statement. Multiple decisions must use multiple statements. It is possible to nest these statements up to twenty levels. The "Not source" decision examines the contents of the "source" and if the value is less than 800 hex (mid value of 80 - F80 hex), the decision is true (greater than or equal to 800 hex is considered true). The "source MEQ source MASK ss" decision is primarily used to interrogate status and alarm words from other station on the link. A source to be bit tested is specified as well as a bit mask and a compare mask. The bit mask specifies which bits in the source are of to be tested (1 means test the bit, 0 means do not test the bit). The compare mask specifies to which state the bit is to be tested (1 means test the bit for 1, 0 means test the bit for 0). For example, if the bit mask is 0000 0000 0000 0001 binary and the compare mask is 0000 0000 0000 0000 binary, this means that the user wishes to test the first bit in the source to see if it is zero.

All multi-statements reference the nonterminal "block". A "block" is a group of statements that can cross step boundaries. A "block" is defined as a group of statements and it can extend over step boundaries. It was defined to allow the user the capability of writing a group of statements without concern as to where it starts or ends.

```
block ::= statements { step_end steps step_desc statements } statements ::= {statement_line}[*]  
steps ::= {step}[*]
```

```
| IF source op source THEN {comment} ! block { ELSE {comment} ! block } ENDIF
```

```
| IF NOT source THEN {comment} ! block { ELSE {comment} ! block } ENDIF
| IF source MEQ source MASK ss THEN {comment} ! block { ELSE {comment} ! block } ENDIF
```

```
| WHILE source op source DO {comment} ! block ENDWHILE
| WHILE NOT source DO {comment} ! block ENDWHILE
| WHILE source MEQ source MASK ss DO {comment} ! block ENDWHILE
```

```
| REPEAT {comment} ! block UNTIL source op source
| REPEAT {comment} ! block UNTIL NOT source
| REPEAT {comment} ! block UNTIL source MEQ source MASK ss
```

Other multi-statements are the timer and counter. Both of these statements use the "block" notation and will execute the statements in the "block" until the accumulator reaches the preset.

```
| COUNTER sd UNTIL ss {comment} ! block ENDCOUNTER
| TIMER sd UNTIL ss {comment} ! block ENDTIMER
```

The final group of statements are the link communication commands. The send statement allows the knowledgeable user to send any type of command to any station on the link. Most commands are sent with the "computer" privilege, but certain commands require the "any" privilege. Link_op describes the type of data the user wishes to send. Abs means sixteen bit integer absolute; set (actually, tron) means mask a certain bit on; reset (troff) means mask a bit off; relative means twelve bit integer relative; range12 means twelve bit integer absolute with range bits for the data stored in bits twelve and thirteen; and range14 means twelve bit integer absolute with range bits for the data stored in bits fourteen and fifteen. Org describes the link address that the data is to be gotten from. Obviously, this command is not for the casual user.

```
| SEND privilege link_op source TO ORG
```

privilege ::= <COMPUTER | ANY>

link_op ::= ABS | set | reset | RELATIVE | RANGE12 | RANGE14

The message statement is used to send a user defined character string to another station on the link. Msg identifies the specific message to be sent.

```
| MESSAGE MSG TO ORG
```

Get external is used to retrieve a non-global piece of data from another station on the link. The get external may optionally send a message, typically a prompt, to another station on the link. It can also be configured to verify that the data received is within a desired range and that the data was received within a specified time period.

```
| GET_EXT ORG dest {MSG TO ORG} {BETWEEN ss AND ss UNTIL ss}
```

The send_com statement is currently designed for SLDC specific communications. This statement has the capability of being expanded for use with other controllers. From the syntax directed editor, the configuration of these commands is all menu driven. The user does not need to be concerned with the details that the generic command require, all of the information is built in to the compiler. For instance, if the user wanted to send a command to an SLDC to tell it to go into the automatic mode, the syntax for the SLDC specific command would look like:

```
send_com *sta01 sldc station auto
```

The generic command, on the other hand would require the following syntax:

```
send any tron *dr001 to *org01.009.001
```

and *dr001 would have to be initialized with 1 hex.

