

Reliable application software for digital
controllers using a "Syntax Directed Editor"

Wayne D. Woodruff	Max R. Yaffe
Project Engineer	Project Engineer
Moore Products, Inc.	Myriad Software Development
Spring House, PA 19477	Maple Glen, PA 19002

A. Abstract

Programmable controllers used in the process industries require application software to define the specific control strategy. This application software may take the form of function blocks, ladder diagrams, or a high-level batch language and it is often generated by an end user or contractor. This software can be the source of faults which can disable the controller, just as serious as hardware failures. These types of faults are especially critical in process control and particularly in safety interlocking applications.

In a new control device called the "Programmable Sequence Controller", (PSC), a process computer language was developed to allow flexible use of the controllers resources. With any programming language or configuration means, errors tend to reduce the reliability of the product. To overcome these problems, a "syntax directed editor" was developed. This editor prevents errors by prompting the user for all required information, analyzing the users entries for errors, and formatting them into the required syntax, thus eliminating the possibilities of typing errors and missing information.

B. Keywords:

1. Programmable Controller
2. Language & Syntax
3. User-Friendly
4. Configuration

C. Introduction

The Programmable Sequence Controller (PSC) offers both ladder logic programmability and a sequentially executed language for batch control. The PSC can also communicate via a Local Instrument Link to form a distributed batch control network.

The typical target end user is trying to control some small to medium scale batch industrial process requiring a sequence of steps and ladder logic based interlocks. The user may also require alarm functions and ability to do distributed loop control. Since the PSC is a programmable device it requires configuration expertise. The end user may use in-house staff, consultants, or vendor engineers to provide configurations. The configuration software package is designed to run on an IBM-PC or compatible computer. It is also designed for a user who may be using it infrequently. This means the configuration system must be easy to use and not force the user to rely on manuals.

At the same time, the user expects to create a reliable control configuration. Reliability in this case means a configuration, which is functionally correct, when it is created and also robust to both anticipated and unexpected events when operating. The types of faults associated with controller hardware have been continually diminishing with time. However as controller become more complex and configurable, more of the burden of correct operation is being placed on the configurer. Therefore, any effort we take to make the configuration process simpler and less error prone will improve reliability.

D. Configuring the PSC

A PSC configuration is contained in a file on a PC-DOS disk. When the configuration program is run this file is loaded into memory. The configuration process consists of editing the configuration, compiling the configuration into a PSC loadable format, and the loading the configuration into a PSC via a Local Instrument Link. Also included in the configuration program is a documentation and cross-referencing section.

The configuration environment is completely menu driven which is key to making it easy to use. A status line at the top of the screen lets the user know exactly where he is in the program. (Fig. 1). The status line at the top & menu sections at the bottom are written on a contrasting background from the main screen. These colors are consistent throughout the program. As different information appears on the main screen, the menus change. Keystrokes are also consistent throughout the environment. Most important is the use of a semi-graphical cursor controls, i.e. even though we are using a text display mode for the PC, users chose the item to be edited via cursor control keys.

We chose a fairly simple menu presentation using only the unshifted function keys. Also a small set of Alt-Alphabetic keys were used for similar functions throughout the environment, e.g. Alt-H for Help, Alt-E for Edit, Alt-S for Save. There are as many menu systems as there are programmers and most would work as well.

Another key to creating a straightforward environment was our use of the Dialog Box. This software metaphor creates a fill-in-the-blank environment with multiple field entry points. (Fig. 2A) Unlike most such environments, the user is still allowed to roam around the fields correcting and modifying individual entries almost at will. One nice feature is the use of individual menus for each field. Also, we implemented several different types of fields ranging from menu-selectable to any-text-goes. Fig. 4 shows a dialog box for configuring a timer.

When information is put into a field the results are checked with respect to the contents of other fields and the state of the configuration. Another set of checks is performed on the field data when the dialog box is "accepted", i.e. when the user is done with the dialog box. At either of these checks, the system may accept the data, open another dialog box for more details (Fig. 2B), or reject it with an error message (Fig. 2C).

Detailed prompting and error checking enhances the reliability of the configuration because invalid data is not accepted. Errors are easier to eliminate when the error checking is done when the information is being entered, not later when the context has been forgotten.

