Programming PLCs using Structured Text

Nieke Roos
Department of Computing Science
University of Nijmegen
Toernooiveld 1, NL-6525 ED, Nijmegen, The Netherlands
nieker@sci.kun.nl

Keywords: Programmable Logic Controller, PLC, Structured Text, ST, Pascal

Abstract

An ever increasing part of the industrial and safety-critical processes is presently controlled by Programmable Logic Controllers (PLCs). As its name already suggests, a PLC can be and has to be programmed to perform its task. Since its birth, a wide range of different PLCs has been put on the market that makes use of an even wider range of programming languages. Recently, the IEC 1131-3 standard has been published in an effort to reduce chaos to order in the universe of PLC programming languages.

This paper takes a closer look at one of the programming languages defined by the IEC 1131-3 standard, namely the Structured Text programming language. A comparison is made between Structured Text and the language upon which it was based, the Pascal programming language. The primary differences that result from this comparison are then discussed and some conclusions are drawn about the IEC 1131-3 standard.

1 The Programmable Logic Controller

1.1 Introduction

As the Industrial Revolution stormed the civilized world late last century man started to create machines that could alleviate the hard labour some existing tasks brought with them. This very first large-scale reflection of man's desire to be able to automate was the incentive to an evolutionary developement of industrially employed equipment. The first machines ran on steam, later on gasoline was used as primary source of energy and at present the majority of the devices in the industry is powered by electricity. At the same time the dimensions of these devices decreased from steam engines the size of an average suburban dwelling to electrical appliances not bigger than a lunchbox containing even smaller circuits and wiring.

The invention of the computer and its massive application following the Digital Revolution taking place in the 1960s, gave birth to a new phenomenon: the art of programming. This revolutionary ability to control the behaviour of an apparatus by precisely conveying the actions it has to perform exerted enormous influences not only on the computer science domain but on other domains as well. An important and interesting example of the influence programmability has had on other domains is the Programmable Logic Controller (PLC), resulting from the synthesis of the electrical world and the universe of computer science in an effort to combine this programmability with electrical circuitry.

1.2 The architecture

The primary purpose of a PLC is to control a particular process or collection of processes in industrial or other safety-critical applications. This system of processes has certain properties, which are to be controlled by the PLC. These properties are communicated to the PLC through digitized analogue signals directly or indirectly emanating from the system. In response to these inputs the PLC produces electrical control signals that are fed back to the system. The main task of a PLC, therefore, is to convert the inputs received to particular related outputs.

1.2.1 The hardware

In essence, the PLC is a miniaturized version of a general-purpose computer. Like its big brother, its structure can be divided into a hardware-part and a software-part. As depicted in Figure 1, the PLC's