The SLDC specific command reads: "send a command to station 1, which is an SLDC type station, and place the station device into automatic". The generic command reads: "send data register 1 as a mask on command type with the any source to station 1, channel 9, parameter 1". With the specific command, the user need not be concerned with the command type, command source, the address within the station the command is to be sent to, or the specific data value that needs to be sent. With any of the bit manipulation commands in the SLDC specific statements, constants are built into the compiler so the user does not have to declare any of the user variables or constants as data values to be sent. Upon examination of the BNF, sta represents the station address of the SLDC that this command is to be sent to. Station type describes the type of station that this command is being sent to. Currently the SLDC is the only product this statement is set up for. The dev is the internal device to which the command is to be addressed to. The SLDC has seven internal devices. The commands are somewhat complicated. Certain commands are only valid to certain devices. The syntax directed editor allows the user to select from a command menu after a specific internal device has been selected. The optional command_data nonterminal allows the user to send specific data values when necessary. Not all commands require dat to be sent, as can be seen in the above example. This paper will not address the specifics of the SLDC and what each of the commands mean. The language has provided a powerful means for the user to control the SLDC.

```
| SEND_COM STA station_type dev command {command_data}
```

```
station_type ::= SLDC /* | SLC | PSC (maybe later) */
```

```
dev ::= <STATION | LOOP1 | LOOP2 | CHAN_A | CHAN_B | CHAN_C | CHAN_D>
```

```
command ::= < COMMAND | AUTO | MANUAL | COMPUTER >
```

```
COMMAND ::= < HOLD | RUN | CLEAR_ERR | FLASHER_ACK | LOCAL | C_COMPUTER  
| C_AUTO | C_MANUAL | CONSOLE | INTERNAL | EXTERNAL | ALARM_OPERATIONAL  
| ALARM_OUT_OF_SVC | RATIO | BIAS | LEAD | LAG | OUT_70 | OUT_71 | OUT_72
```

```
| OUT_73 | OUT_74 | OUT_75 | OUT_76 | OUT_77 | RAMP_ON | RAMP_OFF | PROP_GAIN  
| INT_TIME | DERV_TIME | DERV_GAIN | MAN_RESET | SETPT_ABS | SETPT_REL  
| TARGET | HI_LIMIT | LO_LIMIT | VALVE_ABS | VALVE_REL | RAMP_TIME  
| ALARM_EN | ALARM_DIS | ALARM_ACK >
```

The final statement is the end statement. When executed, this statement places the sequencer out of service, indicating that it has finished execution the program.

```
| END > /* end statement list */
```

Development of the Compiler

The translation of the formal grammar into a compiler was a straightforward, but not trivial, process. This compiler was developed using a parser generator, YACC (Yet Another Compiler Compiler). YACC takes a set of rules and actions and creates a parser, `yyparse`, which is a "C" function. Rules are derived from the BNF, and in many cases, the rules can be an exact copy of the BNF. Actions are "C" code written to perform a desired function and may occur at anytime within a rule. For example, the BNF for the entire configuration is:

```
config_file ::= config_blk io_blk uses_blk {recipe}[0.*] {device}[0.*] config_end
```

while the corresponding YACC rule is:

```
config_file  
  
    : config_blk  
  
    {  
        init_cfg();  
    }  
  
    io_blk uses_blk recipes devices config_end  
  
    ;
```

There is almost a one to one correspondence between the BNF and the rule. The expression between the { } is called an action. In this case, the action is to perform a function call, `init_cfg()`. This rule indicates that this call to `init_cfg()` must take place after the `config_blk` is parsed and before the `io_blk` is parsed. It is easy to see that YACC is extremely flexible in allowing the compiler designer to execute code at anytime during the parsing.

Another feature of YACC is built in recursion. From the above example, there can be zero or more recipes, `{recipe}[0.*]`. The YACC source defines recipes as:

```
recipes  
:  
;
```

```
| recipes recipe
```

```
;
```

The ":" followed by nothing indicates an empty rule, i.e. no recipes. The "|" indicates an "or" condition. The first nonterminal is "recipes" which calls the nonterminal that is being defined. This definition means that "recipes" consist of zero or more "recipe".