With these features we have created a configuration environment which will get the job done. But to prove that we haven't just written another pretty program, let's study it as a paradigm of a syntax directed editor.

E. The hidden language - from practice to theory.

High level languages such as Pascal and C, are defined using a specification called a grammar. A major portion of the grammar describes how predefined words, known as keywords, and user defined words, such as variables, may be chained together to form meaningful program statements. This section of the grammar is known as the "syntax".

There is, indeed, a language that underpins the PSC configuration program. This can be shown by a section of the configuration file generated by the timer example. The example was actually created as a part of the configuration process and can be loaded into the configuration program to be edited.

```
1  config Demo
2  ...
3  uses
4   *dr002 alias Tau18 init 0 dec
5   *co100 alias Tau17 init 0f80 hex
6  enduses
7   device LOG1 logic
8   ...
9   network network3 ;Test Network
10   4-3 timer one Tau17 Tau18
11  endnetwork
12  enddevice
13  endconfig
```

Line 10 is the final result of the timer dialog. Lines 4 & 5 came from the variable initialization dialogs.

The example above shows something more similar to a computer language. Let's dissect a simpler example, the Normally Open contact (NOC).

a) The picture - a Normally Open Contact

The NOC is a fundamental component of ladder-logic programming. A sample is shown in Fig. 3. A contact must have a row & column location to specify the interconnections and it must be closed by some virtual coil.

b) The language statement.

The PSC language statement representing the NOC is written as:

```
1-1 noc *Io002
```

The ladder logic statement indicates that at column 1 rung 1, there is a normally open contact referenced to local output channel number 2.

c) Choices represented in the language statement

There is a syntax rule delimiting this statement:

```
NOC_line ::= col-row NOC coil {comment}
```

On the right, the terms "col-row" limit the range of available rows and columns. The next term defines what function is to take place, the NOC. The term "coil" says that any readable coil may control this particular NOC. Finally there is an optional comment bringing up the rear. Those terms written in small letters are further defined later in later rules. For example the col is defined by:

```
col ::= <1..10>
```

All of these rules taken together form a syntax for the PSC configuration language. The syntax for the PSC configuration is some 400 lines long and is described in reference 1.

F. PSC configuration as a Syntax Directed Editor

Now that we have established the relationship of the PSC configuration to a syntactically defined computer language we can proceed to develop the concept of a syntax directed editor (SDE). Syntax directed editors use the language syntax to guide the programmer to correctly define a program. To do this, the editor usually presents the programmer with a menu of the programming statements.

Some SDE's for high level programming languages generate a skeleton statement and highlight the fields that need further definition. When the programmer has successfully entered a highlighted field, the highlighting disappears. When all highlighted fields are gone, the syntax conforms to the rules of the programming language. Errors are flagged at the field of entry.

The dialog box method described above gives another method of filling in a statement according to the prescribed syntax. Using this technique, when the programmer has successfully entered all required fields and closes the dialog box, all of the entries are checked for validity and the editor formats the programmer's entries to create a syntactically correct statement. Errors checking is performed on a full line of code, as well as on a field by field basis.

Let's return to the timer example to show how its dialog box and its statement syntax are related. Refer to figure 4A for the timer dialog box. Figure 4B shows the completed timer as a part of a ladder-logic network. (Remember that the casual user will only see these two facades and will not be aware of the underlying language.) The dialog box in the figure generated the internal source statement:

```
4-3 timer one Tau17 Tau18
```

The syntax of the timer statement is:

```
TIMER_line ::= col-row TIMER timebase ss sd {NONRETENT}{comment}
```

Each term in the TIMER_line specification starting with "base", the timebase, corresponds to a field in the dialog box. The col-row term has been implicitly chosen by the cursor position while the TIMER term has been implicitly chosen by asking for a timer at this point in the first place. The timebase term is defined by:

```
timebase ::= <"one"|"tenth">
```

which means to choose either a 1 second or a 0.1 second timebase. Figure 4A shows the dialog box and menu while the timebase is the selected field. As we saw previously, if more detail is required to define a term, a derivative dialog can be started while the cursor is in the field corresponding to that term.