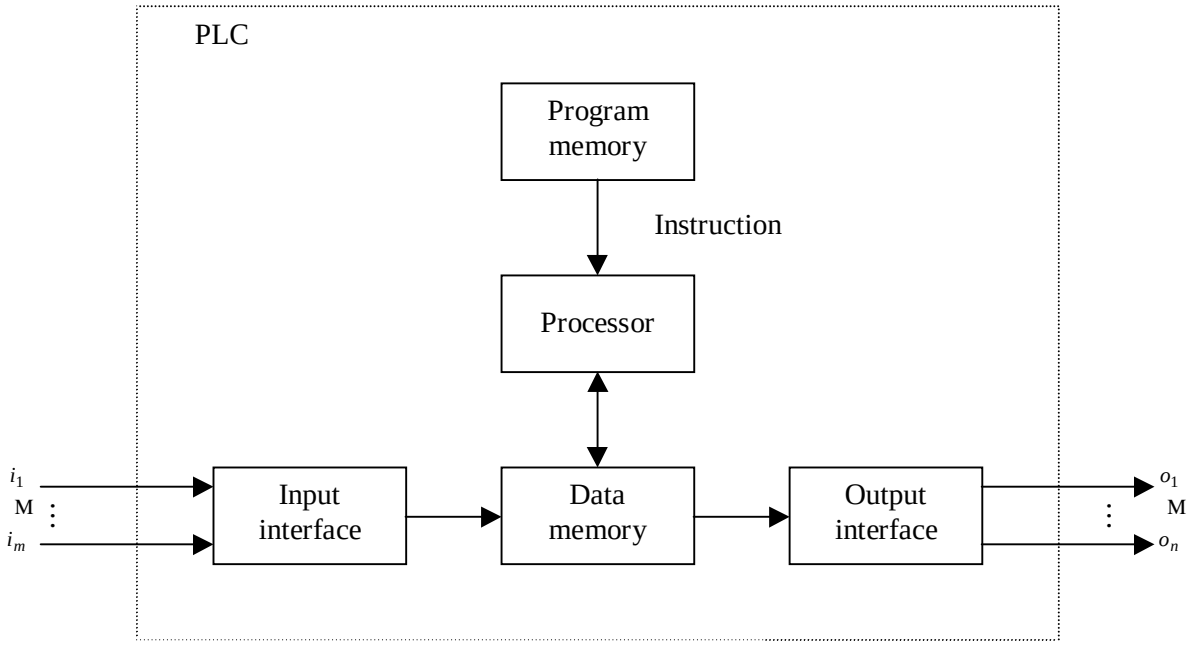

Figure 1: PLC hardware architecture

hardware is composed of:

- an input interface,
- an output interface,
- a program memory,
- a data memory, and
- a processor.

The PLC's task commences as the input signals from the system of processes that has to be controlled arrive at the input interface. These inputs are then stored in the PLC's data memory. Next, the PLC's processor starts fetching and executing instructions from the PLC's software contained in the PLC's program memory. As the processor is doing this, the data memory serves as storage of intermediate and/or final results produced by the execution of a particular program instruction. After the processor has executed all the instructions in the program memory, the output interface is responsible for communicating the final result or results back to the system of processes that is under the PLC's control. Finally, the PLC's task starts all over again as new input signals arrive at the input interface. Hence, the PLC can be regarded as executing its task indefinitely.

Note that the before-mentioned task-loop usually is said to be executed not by the hardware but by something that is a combination of hardware and software, called firmware. Firmware is software that has been written onto read-only memory.

1.2.2 The software

When one takes a closer look at this task-loop, one can observe that it basically is the responsibility of the PLC's software contained in the program memory to convert the inputs received at the input interface to related outputs emanating from the output interface. Like every software program, the program that resides in the PLC's program memory and that performs this conversion has to be written in a particular programming language. However, which programming language should a PLC manufacturer choose to program his/her products?

Until recently, this posed a huge problem. Because there was no such thing as a uniform language for programming PLC's, every manufacturer developed and applied a programming language of his/her own. This caused a virtually endless series of different programming dialects to be created. For a company new to the field to pick one to use in its designs was the same as looking for a needle in a haystack. In 1993, however, the International Electrotechnical Commission (IEC) decided to facilitate the needle finding by publishing the IEC 1131 International Standard for Programmable Controllers. The third part of this standard, entitled "1993 Programmable languages" but commonly called IEC 1131-3, contains the definition of five languages for programming PLCs ([IEC]).

2 IEC 1131-3

2.1 The software model

Part three of the IEC 1131 standard defines an architecture for the PLC software called the software model (see Figure 2). At the highest level, this software model describes the notion of a configuration. In fact, this notion of configuration is just a fancy way to refer to the entire PLC software. On a lower level, a configuration can contain one or more resources. A resource can be seen as providing a processing environment for executing the program contained within the resource. A program is composed of a collection of definitions of software elements called functions and function blocks and their application within a sequence of statements. Functions and function blocks are the basic building blocks of a program, able to exchange data. A function is a particular algorithm, i.e. sequence of statements, with zero or more input values and one output value. A function always produces the same output value for a given set of input values, as opposed to a function block that can produce differing output values each time it is executed with the same set of input values. This is because, in addition to the algorithm a function consists of, a function block also contains a data structure, in which values can be stored, even after the function block has finished executing. These stored values can influence the function block's result, which accounts for the possibility of differing outputs when executed with the same set of inputs.