YACC has a built in error handling mechanism and YACC reserves the token "error" for this purpose. The designer may place the error token where errors are expected to occur. When an error occurs, YACC attempts to recover by popping states off the state stack until error is a valid state. It then executes the user action and attempts to resynchronize by discarding any further tokens in the current rule. For instance, the following abbreviated YACC rule represents one way of handling errors:

```
use_line
```

```
: abs_ref default_init nl
```

```
| abs_ref ALIAS STRING default_init nl
```

```
| error
```

```
{
```

```
    comperror(SERIOUS,29,uselineno);
```

```
}
```

```
;
```

In this rule, a user defined variable consists of an absolute address, an optional variable name, and an initialization. If, for instance, the user used a reserved word for a variable name, the "alias" state and the "abs_ref" state would be popped from the state stack and the use_line : error state would be a valid state. The parser would then execute the action, comperror(), which prints an error message on the console and printer. The parser would then discard the remaining tokens on the bad line, and then resynchronize on the next line. This allows the compiler to find any other errors and report them to the user.

An integral part of any part of a compiler is the lexical analyzer. The lexical analyzer is a front end program that reads the source, character by character, until it matches a pre-determined sequence of characters called a regular expression. The lexical analyzer for this language was developed using a lexical analyzer generator, Lex. Lex is another tool which is supplied with a source file that contains a set of specifications about what regular expressions it is to match and what to do when it matches an one. This file is run through Lex and it produces a file containing "C" code which performs the analysis. The compiler, yyparse, calls the lexical analyzer, yylex, which scans the input file until it matches one of the expressions. When an expression is matched, yylex returns

some value to yyparse. The values returned by lex are called "tokens". Tokens are defined in the YACC source file. When a file is YACCed, YACC produces a header file, y.tab.h, that contains numeric definitions for all of the tokens. This provides a convenient method of communication between the lexical analyzer and the parser. In many instances, however, the parser needs more information than just a token. YACC and Lex have additional communications mechanisms, namely, yylval, yytext, and yyleng. The variable yylval is actually a stack that YACC maintains. This stack can be typed to any valid "C" datatype, thereby allowing the compiler designer another level of flexibility. The variable yytext is a pointer to the character string that Lex matched, while yyleng is the length of this string. These variables are global, so they are available to any of the compiler program.

The following is an example of a Lex input specification:

```
[\-0-9][0-9a-fA-F\.]* {  
    yylval = atoi(yytext);  
    return NUMBER;  
}
```

The brackets represent grouping, `[\-0-9]` means a minus sign or a single digit that is 0 to 9. The `"\"` tells Lex to interpret the next character literally; Some characters such as `"." "-" "*" "` are used by Lex in defining regular expressions. The rest of the expression means a digit or character (a-f for hex numbers) or a decimal point `"\"`. The `"*"` tells Lex that the previous expression can be repeated zero or more times. This is rule means that the token "NUMBER" consists of at least one minus sign or digit followed by an optional string of digits, the characters a-f, or decimal points. The following expressions will match "NUMBER":

```
12234455  
0  
-.1  
-.abde  
0.....
```

The first three expressions are meaningful numbers while the last two expressions are meaningless. Lex is not responsible for making sense of the input, only for matching regular expressions.

The expression between the `{ }` is the same as a YACC action. In this rule, yylval is set to the return value of the function atoi and then Lex returns the token "NUMBER". The parser sees the token "NUMBER" and knows that the lexical analyzer has set yylval to some value. If the parser wants to know the exact expression that yylex matched, it look at yytext.

Conclusions

To use the formal grammar method, the designer(s) must write the BNF at the outset of the project. This requires the designer(s) to examine the functional description of the language in detail and resolve any questions early on. The structure of the BNF guides the designer(s) towards creating a

design that contains fewer inconsistencies. This, in turn, helps the customer to understand the final product.

Another benefit of the BNF is the use of an ASCII source file rather than some proprietary format. ASCII is easy to read and manipulate and using it allowed easy creation of test configurations during compiler development.

