ST Programming Manual
MasterTool Extended Edition
MT8000

Rev. C 08/2010
Doc.: MP399603

altus

No part of this document may be copied or reproduced in any form without the prior written consent of Altus Sistemas de Informática S.A. who reserves the right to carry out alterations without prior advice.

According to current legislation in Brazil, the Consumer Defense Code, we are giving the following information to clients who use our products, regarding personal safety and premises.

The industrial automation equipment, manufactured by Altus, is strong and reliable due to the stringent quality control it is subjected to. However, any electronic industrial control equipment (programmable controllers, numerical commands, etc.) can damage machines or processes controlled by them when there are defective components and/or when a programming or installation error occurs. This can even put human lives at risk.

The user should consider the possible consequences of the defects and should provide additional external installations for safety reasons. This concern is higher when in initial commissioning and testing.

The equipment manufactured by Altus does not directly expose the environment to hazards, since they do not issue any kind of pollutant during their use. However, concerning the disposal of equipment, it is important to point out that built-in electronics may contain materials which are harmful to nature when improperly discarded. Therefore, it is recommended that whenever discarding this type of product, it should be forwarded to recycling plants, which guarantee proper waste management.

It is essential to read and understand the product documentation, such as manuals and technical characteristics before its installation or use.

The examples and figures presented in this document are solely for illustrative purposes. Due to possible upgrades and improvements that the products may present, Altus assumes no responsibility for the use of these examples and figures in real applications. They should only be used to assist user trainings and improve experience with the products and their features.

Altus warrants its equipment as described in General Conditions of Supply, attached to the commercial proposals.

Altus guarantees that their equipment works in accordance with the clear instructions contained in their manuals and/or technical characteristics, not guaranteeing the success of any particular type of application of the equipment.

Altus does not acknowledge any other guarantee, directly or implied, mainly when end customers are dealing with third-party suppliers.

The requests for additional information about the supply, equipment features and/or any other Altus services must be made in writing form. Altus is not responsible for supplying information about its equipment without formal request.

Summary

1. Introduction

ST Language (Structured Text) is a structured text language, high level, with resources similar to C and Pascal languages. It can be used for writing programs with commands IF, THEN, ELSE, loops FOR and WHILE, local variables and vectors can be created, creation and calling of sub-routines, etc. It is an alternative to the use of Ladder graphic language, as it can access operands.

Programming in ST language is facility of MasterTool XE Advanced. It is available to use with CPU AL-2004, as well as Ponto Series PLCs. Through this language it is possible to create procedure modules (P module) and function modules (F modules), as defined by the norm IEC-1131-3.

After creation, the ST modules have the same characteristics as the other F of P modules already existent. The parameters number of input and output can be configured. The calling of new ST module must be done on the application program Ladder with CHF or CHP instruction, as it is done to other modules F and P.

The module created in ST language can be read again from the microcomputer PLC. In this case, the source program ST is normally restored, even with its comments. There are passwords to protect against not allowed reading and/or modifying, protecting the designer technology. MasterTool XE implements a subgroup of ST Language, and it includes most of the structures and data commands defined by IEC-61131-3.

The execution performance of the PLC of a ST module is better than an equivalent module in Ladder, as the ST module is verified as a whole, while in Ladder it is divided in many instruction calls.

The main benefits of the use of ST Language are:

- Another option on programming language, according to international norm
- Possibility of creating programming, according to international norm
- Text language instead of graphic: copy/paste/substitute or traditional editors text macros
- Shorter time to developing, resulting in less engineering costs
- Better execution performance

The following picture is an example of ST Language use on MasterTool XE:

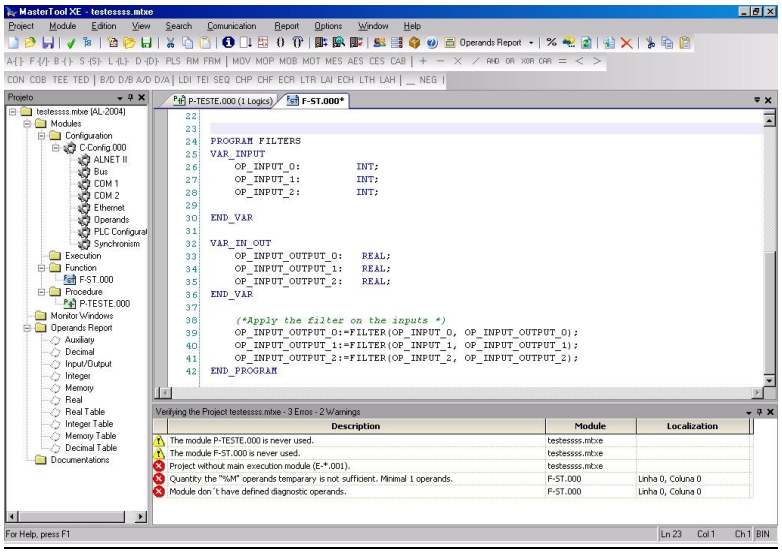

Figure 1-1. Example of use of ST language

The software MasterTool Extended Edition has three distribution versions, each one with an optimized profile, in accordance with the necessity of the user. They are:

- **Lite**: programming software specific for small applications. This version does not support ST.
- **Professional**: programming software with tools for all lines of Altus CLPs.
- **Advanced**: programming software with tools for bigger applications

Each version has its characteristics, ends and functions specified for each purpose. Further details about these differences can be seen on Master Tool XE Using Manual.

Documents Related to this Manual

To get additional information about the use of ST Language other documents can be consulted (manuals and techniques characteristics) beyond this. These documents are available in its last revision at www.altus.com.br.

Each product has a document called Technical Characteristics (CT), where the characteristics of the product can be found. The product can have, in addition, User's Manual (the codes are cited on the CTs)

The following documents are suggested as source of additional information:

- Technical Characteristics MT8000
- MasterTool XE Using Manual
- MasterTool XE Programming Manual

Visual Inspection

Before proceeding the installation, it is recommendable to make a careful visual inspection of the material, verifying if it does not have damages caused by the transport. Check if all the products received are in perfect state. In case of defects, inform the transport company and the nearer Altus representative.

It is important to register the serial number of each equipment received, as well as the revisions of software, in case it exists. This information will be necessary in case of need to contact the Altus technical support.

Technical Support

To contact Altus technical support in São Leopoldo, RS, Call +55-51-35899500. To know Altus technical support centers in others localities, check our site (www.altus.com.br) or send an email to altus@altus.com.br.

If the equipment is already installed, please have the following information upon requesting the assistance:

- The model of the equipments used and the configuration of the installed system.
- The PLC serial number.
- Equipments review and executive software version, in the label on the side of the product.
- Information about the operating mode of the PLC, obtained through Master Tool programmer.
- The content of the application program (modules), obtained through MasterTool programmer.
- The version of the programmer used.

Warning Messages Used on this Manual

On this manual, the messages of warnings will be presented in the following formats and meanings:

DANGER: indicates a risk to life, production, serious harm to people, or that substantial material or environmental damage may happen it the necessary precautions are not taken.

WARNING:
Indicates configuration, application and installation details that *must* be followed to avoid situations that can cause system errors and related consequences.

ATTENTION:
Indicates important configuration, application or installation details to obtain the maximum performance of the system.

2. Technical Description

This section presents the technical characteristics of ST Editor.

Software Characteristics

Types of Data

The available types of data are shown in the following table:

	Description	Bits	Related PLC Operand
BOOL	Boolean	1	Bit of %A, %E, %S or %M
BYTE	8 bits sequence	8	Operand %A, %E or %S
WORD	16 bits sequence	16	Operand %M or %TM
DWORD	32 bits sequence	32	Operand %I or %TI
USINT	Short int not signed	8	Operand %A, %E or %S
INT	Integer	16	Operand %M or %TM
DINT	Long Integer	32	Operand %I or %TI
REAL	Real	32	Operand %F or %TF

Table 2-1. Types of data

A variable is a memory area that stores a value defined by its type of data. All the VARIÁVEIS must be declared before its use. Its scope is limited to the function or the program in which they were declare, allowing that the names can also be used in other parts of the software, without the occurrence of any conflict.

The declared variables can be associated to operands of the programmable controller, so, allowing the creating of algorithms to the execution of the control of the desired industrial process.

Software Limits

	Description
Call nested functions	16 function call
Size of módule P or F	Up to 32KB
Data area for variables	Up to 3KB[1]
Function quantity	It is size limit of module.

Table 2-2. Software limits

[1] To variables declared between VAR..END_VAR and are not mapped to PLC operand, and to variables alloced automatically for MasterTool XE and used in temporary operations.

PLC Operands

A module developed through ST language can access CLP operands. The following table shows the PLC operands, as well as its compatibility with ST Language:

Operand Type	Can be used by ST ?
%A, %E and %S	Yes
%M and %TM	Yes
%I and %TI	Yes
%F and %TF	Yes
%D and %TD	No

Table 2-3. Operands type

The variables used on ST modules must be declared before being used. PLC operands cannot be used directly on the ST function, but it is possible to place those addresses on the variables declared on the function.