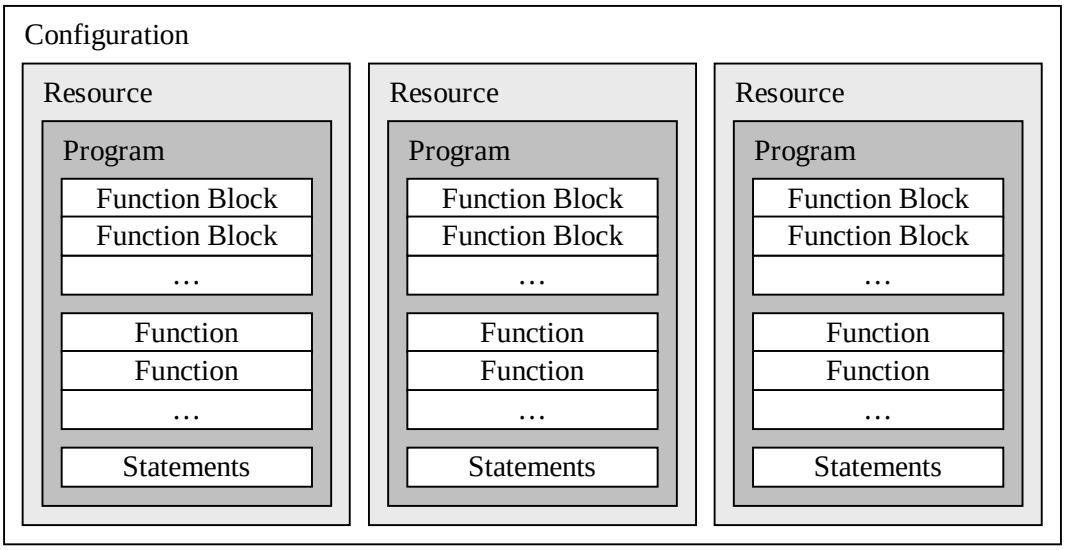

Figure 2: PLC software model

2.2 The languages

The third part of the IEC 1131 standard also contains a definition of five different programming languages, each of which can be used to create the program statements, both the statements that constitute the algorithm of a function or function block and the sequence of statements that appear "stand-alone" in the body of the program. These five languages are:

- Instruction List, a textual programming language resembling assembler.
- Structured Text, another textual programming language resembling Pascal.
- Ladder Diagram, a graphical programming language based on the graphical presentation of Relay Ladder Logic used by electrotechnical engineers.
- Function Block Diagram, another graphical programming language resembling electronic circuit diagrams.
- Sequential Function Chart, a third graphical programming language derived from Petri Nets.

The next section will take a closer look at one of these programming languages, the Structured Text programming language.

3 Structured Text

The Structured Text programming language is a high level programming language. Because the Structured Text language is based upon another high level programming language, namely Pascal, both languages bear strong resemblance syntactically. This resemblance is used to clarify the Structured Text syntax by comparing Pascal code to equivalent program pieces written in Structured Text. To be able to write pieces of Structured Text, the language constructs and the data types that these pieces are composed of have to be introduced first. This is done in the next two paragraphs.

3.1 The language constructs

One of the problems concerning Structured Text within the framework of the IEC 1131-3 standard is determining exactly what is part of the language and what is not. The standard defines Structured Text to be just the statements that can be used to construct function and function block algorithms and program bodies. However, within the standard, elements common to all five programming languages defined in the standard, such as elements that identify, for instance, a configuration, a resource or a program and the declaration of variables, are written using a syntax that also resembles the Pascal syntax in such a way that many of the people that have applied the standard have unjustly regarded these common elements as being part of the Structured Text programming language as well, in spite of the standard. Clearly, the standard isn't as clear on this as one would like a standard to be.

A comprehensive context-free description of the syntax of the language constructs that, according to the IEC 1131-3 standard, do belong to the Structured Text programming language is included in appendix A.