A third benefit of developing this way is maintainability. The BNF provides a common design document for all developers. Changes to the BNF are usually easy and painless and the concepts are easy to understand. New personnel added to the project are quick to comprehend the scope of the project. The BNF is compact, yet contains a high informational density.

References

Heckel, Paul, Elements of Friendly Software Design Warner Books, 1982.

Hunter, The Design and Construction of Compilers.

Pratt, Terrence W., Programming Languages: Design and Implementation, Prentice Hall, 1975.

Schreiner and Friedman, Introduction to Compiler Construction with Unix, Prentice Hall, 1985.

Waite, William M. and Gerhard Goos, Compiler Construction, Springer-Verlag, 1984.

Zarella, John, Language Translators.

System V/68 Documentation, Volume 3, Section 8 Lexical Analyzer (LEX), Motorola Microsystems, 1984.

System V/68 Documentation Volume 3, Section 9 Yet Another Compiler Compiler (YACC), Motorola Microsystems, 1984.

Appendix A Backus-Naur Specification

/* Meta Syntax - '/' at the end of a line is a line continuation character Any string between '/*' and '*/' is a comment '::=' means "defined as" | means alternate choices. Also alternate choices are on different lines. < > is used to group things [a.b] means repeated anywhere from a to b times [0.*] means an arbitrary number of times {string} means a string is optional <#L-#H> means a numeric range, e.g. any number from #L to #H "... " means a regular expression: #=number, =alpha, &=alphanum When a symbol name is defined in lower case, that symbol is a non-terminal for YACC. When a symbol name is defined in UPPER case that is a terminal symbol (token) for YACC and is decoded by LEX.

--> any new comments added by wdw are bounded by /** **/ */

config_file ::= cfgid_blk io_blk uses_blk {recipe}[0.*] {device}[0.*] cfg_end

! ::= <//*new line */>

cfgid_blk ::= config_id config_dest author revision stamp {comment_blk} config_id ::= CONFIG LIDENT ! config_dest ::= PSC 324 ! author ::= AUTHOR LIDENT ! revision ::= REVISION <1-65535> !

stamp ::= TIMESTAMP DATE TIME ! DATE ::= <1-12>'/'<1-31>'/'<00-99> TIME ::= <0-23>':'<00-59>

cfg_end ::= ENDCONFIG !

LIDENT ::= <"string of 12 char"> FIDENT ::= <"string of 8 char"> NAME ::= <"string of 6 char"> comment_blk ::= {comment}[1.20] comment ::= ';' {CHAR}[1.59] ! NUMBER ::= <+0-9> <0-9a-fA-F.>[0.14]

uses_blk ::= USES ! {data_spec}[*] ENDUSES !

data_spec ::=

< DR_ABS {ALIAS NAME} {INIT NUMBER datatype} !

| AO_ABS {ALIAS NAME} !

| AI_ABS {ALIAS NAME} !

| CON_ABS {ALIAS NAME} {INIT NUMBER datatype} !

| GI_ABS ORG {ALIAS NAME} !

| IC_ABS {ALIAS NAME} !

| LO_ABS {ALIAS NAME} !

| LI_ABS {ALIAS NAME} !

| MSG_ABS {ALIAS NAME} {INIT "20 CHAR"} !

| ORIGIN_ABS {ALIAS NAME} !