The dialog box is not the only method we use to translate syntax to editor program flow. Another important method is used at a higher level to control the program flow. The PSC is designed to look like a cluster of smaller controllers known as devices. There are two classes of PSC devices, sequence devices and ladder logic devices. The ladder logic devices are configured like conventional programmable controllers simulating relay logic circuits. A single PSC ladder logic device contains an array of up to 128 networks. The formal syntax for a list of networks is:

```
logic_code ::= {network}[0.128]
```

The program representation for this list is shown in fig 5. The program allows one to move up and down on this list, to insert new networks, to delete networks, and to edit them. A network is edited by hitting Alt-E while that network is highlighted. But even this list manager is strictly defined by the syntax statement above -- one must edit a network from this point only.

G. What difference does having a syntax make?

We hopefully have convinced the reader that the visually oriented PSC configuration program is really a syntax directed editor for the PSC language. We've also seen how a syntax directed editor minimizes aggravation and reduces mistakes by detecting errors quickly. Now we want to relate some of the less obvious benefits.

Systems programming languages such as C or assembler are usually very generalized to permit maximum speed and flexibility. Because of this generalization, the reliability or safety of the language can be marginal. Applications languages on the other hand usually have a more restricted syntax optimized for the application at hand. Some languages such as LOGO or Pascal are designed for the novice or infrequent user and have built in checking for invalid operations. In all of these cases there is a definite syntax associated with each computer language so that having a syntax by itself does not improve reliability. What the syntax does afford the language designer is a definite point in the language design to specify constraints.

For example in the PSC language we decided a statement like

1-1 NOC register

would be illegal since a contact must reference a coil variable and not a 16 bit arithmetic register. We could have just as easily decided to make the above statement legal and automatically assigned some ON/OFF state to any possible register value. In that case the NOC syntax would look something like:

```
NOC_line ::= col-row NOC <coil|register> {comment}
```

Alternately we could have made a statement like:

1-1 NOC tocoil(register, bit_number)

where the transition from register to coil was explicitly specified.

The main point is that the syntax forces the language designer to make those types of choices very early in the game. It also raises design questions that might not have been perceived earlier. The controller designer will be sensitive to the ability to define more foolproof languages by eliminating those choices, which don't make physical sense.

The limitation of choices issue is more critical than we first imagined. When the PSC's syntax was first developed from the functional specification, the document was incredibly complicated. As we studied the rules we saw many places where functionality could be regularized or even eliminated with no loss of system capability. These changes resulted in the coalescing of rules in the syntax. Later on when the editor was created, we realized that each rule in the syntax must be represented by a menu choice in the editor. Again, the complexity of the syntax inherited by the user interface. By minimizing those rules the user interface became simpler and therefore allowed the user to create more reliable configurations.

We stumbled on one interesting observation. There are two schools of thought about how to design configuration programs. One school espouses menu-driven programs where the users are given detailed menus and prompts for information. The second school contends that the program should get out of the way of the user allowing him to directly enter information in the form of mnemonic commands. Both schools contend that they create "user-friendly" programs. We found that neither method really correlated with friendliness. Instead we found that the fewer the number of rules there are in the configuration syntax, the friendlier the program appears. To the first design school each rule corresponds to a menu choice or checkpoint -- the fewer the rules, the simpler the menu structure becomes. To the second design school each rule corresponds to another command or restriction on a command that must be mastered -- the fewer the rules, the simple the command structure becomes.

A third benefit of this technique is the elimination of the constraints of the database method of configuration. In a database driven design fixed field-lengths and fixed record limits are generally

the rule. Because of this, a lot of space is wasted. However, a source configuration for the PSC contains few arbitrary limits and can be more compact. Some space is wasted in a source configuration by using words where smaller symbols would do but, in general, space is well utilized. We attribute this to our continually thinking of writing a configuration in a text oriented language rather than filling in the blanks in a database.

A final benefit of the "Syntax Directed" design method is the existence of a remarkably compact detailed design document, the syntax document. It is a mathematical statement of what the machine can and cannot do. With a little effort the technique of interpreting this document can be learned by everyone involved with the design, not just programmers. Since the document is concise a historical record of design modifications is easily maintained.

H. Conclusions

a) User friendly correlates with simple syntax.