3.2 Data types

Like almost every other programming language, the IEC 1131-3 standard also allows the use of a variety of different types of data in Structured Text programs, both elementary and derived.

3.2.1 Elementary data types

The elementary data types provided by the standard are the following:

- Integers
- Floating point numbers
- Data types that represent time
- Strings
- Bit strings

3.2.1.1 Integers

IEC Data type	Description	Range
SINT	Short integer	-128 … +127
INT	Integer	-32768 … +32767
DINT	Double integer	-2^{31} … $+2^{31}-1$
LINT	Long integer	-2^{63} … $+2^{63}-1$
USINT	Unsigned short integer	0 … +255
UINT	Unsigned integer	0 … $+2^{16}-1$
UDINT	Unsigned double integer	0 … $+2^{32}-1$
ULINT	Unsigned long integer	0 … $+2^{64}-1$

Table 1: Integer data types

Examples: -18, 0, 1977 (decimal), 2#00010010 (binary), 8#22 (octal), 16#12 (hexadecimal)

3.2.1.2 Floating point numbers

IEC Data type	Description	Range
REAL	Real numbers	$\pm 10^{\pm 38}$
LREAL	Long real numbers	$\pm 10^{\pm 308}$

Table 2: Floating point data types

Examples: 18.7, 18.7E-77, 0,19e+77

3.2.1.3 Data types that represent time

IEC Data type	Description	Examples
TIME	The duration of time after an event	T#18d7h19m7s7ms TIME#18h7s
DATE	Calendar date	D#1977-07-18 DATE#1977-07-18
TIME_OF_DAY	Time of day	TOD#18:07:19 TIME_OF_DAY#23:59:59.99
DATE_AND_TIME	Date and time of day	DT#1977-07-18-18:07:19.77 DATE_AND_TIME#1977-07-18-12:00:00

Table 3: Time data types

3.2.1.4 Strings

IEC Data type	Description	Examples
STRING	Character strings	‘Hello world’ ‘’

Table 4: String data type

3.2.1.5 Bit strings

IEC Data type	Description
BOOL	Bit string of 1 bit
BYTE	Bit string of 8 bits
WORD	Bit string of 16 bits
DWORD	Bit string of 32 bits
LWORD	Bit string of 64 bits

Table 5: Bit string data types

Examples: 00010010, FALSE (equivalent to 0), TRUE (equivalent to 1)

3.2.2 Derived data types

As well as providing the elementary data types mentioned above the IEC 1131-3 standard also presents the Structured Text programmer with the ability to build totally new data types. A new data type is constructed in Structured Text by enclosing its definition in the keywords TYPE and END_TYPE. The definition consists of the name of the data type under construction, followed by a colon, followed by the kind of derived data type to be constructed. There are four kinds of derived data types:

- Structured data types
- Enumerated data types
- Sub-ranges data types
- Array data types

3.2.2.1 Structured data types

A structured data type is a composite data type, constructed by enclosing the elements of the composition in the keywords STRUCT and END_STRUCT. Each element consists of its name followed by a colon followed by its data type. A structured data type itself can contain one or more structured data types. For example:

```
TYPE Rectangle :
  STRUCT
    TopLeft : Point;
    Height  : INT;
    Width   : INT;
  END_STRUCT;
END_TYPE
```

In the example above, the `Point` data type is a structured data type itself.

3.2.2.2 Enumerated data types

An enumerated data type is constructed by enclosing the elements of the enumeration in parentheses. Each element consists of a different name. For example:

```
TYPE Color :
  (Red, White, Blue);
END_TYPE
```

3.2.2.3 Sub-ranges data types

A sub-ranges data type is constructed by limiting the range of another data type, usually integers. Limited versions of a data type consist of the name of the data type to be restricted, followed by a lower and an upper bound separated by two dots and enclosed in parentheses. For example:

```
TYPE Angle :
  INT(-180..+180);
END_TYPE
```