>

/* data type definitions -----*/

DR_ABS ::= "dr<1-512>"

```
AO_ABS ::= "*ao<1-128>"
AI_ABS ::= "*ai<1-128>"
CON_ABS ::= "*co<1-110>" /* last 10 used in SLDC commands */
GI_ABS ::= "*gi<1-256>"
IC_ABS ::= "*ic<1-256>"
LO_ABS ::= "*lo<1-192>"
LI_ABS ::= "*li<1-192>"
MSG_ABS ::= "*msg<1-64>"
ORIGIN_ABS ::= "*org<1-64>.<1-256>.<1-256>"
STATION_ABS ::= "*sta<1-64>"
```

```
IC ::= NAME|IC_ABS /* internal coil */
LI ::= NAME|LI_ABS /* logic input */
LO ::= NAME|LO_ABS /* logic output */
DR ::= NAME|DR_ABS /* data register */
AI ::= NAME|AI_ABS /* analog input */
AO ::= NAME|AO_ABS /* analog output */
GI ::= NAME|GI_ABS /* global input */
CON ::= NAME|CON_ABS /* constant */
MSG ::= NAME | MSG_ABS /* string */
STA ::= NAME | STATION_ABS ORG ::= NAME | ORIGIN_ABS
```

```
/* data type SHORT; 16 bit unsigned integer */ short ::= DR|AI|AO|GI|CON
```

```
/* data type COIL; 16 bit; see PSC design spec for structure */ coil ::= IC|LO|LI
```

```
ss ::= DR|AI|AO|GI|CON /* short source */
sd ::= DR|AO /* short dest */
cs ::= IC|LI|LO /* coil source */
cd ::= IC|LO /* coil dest */
ld ::= DR /* long destination */
ls ::= DR|CON /* long source */
source ::= ss|cs dest ::= sd|cd /* note: is a proper subset of source */
variable ::= <coil|short> ALARMNO ::= *a<1-4>
```

```
/* datatypes - first 4 are general purpose, the remaining ones are for the SLDC. */
```

```
/* this did not match file "words" changed to match 9/21/86 wdw datatype ::=
```

```
< HEX | BIN | DEC | PCT | RA1 | BIA | REL | RT1 | LD1 | TI1 | TD1 | DG2 | LD2 | TD2 | TI2 |
TI3 | PG1 | PG2 | PG3 | PG4 | LNG >
```

```
*/
```

```
datatype ::=
```

< BIN | DEC | HEX | PCT | RA1 | BIA | LD1 | REL | PG1 | PG2 | PG3 | PG4 | TI1 | TI2 | TI3 | DG1
| TD1 | LD2 | RT1 | TD2 | LNG >

/* IO section-----*/

io_blk ::= IO ! {iospec}[0.*] ENDIO !

iospec ::= io_type box_no slot_no

box_no ::= <1-8>

slot_no ::= <1-8>

io_type ::= <DISCRETE OUT | DISCRETE IN | FREQ IN | BCD IN | BCD OUT | CLOCK |
WDT>

/* Recipes -----*/

recipe ::= RECIPE LIDENT comment ! init_spec[0.*] ENDRECIPE !

init_spec ::= DR INIT NUMBER datatype

/* Devices -----*/

device ::= seq_dev | lgc_dev

lgc_dev ::= lgc_desc dev_type unit_desc {comment_blk} {lil_blk} lgc_code device_end

seq_dev ::= seq_desc dev_type unit_desc {comment_blk} {lil_blk} seq_code device_end

def_type ::= TYPE NUMBER !

unit_desc ::= UNIT NUMBER !

lil_blk ::= {channel_no} {update_set}[0.*] {parameter_set}[0.*] {emerg}

channel_no ::= CHANNEL <5-242> !

update_set ::= UPDATE variable !

parameter_set ::= DESTINATION dest !

emerg ::= EMERGENCY LIDENT ! /* invalid for logic */

dev_end ::= ENDDEVICE !

/* Logic -----*/

lgc_desc ::= DEVICE NAME LOGIC !

lgc_code ::= {network}[0.128] !

network ::= network_desc {element_line}[0.*] ENDNETWORK !

NETWORK_NAME :: LIDENT

network_desc ::= NETWORK NETWORK_NAME {comment} !

element_line ::= col_num'b-'row_num'b element {comment}!