Yaffe's Hypothesis - Conservation of confusion. The more complex the syntax, the more difficult the system will be to configure. Therefore to make a system simpler -- make the syntax simpler.

b) Complexity inverse correlate of reliability.

Woodruff's Corollary - Configuration complexity inversely correlates with reliability. Simpler configuration rules generate easier to understand configurations thereby increasing reliability.

c) Concise syntax yields reliable programs.

The PSC Proposition - A syntax directed editor derived from a concise formal syntax will provide an environment for generating more reliable controller configurations.

The techniques we have developed are straight forward but do take some effort to learn. However, in this project, the effort seems to have paid off with a well received final product. Syntax derived design and syntax driven editors are a welcome addition to programmable controller technology.

I. References

- 1) Woodruff, W., "A Formal Language for Process Control Computers", Villanova University, Electrical Engineering Dept. 1986.
- 2) Schreiner, A.T, Friedman, H.G., "Introduction to Compiler Construction with Unix", Prentice-Hall, Englewood Cliffs, NJ, 1985 . 3) Zarrella, J, "Language Translators", Microcomputer Applications, Suisun City, Cal., 1982.
- 4) Heckel, P., "The Elements of Friendly Software Design", Warner Books, New York, NY, 1982.

J. Figures

```
PSC/New/dev1/step1 Editor
Step Descr.:
  Stmt.Descr.:
1 tron *lo001 *lo002
2 alarm tron *a1
3 message *msg01 to *org01.001.001
4 if *dr001 eq *co001 then
5 go_to next_page
6 endif
7
8
9
10
11
12
13
14
15
16
17
F1 Goto      F4 Release  F10 Next Menu
F2 Next      F5 Mode     Alt-H Help
F3 Emergency F6 SLDC Command ESC Exit
```

Figure 1 Screen Layout

Dialog Box Editing

```
===TIMER===
Preset: *dr001
Accumulator: *dr001
Descr:
```

Figure 2A

```
===TIMER===
Pre
Acc ===Tau17===
Des Absolute Reference: *ic001
```

Figure 2B

```
===T Warning: "*ic001" is not valid here.
Pre
Accu
Descr:
```

Figure 2C

*li001

*lo001

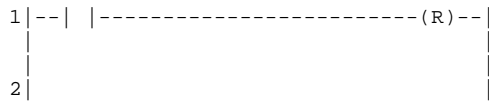


Figure 3A User's view of a Ladder

```

1-1 noc *li001
2-1 hs 2
4-1 coil *lo001 retentive

```

Figure 3B Resulting Program Statements

```

PSC/New/dev1/network1 Editor
Netw.Descr.:
Elem.Descr.:
  1  2  3  4  5  6  7  8  9  10
1  [ ]
2
3  ===TIMER===
4  Preset: *dr001
5  Accumulator: *dr001
6  Time Base: one
7  Mode: retentive
8  Descr:
9
10
F1 1.0 Sec
F2 0.1 Sec
Alt-U Undo
Alt-H Help
ESC Exit