3.2.2.4 Array data types

An array data type is constructed by the keyword ARRAY, followed by a multi-dimensional vector of ranges of indices, followed by the keyword `OF`, followed by the (elementary or derived) data type that is going to be contained in the array. A multi-dimensional vector of ranges of inidices consists of a comma-delimited list of ranges of indices, enclosed in brackets. A range of indices consists of a lower and upper bound of the range, both integers, separated by two dots. For example:

```
TYPE Display :
  ARRAY[1..768, 1..1024] OF Color;
END_TYPE
```

3.3 Usage

In order to clarify the Structured Text syntax, the next few paragraphs contain a comparison between the following constructs that can be found in both the Structured Text programming language and Pascal: assignments, function calls, conditional statement and iteration statements. Each time the notable differences will be put forward.

3.3.1 Assignments

Structured Text	Pascal
`VarA := (VarB * 24 MOD 2 = 1) XOR (VarC <= VarD + 34);`	`VarA := (VarB * 24 MOD 2 = 1) XOR (VarC <= VarD + 34);`

`(* AnArray is an array containing twenty integers *)`
`AnArray := 10(1), 5(2), 5(3);`
	`(* AnArray is an array containing twenty integers *)`
	`FOR ElementNr := 1 TO 10 DO`
	`BEGIN`
	`  AnArray[ElementNr] := 1;`
	`END;`

	FOR ElementNr := 11 TO 15 DO
	BEGIN
	  AnArray[ElementNr] := 2;
	END;
	FOR ElementNr := 16 TO 20 DO
	BEGIN
	  AnArray[ElementNr] := 3;
	END;

VarA := AND(Var1, Var2, ..., VarN);	VarA := Var1 AND Var2 AND ... AND VarN;

Table 6: Assignments in Structured Text versus assignments in Pascal

Assignment-wise one major dissimilarity can be observed. Notice the ease with which an array can be initialized in the Structured Text language. Also notice the effort it requires to do the same in Pascal!

Operator-wise there is another difference between the two languages. In the Structured Text language, some of the operators are extensible, i.e. some operators can have any number of arguments (of the same type).

3.3.2 Function calls

Structured Text	Pascal

FUNCTION Ave_REAL : REAL
  VAR_INPUT
  Input1, Input2 : REAL;
  END_VAR
  Ave_REAL := (Input1 + Input2) / 2;
END_FUNCTION
 	
Average1 := Ave_REAL(5.0, 4.0);
Average2 := Ave_REAL(Input2 := 6.0);
(* Value 3.0 assigned to Average2 *)
	FUNCTION Ave_REAL
	  (Input1, Input2 : Real): Real;
	BEGIN
	  Ave_REAL := (Input1 + Input2) / 2;
	END;
	
	Average1 := Ave_REAL(5.0, 4.0);
	Average2 := Ave_REAL(0.0, 6.0);

Table 7: Function calls in Structured Text versus function calls in Pascal

Note that, strictly speaking, the FUNCTION … END_FUNCTION and the VAR … END_VAR constructs mentioned in this paragraph do not belong to the Structured text programming language, as discussed earlier.

3.3.3 Conditional statements

Structured Text	Pascal

IF VarB > 0 THEN
  IF VarC = 3 THEN
  VarA := TRUE;
  ELSE
  VarC := 3;
  VarA := FALSE;
  END_IF;
ELSIF VarB < 0 THEN
  VarA := FALSE;
ELSE
  VarC := 3;
  VarA := TRUE;
END_IF;
	IF VarB > 0
	THEN
	 BEGIN
	  IF VarC = 3
	  THEN VarA := TRUE
	  ELSE
	  BEGIN
	  VarC := 3;
	  VarA := FALSE;
	  END
	 END
	ELSE IF VarB < 0
	THEN VarA := FALSE
	ELSE
	 BEGIN
	  VarC := 3;

	VarA := TRUE; END;