Operators

The operators of arithmetic expressions and logic in ST are similar to other languages, as shown in the following table:

Type	Operators	
Mathematics	+	Sum
	-	Subtraction
	*	Multiplication
	/	Division
	MOD	Rest
Logics	& AND	Operation “AND”
	OR	Operation “OR”
	XOR	Operation “OR” exclusive
	NOT	Operation “NOT”
Comparing	<	Less Than
	>	Greater Than
	<=	Lesser or Equal
	>=	Greater or Equal
Equals	=	Equal
	<>	Difference

Table 2-4. Operators and arithmetic expressions

Commands

The following table shows the commands of ST language:

Typo	Commands
Repetition	WHILE REPEAT FOR
Selection	IF CASE

Table 2-5. Commands of ST language

Items not Implemented on IEC 61131-3 Standard

The following items are part of the definitions of the norm IEC 61131-3 and are not implemented in this product:

- Types of data SINT, UINT, UDINT, LREAL
- Types of data TIME, DATE e STRING
- *Enumerated, Sub-Range*
- *Arrays* with more than one dimension (matrixes)
- *Structs*
- *Function Blocks*
- *Resource, Tasks, Configurations*

3. Procedures

In this section is presented an example of how to generate a procedure module or a function module with ST Editor. In both examples, a digital filter program that must be applied on three input operands will be used.

Creating a Module P or F in ST Language

Once MasterTool XE is executed, the option Module/New must be selected. The following screen will be shown:

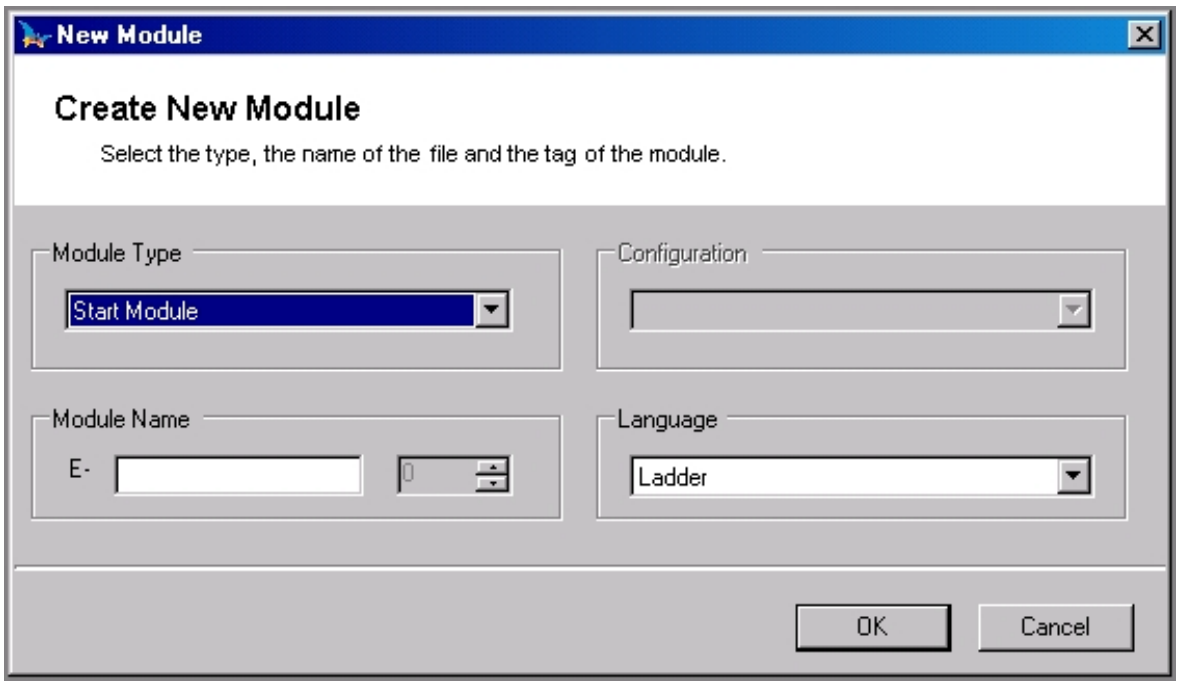

Figure 3-1. Creating a ST module

Selects the following options in "Type of Module": "Module ST Function" or "Module Procedure ST". In the box of selection "Language" selects "ST". ST Editor will be executed and will be ready for the edition, after this selection OK button.

