HIGH LEVEL PLC LANGUAGE AND DEVELOPMENT SYSTEM

A. SZEGI

Department of Measurement and Instrument Engineering, Technical University, H-1521 Budapest

Received March 5, 1987

Abstract

The "PIV" high level PLC language developed in the Department of Measurement Engineering, Technical University of Budapest, ensures bit, byte and word processing, the possibility of symbolic I/O and internal variable names, single line expressions, IF-THEN-ELSEIF-ELSE conditional structures and state transition graph structures. The development system makes the line-by-line correction possible and ensures a PIV language level debugger with built in LSS (logic state store).

Introduction

The MMT micropocessor application system has been developed at our Department cooperating with MEDICOR Works. Originally it was proposed for building laboratory-measurement equipment. Several Hungarian firms have bought the license of the system and most of them are interested in industrial applications, too. Therefore the idea of applying the MMT system in industrial environment arose. To accomplish this; additional HW and SW elements are necessary.

In this paper a high level language and development system for industrial application is dealt with. The language is dedicated for medium-high complexity systems with mostly two state I/O signals, but restricted complexity analog I/O handling is also possible. The main characteristics of the language, the requirements of an interactive program development system and the debugging possibilities built into the system are described in the paper.

PLC programming

To solve not very high speed industrial control problems, PLC-s (Programmable Logic Controllers) are widely used. Their operation may be sequential, when a step-by-step program is executed depending on input conditions, or cyclic, when the program is executed cyclically, or the combination of the above modes. For programming PLC-s different specialized languages have been developed:

— (1) circuit diagram equivalent description. Its typical and most widely used form is the 'ladder diagram' when a relay network or its equivalent solving the problem has to be given [1]. This program can be realized by cyclic operation.

— (2) step control description [2, 3]. It gives the conditions of state transitions. Usually it is used together with (1) which describes the operation in a given state.

— (3) "IF condition THEN operation" type description [4]. Both cyclic and sequential operations are solved in the given example.

— (4) high level languages. Either a high level PLC language based on a high level language is defined (e.g. CONDOR based on ALGOL [5]) or a high level language cooperates with a PLC language (e.g. BASIC and ladder diagram in [6]).

Most of the PLC languages are of low level: symbolic names, single line expressions, labels are not allowed. It can be explained by three reasons: the development device must not be expensive, the programs are relatively simple and it can be useful if the source can be regenerated from the running code. In our opinion, the technological breakthrough in electronics makes it possible to reevaluate the above points of view and a high level PLC development system (language + development device) can be of vital importance.

The high level PLC language

The system operates basically in cycles. The cyclic operation means the repetition of the input sample-internal data processing-output set sequence (see Fig. 1). The adequate program structure is shown in Fig. 2.

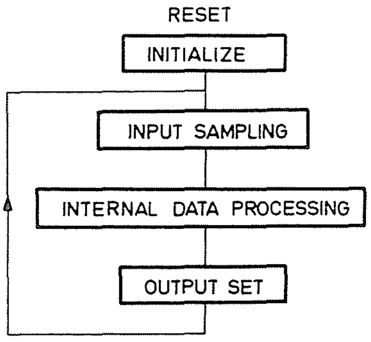

Fig. 1. PLC cyclic operation

```
PROGRAM
                data, procedure, graph
                declarations
INIT
                initialization program
CYCLE
                cyclic program
END
```

Fig. 2. PLC program structure

Data types

A data unit can be a bit (of a byte), a byte or a word. From the point of view of usage it can be INPUT, OUTPUT, (internal)DATA, TIMER. An example for the declaration of different data types and some of their bits are shown in Fig. 3. E.g. the INPUT declaration gives the symbolic name "door" to the input byte of address 12, the symbolic name "closed" to its bit 7 etc.

```
INPUT     12, door (7: closed, 6: opened, 5: toclose, 4: toopen)
OUTPUT    7. dooroperation (7: close, 6: open)
DATA      error (7: closeerror, 6: openerror)
TIMER     doortime, 100ms
```

Fig. 3. Data declarations

Complex bit expressions

In conditional statements or in bit assignments a bit expression formed of input, output and internal bitnames, timer names and bitfunction names using * (AND), + (OR) and \ (negation) operators and brackets of any complexity can be given. E.g. see Figure 4.

```
IF          close*\closed*\doortime THEN
            closeerror = TRUE
ENDIF
```

Fig. 4. Conditional statement

State transition graph

A door operation state transition graph (Fig. 5) and a part of its program representation (Fig. 6) is given as an example.

The possible graph handling operations:

NEXT state	; state set within the graph
graphname = statename	; graph state set outside the graph
(graphname = statename)	; graph state test in a bit expression
graphname	; activates the graph

(The activated graph takes one step which means that the program in the actual state is executed until it arrives to a NEXT or an ENDSTATE statement. In the program several graphs and out-of-graph program parts can cooperate or run independently in parallel.)