CASE VarC OF
1, 2 : VarB := 1;
3 : VarB := 2; VarA := TRUE;
4..10 : VarB := 3;
ELSE
VarB := 0; VarA := FALSE;
END_CASE;
	CASE VarC OF
	0, 1 : VarB := 1;
	3 : BEGIN
	VarB := 2;
	VarA := TRUE;
	END;
	4..10 : VarB := 3;
	ELSE
	VarB := 0; VarA := FALSE;
	END;

Table 8: Conditional statements in Structured Text versus conditional statements in Pascal

The main difference between the Structured Text language and the Pascal language, when one looks at the conditional statements in both languages, is the explicit use of the `BEGIN` and `END` keywords in the Pascal code in cases that construct bodies contain two or more statements. Note the inconsistent omission of these keywords in the `ELSE` part of the Pascal `CASE` construct.

Another, minor, difference is the necessary omission of the semicolon before the `ELSE IF` or `ELSE` part of the `IF` construct in the Pascal programming language.

3.3.4 Iteration statements

Structured Text	Pascal

FOR VarB := 1 TO VarC DO
VarD := VarD + 1;
VarE := 2 * VarE;
END_FOR;
	FOR VarB := 1 TO VarC DO
	BEGIN
	VarD := VarD + 1;
	VarE := 2 * VarE;
	END;

FOR VarB := 100 TO 0 BY –2 DO VarD := VarD + VarB; END_FOR;	FOR VarB := 50 DOWNTO 0 DO VarD := VarD + 2 * VarB; END_FOR;

WHILE VarA AND (VarD <= VarC) DO
VarD := VarD + 1;
VarE := 2 * VarE;
VarA := VarE < 100;
END_WHILE;
	WHILE VarA AND (VarD <= VarC) DO
	BEGIN
	VarD := VarD + 1;
	VarE := 2 * VarE;
	VarA := VarE < 100;
	END;

REPEAT VarD := VarD + 1; VarE := 2 * VarE; VarA := VarE < 100; UNTIL NOT(VarA AND (VarD <= VarC));	REPEAT VarD := VarD + 1; VarE := 2 * VarE; VarA := VarE < 100; UNTIL NOT(VarA AND (VarD <= VarC));

Table 9: Iteration statements in Structured Text versus iteration statements in Pascal

As was the case with conditional statements, the primary difference between the Structured Text language and the Pascal language, when one looks at the iteration statements in both languages, again, is the explicit use of the `BEGIN` and `END` keywords in the Pascal code in cases that construct bodies contain two or more statements. Note another inconsistent omission of these keywords in the Pascal `REPEAT` construct.

Another difference is the possibility of including a `BY` part in the `FOR` construct of the Structured Text language, indicating the magnitude of the step taken every time a `FOR` loop is executed. The need of a similar construct in the Pascal language can, awkwardly, be circumvented by slightly adjusting

the statements within the FOR construct, possibly combining this with the use of the DOWNTO construct available in the Pascal language, as illustrated by the example above. This DOWNTO construct in Pascal is equivalent to using the TO construct in combination with a BY -1 in the Structured Text programming language.

3.4 Fundamental differences between Structured Text and Pascal

In the last section a couple of minor, mostly syntactical, dissimilarities between the Structured Text programming language and Pascal have been mentioned. There are, however, more fundamental differences between the two. These are discussed in the following paragraphs.

3.4.1 Function blocks

The first fundamental difference is the fact that the concept of function block, as discussed earlier, has no precise Pascal equivalent. This discrepancy can be accounted for, however, as everything that can be done by function blocks in Structured Text can be achieved by the appropriate use of functions or procedures and (global) variables in Pascal.

3.4.2 Directly represented variables

Another concept occurring in the Structured Text language but not found in Pascal, is the notion of directly represented variables. Directly represented variables are variables that are not identified by a name, but that are addressed by their position in the PLC's data memory. This concept has been adopted from programming languages, such as the Ladder Diagram programming language defined by the IEC 1131-3 standard, that make use of so-called ladder diagrams. Another reason why it has been included in Structured Text is the fact that it is commonly used in many PLCs.