```

Figure 4A Timer Dialog with Menu

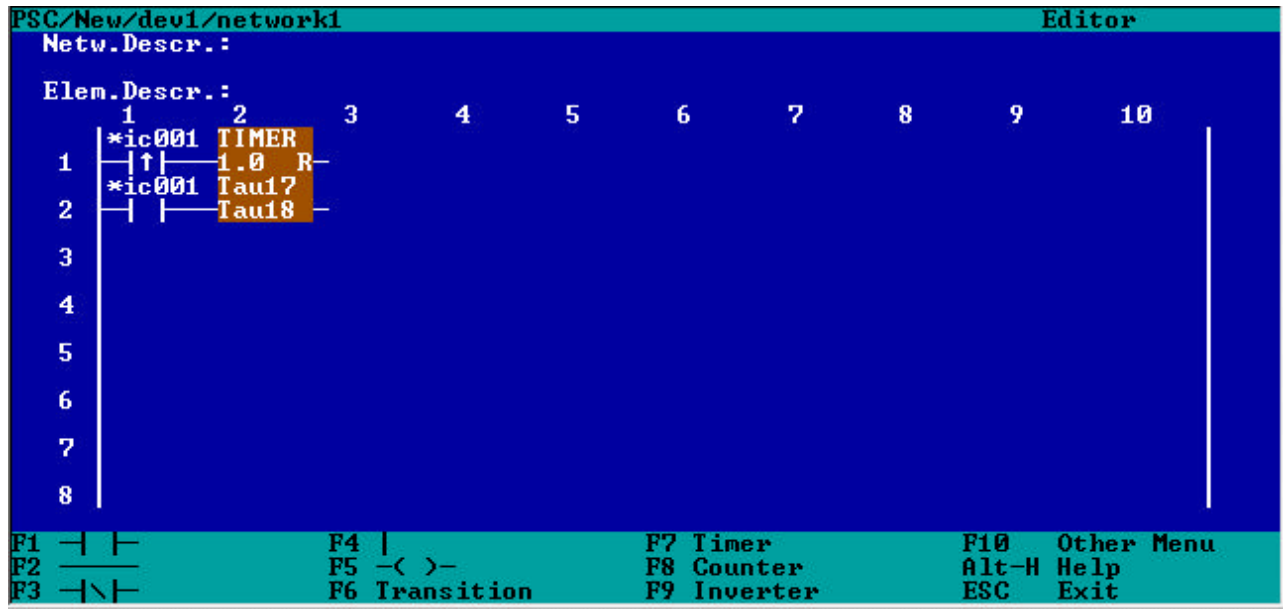


Figure 4B Timer Picture

4-3 timer one Tau17 Tau18 retentive

Figure 4C Timer Statement

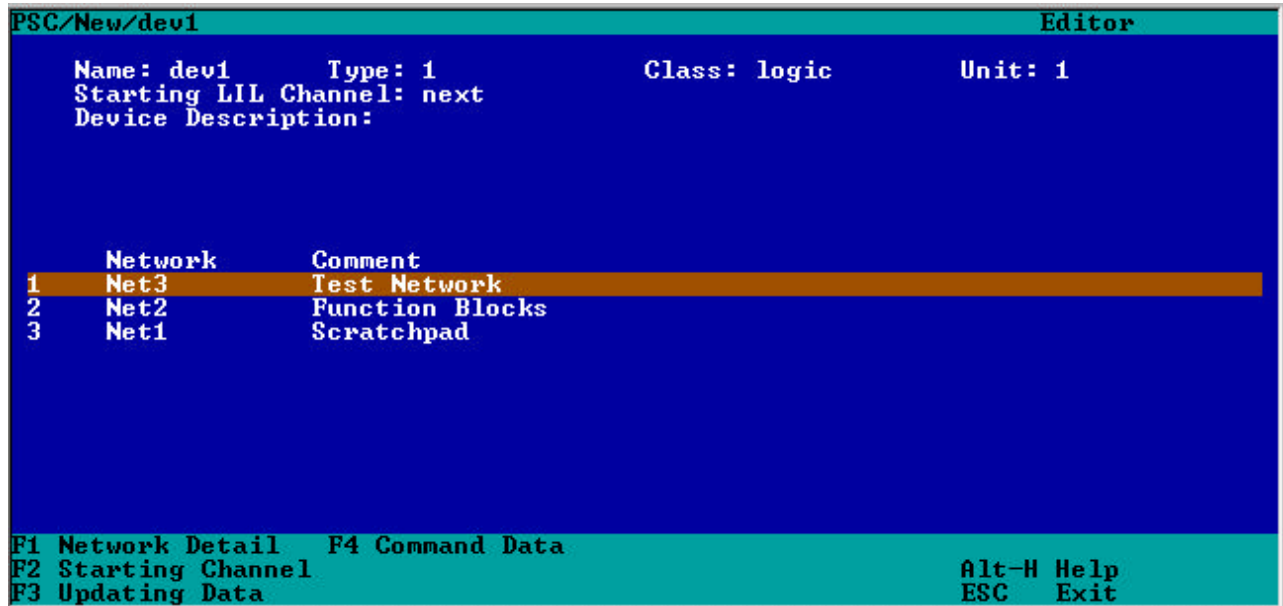


Figure 5A User's view of a list

Network Net1 ;Scratchpad

....

Endnetwork

```
network Net2 ;Functions Blocks
....
Endnetwork
network Net3 ;Test Network
....
endnetwork
```

Figure 5B Program Statements