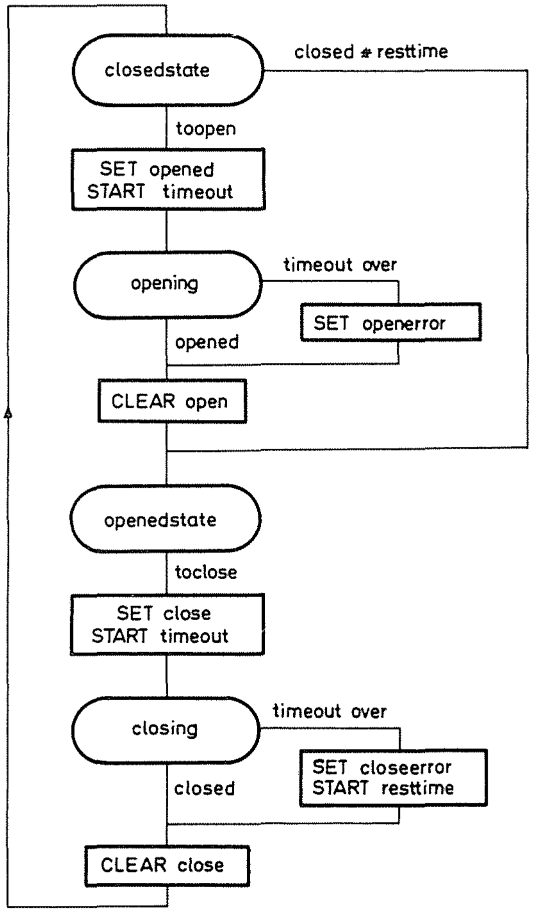

Fig. 5. Door movement control graph

```
GRAPH     doormove
BEGIN
STATE     closedstate
          IF        toopen
                    doortime=timeout
                    NEXT      opening
          ELSEIF    \closed*\doortime
                    GOTO      startclose
          ENDIF
ENDSTATE

STATE     opening
          IF        opened
                    GOTO      11
          ELSEIF    \doortime
                    SET openerror
11:
                    CLEAR     open
                    NEXT      openedstate
          ENDIF
ENDSTATE

END
```

Fig. 6. Door movement control graph program detail

Development system

Its main requirements are:

— interactive high speed correction
— in site correction → portability
— real time debugging

A PLC development device of higher performance is usually a portable CRT terminal like equipment with built in floppy disk, Eprom programmer and eventually u.v. eraser.

The best known method for high speed program correction is the application of interpretation. It is either very slow or requires some preprocessing (e.g. expression transformation into a sequence of operations, label evaluation). We worked out a method for running compiled machine code which allows the correction of single lines without redundancy in the correction free program. The principle of the method is the following:

When the compiler translates the program the beginning address of the binary code of each line is stored in an address table. When a source line has to be corrected it can be done by a specialized editor which knows its serial number. Then the binary correction is realized by patching the new binary code to the end of the program. The basic scheme is shown in Fig. 7. Assume that the program is 3 lines long and the second line has to be corrected.

The shown method can be refined in some details:
— If the new code is not longer than the old one then it can be replaced (e.g. if only a value or a variable name was changed);
— the jump chains resulting from multiple correction can be eliminated.
By this method line delete and insert can also be solved. Not all the line types can be corrected by this method. In general, declarations or definitions cannot

Before correction		After correction	
Line Address Table	Binary Program	Line Address Table	Binary Program
A1	A1: line1	A1	A1: line1
A2	A2: line2	A2	A2: JUMP A2'
A3	A3: line3	A3	A3: line3
			A2': new line2. JUMP A3

Fig. 7. Scheme of line correction

be deleted or modified. E.g. if a label were erased then the jumps related to it would become undefined (the GOTO statement can be deleted!). In this case a full recompilation is necessary.

The source code is memory resident, the corrections in it and in the address table are realized by direct erasure/insertion.

Debugging

The development terminal is connected to the PLC under development through a high speed serial link. The PLC receiver can cause an NMI in the normal run. In the NMI state the variables or graphs can be examined or changed in the usual way. An interesting possibility is the LSS (logic state store). The storage is solved automatically by the PLC by the method shown in Fig. 8. Storing takes place in every program cycle shown in Fig. 1.

The input-internal-output data field is multiplied and forms an LSS. At the end of each cycle the internal data (D) and the output data (O) are copied into the next region, and the input data are sampled into this region. So the previous region reserves the input data and the results of the previous cycle. Of course this is repeated cyclically, i.e. after the last region the first one is the following. Breaking the run the data of the last *n* cycles can be examined.

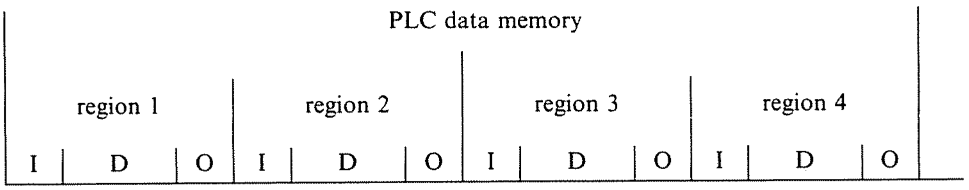

Fig. 8. Structure of the PLC data memory

References

1. H. BERGER: Programming of Control systems in STEP5, Volume 1 Basic software, Siemens Aktiengesellschaft, 1980.
2. M. LLOYD: Graphical Function Chart Programming for Programmable Controllers, Control Engineering, October 1985.
3. P. BRICH–S. GERBER–H. P. OTTO: Programming of Sequence Controls with the SIMATIC S5 Programmable Controller System, Siemens Power Eengineeing, 984 No. 3.
4. Festo FPC 606 User's Manual; Introduction to Programming, Festo Electronics.
5. H. TAKAHASHI: An Automatic-Controller Description Language, IEEE Trans. of SW Engineering, Jan. 1980.
6. K. MATSUZAKI–S. HATA–O. OHKOCHI–M. OKAMURA–N. SUGIMOTO: Programmable Controller with a Multiprocessor-based High Speed Interactive Language System, CH1987-8/83/0000–0180, 1983, IEEE.

Dr. András SZEGI H-1521 Budapest