However, the kind of identification that directly represented variables advocate is extremely implementation-dependent and therefore should have no place in a standard such as IEC 1131-3 that addresses implementation-independent issues.

3.4.3 Default initial values

Another difference between the Structured Text language and the Pascal language is the fact that Structured Text variables do not have to be initialized explicitly. When an explicit initialization of a variable is omitted that variable will default to the value all variables of its type default to. Integers, floating point numbers, times and bit strings default to zero, dates default to 0001-01-01 and strings default to the empty string. Derived data types default to the default values of the elementary types they are derived from, unless a default value is contained within the type definition, in which case the default values of the elementary data types are overridden.

This concept of default initial values allows parameters to be omitted from function calls, in which case the unused input parameter will take the default value as given in the function type definition. An example of this can be seen in paragraph 3.3.2, where the value 3.0 is assigned to Average2.

3.4.4 Several output values

A fourth fundamental difference between the two languages is the fact that a function in the Structured Text language can have more than one output value, without them having to be packed together in some kind of structure. If a function has more than one output value, these extra output values are in the output variable section (labelled VAR_OUTPUT) or in the section intended for variables that serve as input as well as output (labelled VAR_IN_OUT).

In the IEC 1131-3 standard, one particular example of the construction mentioned above can be found to play a part in another phenomenon common to the Structured Text language, but unknown to the Pascal programming language: bonus parameters for free. With every function, two bonus boolean

parameters are supplied gratuitously: an EN input parameter indicating whether the function body is executed or not and an ENO output parameter indicating whether the execution of the function has resulted in an error of some kind. In case the latter parameter indicates the occurrence of an error, the function's other result(s) is (are) undetermined.

3.4.5 Overloading

A fifth difference is the fact that the Pascal language does not support overloading, contrary to the Structured Text programming language. Pascal does, however, support the concept of coercion. It is because of this, and not because of the support of overloading, that it is possible in Pascal that the square function, which, according to its definition, requires a real argument, can be supplied with an integer argument.

Note that the Structured Text programming language does not allow one to create an overloaded function oneself. The only overloaded functions existing within the Structured Text language are functions offered by Structured Text itself.

3.4.6 Recursion

The sixth and final fundamental difference, and maybe the most important one, is the total absence of the notion of recursion within the Structured Text programming language. The IEC 1131-3 standard specifically prohibits functions, function blocks and programs to be recursive, that is, their invocation should not cause the invocation of another function, function block or program of the same type. One can only guess at the why and wherefore of this decision, because when it comes to explaining the reasons of banning recursion from within the confines of the Structured Text language, the standard remains silent. One possible reason could be that a program written in the language as defined by the IEC 1131-3 standard can simply be viewed as one big collection of macro-expansions, thus rendering the processing of a Structured Text program less complex. Introducing recursion into the game would totally annul this view. The question is, however, whether the reduction in complexity is as significant as the advocates of the omission of recursion claim it to be, compared to the increase entailed by the introduction of other language elements such as the concept of overloading mentioned above. With all the concepts that have been included in it, the Structured Text language has long outgrown the characterising traits of a basic programming language and one could wonder why stop here and not include recursion? Other possible reasons against allowing recursive Structured Text are the difficulty of testing and the unpredictability of its real-time performance. These are both perfectly valid reasons, but, strangely enough, none of them are used to give an explanation for the absence of recursion in IEC 1131-3. In conclusion, one can say that the unaccounted omission of recursion definitely constitutes one of the standard's major flaws.

4 Conclusions

After a brief introduction to the universe of the Programmable Logic Controller, this paper discussed one of five ways defined by the IEC 1131-3 standard to program this device: the Structured Text programming language. A comparison has been made between Structured Text and the language upon which it was based, the Pascal programming language. This comparison revealed a great similarity existing between the two. Of the few distinctions that have been uncovered, the absence of the ability to write recursive programs in the Structured Text language is the most remarkable one, especially because of the undocumented nature of this omission, as a result of which the choice of what programming constructs to include in the language and what not gives the impression of being a fairly arbitrary one.