row_num'b ::= <1-8>

col_num'b ::= <1-10>

element ::=

< VS

| NOC <cs | GI>

| NCC <cs | GI>

| TRC {POS|NEG} cs

| HSHUNT hs_length

```

| INVERT

| TIMER <ONE|TENTH> ss sd {NONRETENTIVE}
| COUNTER <UP|DOWN> ss sd {NONRETENTIVE}

| TOTAL ss <BCD LOW AI DR HI AI DR>|<BINARY LOW ls HI ls> ld {BCD AO}

| ADD ss TO ss GIVING sd
| SUB ss FROM ss GIVING sd
| MUL ss BY ss GIVING ld
| DIV ls BY ss GIVING ld
| IF source math_op source DIDDLE sd

| TRANSFER REG TABLE tsize source dest ss **
| TRANSFER TABLE REG tsize source dest ss **
| TRANSFER TABLE TABLE tsize source dest ss **
| TRANSFER BLK tsize source dest

| SEARCH tsize source math_op source ss ** /* 1st source = table to search, 2nd source = search
value */

| MATRIX matrix_op tsize source dest
| MATRIX MODIFY tsize dest ss **
| MATRIX SENSE tsize source ss **
| MATRIX COMPARE tsize source source ss **

/* ** --> ss was DR, changed per RPW request 9/19/86 wdw */

| SCALE IN ss sd | SCALE OUT ss sd

| FAN IN source dest | FAN OUT source dest

| FIFO IN source que | FIFO OUT dest que

| SKIP LIDENT

| ALARM ALARMNO

| COIL cd {NONRETENTIVE}

> /* end of element list */

/* LOGIC SPECIFIC DATA STRUCTURES */
/* table and matrix no longer used */
/* table ::= source tsize dest matrix ::= source msize */

```

```
que ::= DR tsize DR DR /* HEAD tsize TAIL START */
hs_length ::= <1-8> /* MAXIMUM MUST BE CALCULATED ON THE FLY */
tsize ::= <1-256>
```

```
/* Sequence -----*/
seq_desc ::= DEVICE NAME SEQUENCE !
seq_code ::= {step}[0.256] !
step ::= step_desc {statement_line}[0.20] step_end
STEP_NAME ::= LIDENT
step_desc ::= STEP STEP_NAME {comment} !
step_end ::= ENDSTEP !
statement_line ::= statement_num statement {comment}!
statement_num ::= <1-20>
```

```
set ::= <OPEN|START|TRON> reset ::= <CLOSE|STOP|TROFF>
```

```
statement ::=
< set {cd}[1.12]
| reset {cd}[1.12]
| ALARM <set|reset> {NAK} ALARMNO
| MOVE <<cs TO cd> | <ss TO sd>>
| DELAY sd UNTIL ss
| NEXT
| GO_TO LIDENT
| MANUAL MODE
| RELEASE
| PARALLEL LIDENT
| ENDPARALLEL |
EMERGENCY LIDENT
```

```
/* Multistatement constructs may cross step boundaries */ /* they count for two statements, & if-else-endif is 3 statements */
```

```
| IF source op source THEN {comment} ! block { ELSE {comment} ! block } ENDIF
| IF NOT source THEN {comment} ! block { ELSE {comment} ! block } ENDIF
| IF source MEQ source MASK ss THEN {comment} ! block { ELSE {comment} ! block } ENDIF
```

```
| WHILE source op source DO {comment} ! block ENDWHILE
| WHILE NOT source DO {comment} ! block ENDWHILE
| WHILE source MEQ source MASK ss DO {comment} ! block ENDWHILE
```

```
| REPEAT {comment} ! block UNTIL source op source
| REPEAT {comment} ! block UNTIL NOT source
| REPEAT {comment} ! block UNTIL source MEQ source MASK ss
```

```
| COUNTER sd UNTIL ss {comment} ! block ENDCOUNTER
```

| TIMER sd UNTIL ss {comment} ! block ENDTIMER

/* LINK commands, generic format */

| SEND privilege link_op source TO ORG

| MESSAGE MSG TO ORG

| GET_EXT ORG dest {MSG TO ORG} {BETWEEN ss AND ss UNTIL ss}

/* see below */

| SEND_COM STA station_type dev command {command_data}

| END > /* end statement list */

statements ::= {statement_line}[*]

steps ::= {step}[*]