References

[HAB] Wolfgang A. Habing and Krzystof M. Sacha. Programmable Logic Controllers. *Real Time Systems: Implementation of Industrial Computerized Process Automation.*

[IEC] International Electrotechnical Commission, Technical Committee No. 65: Industrial-process Measurement and Control, Sub-committee 65B: Devices, Working Group 7: Programmable Controllers. *Working Draft = IEC 61131-3, 2nd Ed. Programmable Controllers – Programming Languages.*

[LEW] R.W. Lewis. *Programming Industrial Control Systems using IEC 1131-3.*

[TOU] Konstantinos Tourlas. *An assessment of the IEC 1131-3 Standard on Languages for Programmable Controllers.*

Appendix A:
The syntax of the Structured Text programming language

A.1 Expressions

Production rules:	
Expression	::= XOR_Expression {'OR' XOR_Expression}
XOR_Expression	::= AND_Expression {'XOR' AND_Expression}
AND_Expression	::= Comparison {('&' | 'AND') Comparison}
Comparison	::= EquExpression { ('=' | '<>') EquExpression}
EquExpression	::= AddExpression {ComparisonOperator AddExpression}
ComparisonOperator	::= '<' | '>' | '<=' | '>='
AddExpression	::= Term {AddOperator Term}
AddOperator	::= '+' | '-'
Term	::= PowerExpression {MultiplyOperator PowerExpression}
MultiplyOperator	::= '*' | '/' | 'MOD'
PowerExpression	::= UnaryExpression {'**' UnaryExpression}
UnaryExpression	::= [UnaryOperator] PrimaryExpression
UnaryOperator	::= '-' | 'NOT'
PrimaryExpression	::= Constant | Variable | '(' Expression ')' | FunctionName '(' [ST_FunctionInputs] ')'
ST_FunctionInputs	::= ST_FunctionInput { ',' ST_FunctionInput}
ST_FunctionInput	::= [VariableName ':='] Expression

Table 10: Production rules of expressions in Structured Text

A.2 Statements

Production rules:	
StatementList	::= Statement ';' {Statement ';'}
Statement	::= NIL | AssignmentStatement | SubprogramControlStatement | SelectionStatement | IterationStatement

Table 11: Production rules of statements in Structured Text

A.3 Assignment statements

Production rule:	
AssignmentStatement	::= Variable ':=' Expression

Table 12: Production rule of assignment statements in Structured Text

A.4 Subprogram control statements

Production rules:	
SubprogramControlStatement	::= FB_Invocation | 'RETURN'
FB_Invocation	::= FBName '(' [FB_InputAssignment {',' FB_InputAssignment}] ')'
FB_InputAssignment	::= VariableName ':=' Expression

Table 13: Production rules of subprogram control statements in Structured Text

A.5 Selection statements

Production rules:	
SelectionStatement	::= IfStatement | CaseStatement
IfStatement	::= 'IF' Expression 'THEN' StatementList {'ELSIF' Expression 'THEN' StatementList} ['ELSE' StatementList] 'END_IF'
CaseStatement	::= 'CASE' Expression 'OF' CaseElement {CaseElement} ['ELSE' StatementList] 'END_CASE'
CaseElement	::= CaseList ':' StatementList
CaseList	::= CaseListElement {',' CaseListElement}
CaseListElement	::= Subrange | SignedInteger

Table 14: Production rules of selection statements in Structured Text

A.6 Iteration statements

Production rules:	
IterationStatement	::= ForStatement | WhileStatement | RepeatStatement | ExitStatement
ForStatement	::= 'FOR' ControlVariable ':=' ForList 'DO' StatementList 'END_FOR'
ControlVariable	::= Identifier
ForList	::= Expression 'TO' Expression ['BY' Expression]
WhileStatement	::= 'WHILE' Expression 'DO' StatementList 'END_WHILE'
RepeatStatement	::= 'REPEAT' StatementList 'UNTIL' Expression 'END_REPEAT'
ExitStatement	::= 'EXIT'

Table 15: Production rules of iteration statements in Structured Text