block ::= statements { step_end steps step_desc statements }

privilege ::= <COMPUTER | ANY> /* ANY source for source change commands */

/* Miscellany----- */

math_op ::= GT | LT | GE | LE | EQ | NE

lgc_op ::= AND | OR | XOR

matrix_op ::= lgc_op | COMPLEMENT

op ::= math_op | lgc_op

char ::= <_a-zA-Z>

link_op ::= ABS | set | reset | RELATIVE | RANGE12 | RANGE14

/* 32 bit int determined on the fly from the sources data type */

/* Device Oriented Command Structures currently defined ----- */

/* for SLDC Command syntax */

station_type ::= SLDC /* | SLC | PSC (maybe later) */

dev ::= <STATION | LOOP1 | LOOP2 | CHAN_A | CHAN_B | CHAN_C | CHAN_D>

command ::= < COMMAND | AUTO | MANUAL | COMPUTER >

COMMAND ::=

< HOLD

| RUN

| CLEAR_ERR

| FLASHER_ACK

| LOCAL

| C_COMPUTER

| C_AUTO

| C_MANUAL

| CONSOLE

| INTERNAL
| EXTERNAL
| ALARM_OPERATIONAL
| ALARM_OUT_OF_SVC
| RATIO
| BIAS
| LEAD
| LAG
| OUT_70
| OUT_71
| OUT_72
| OUT_73
| OUT_74
| OUT_75
| OUT_76
| OUT_77
| RAMP_ON
| RAMP_OFF
| PROP_GAIN
| INT_TIME
| DERV_TIME
| DERV_GAIN
| MAN_RESET
| SETPT_ABS
| SETPT_REL
| TARGET
| HI_LIMIT
| LO_LIMIT
| VALVE_ABS
| VALVE_REL
| RAMP_TIME
| ALARM_EN
| ALARM_DIS
| ALARM_ACK >

/** these alarm commands will not be implemented - see below **/

<
| ALARM_LIMIT
| ALARM_NONE
| ALARM_HI
| ALARM_LO
| ALARM_HI_DEV
| ALARM_LO_DEV
| ALARM_ABS_DEV
| ALARM_OUT_OF_RANGE

```
| ALARM_DEADBAND  
| ALARM_DELAY_IN  
| ALARM_DELAY_OUT  
| ALARM_RINGBACK >
```

```
command_data ::= < ss | ALARMNO >
```

```
/* SLDC (#352) Commands */  
IF device == sldc_station.STATION THEN  
  < command ::=  
    < HOLD  
    | RUN  
    | CLEAR_ERR  
    | FLASHER_ACK  
    | LOCAL  
    | C_COMPUTER  
    | CONSOLE  
    | C_AUTO  
    | C_MANUAL  
    | INTERNAL  
    | EXTERNAL  
    | ALARM_OPERATIONAL  
    | ALARM_OUT_OF_SVC  
    | RATIO ss  
    | BIAS ss  
    | LEAD ss  
    | LAG ss  
    | OUT_70 ss  
    | OUT_71 ss  
    | OUT_72 ss  
    | OUT_73 ss  
    | OUT_74 ss  
    | OUT_75 ss  
    | OUT_76 ss  
    | OUT_77 ss  
  >  
>
```

```
IF device == sldc_station.<LOOP1|LOOP2> < THEN command ::=  
< RAMP_ON  
| RAMP_OFF  
| PROP_GAIN ss  
| INT_TIME ss  
| DERV_TIME ss  
| DERV_GAIN ss  
| MANUAL_RESET ss
```

```
| SETPT_ABS ss
| SETPT_REL ss
| TARGET ss
| RAMP_TIME ss
| HI_LIMIT ss
| LO_LIMIT ss
| VALVE ss /* INVALID for LOOP 2 !!! */ >
>
```

```
IF device == sldc_station.<LOOP1|OUT1|OUT2|OUT3|OUT4> THEN
  < command ::=
    < ALARM_EN sldc_alarmno
    | ALARM_DIS sldc_alarmno
    | ALARM_ACK sldc_alarmno
  >
>
```

```
IF device == sldc_station.LOOP1
THEN <sldc_alarmno ::= <1-4>>
ELSE <sldc_alarmno ::= <1-2>>
```