To follow an example of a possible configuration for a new module of Function in Language ST:

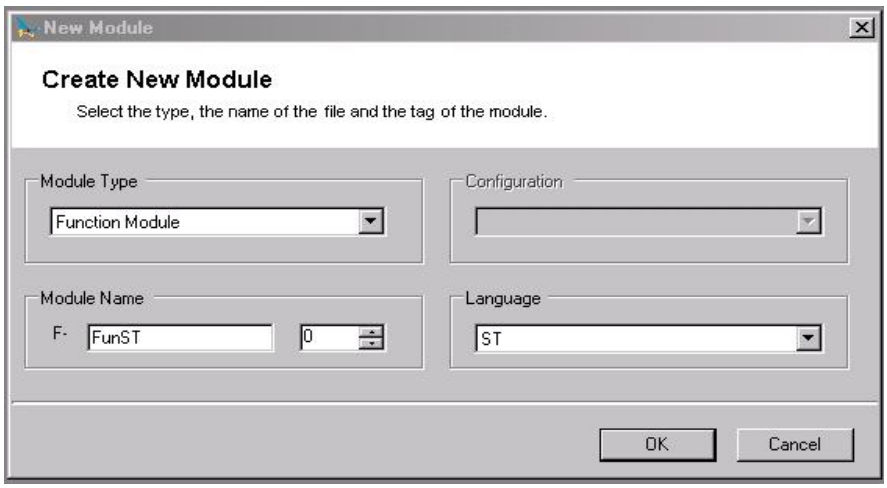

Figure 3-2. Creating a new ST module

Select the option “Module ST Function” or “Module Procedure ST”. The ST Editor will be executed and it will be ready for editing the chosen module after selecting OK button.

Declaring Global Variables

Further details about the declaration and the types of variables can be found on the section **Programming** on the chapter Variables. In this example the constants of the filter are declared as constants of global use.

```
VAR CONSTANT
FILTER_TIME0:          REAL:=0.1;      (*Time constant of the filter*)
SAMPLE_TIME:           REAL:=0.02;     (*Sample time period*)
END_VAR
```

Creating a Function to Implement a Filter

In this example more than an input value is filtered and for that it is necessary the use of a function. This function applies a filter whose time constant and sample time are defined as global variables. It is a recursive filter, so, besides its input value (INPUT), a parameter is necessary as the last filtered value (PREVIOUS OUTPUT).

```
VAR CONSTANT
       FILTER TIME0:          REAL:=0.1;      (*Time constant of the filter*)
       SAMPLE TIME:           REAL:=0.02;     (*Sample time period*)
END_VAR

FUNCTION FILTER:      REAL

VAR_INPUT
       INPUT:                 INT;            (*Input data to be filtered*)
       LAST_OUTPUT:           REAL;           (*Last filter result*)
END_VAR

VAR
       FILTER_FACTOR:         REAL;           (*Local variable to the filter factor*)
END_VAR

       (*Store the division of the sample time and the filter time constant*)
       (*in a local variable*)
       FILTER FACTOR := SAMPLE TIME/FILTER TIMEO;

       (*Apply the filter on the input *)
       TOFILTER := LAST_OUTPUT + (INPUT - LAST_OUTPUT)* FILTER_FACTOR;

END_FUNCTION
```

Writing the Main Code – PROGRAM – to a P Module

The aim of this code is applying a filter defined by the function FILTER on three input operands (%M200, %M201 e %M202). For it, it is necessary to keep the results of the last filtering and, as the result of each filter call will be the previous result in the next call, they can be the same operands (%F0, %F1 e %F2).

```
PROGRAM FILTER
VAR
       INPUT_0              AT %M200:    INT;
       INPUT_1              AT  %M201:    INT;
       INPUT_2              AT  %M202:    INT;
       LAST_RESULT_0        AT  %F0:      REAL;
       LAST_RESULT_1        AT  %F1:      REAL;
       LAST_RESULT_2        AT  %F2:      REAL;
       RESULT_0             AT  %F0:      REAL;
       RESULT_1             AT  %F1:      REAL;
       RESULT_2             AT  %F2:      REAL;
END_VAR

(*Apply the filter on the inputs*)
RESULT_0:= TOFILTER(INPUT_0, LAST_RESULT_0);
RESULT_1:= TOFILTER(INPUT_1, LAST_RESULT_1);
RESULT_2:= TOFILTER(INPUT_2, LAST_RESULT_2);

END_PROGRAM
```

Writing the Main Code – PROGRAM – to a F Module

In case of F module, the input and output operands can be declared through input parameters. In this case there are 6 input parameters (%M200, %M201, %M202, %F0, %F1, %F2).

Here there is an important characteristic on the passage of parameters, where the three first operands are operands exclusively input, but the three last operands are input and output as, it is a recursive filter, the last result will be used as input of the following iteration. They must be all declared on the input area of the CHF and discriminated in input and/or output in the respective declaration on ST module. To further details check the chapter Passage of Parameters to a F Module on the section **Programming**.

```
PROGRAM FILTER
VAR_INPUT
OP_INPUT_0:          INT;
OP_INPUT_1:          INT;
OP_INPUT_2:          INT;
END_VAR

VAR_IN_OUT
OP_INPUT_OUTPUT_0:    REAL;
OP_INPUT_OUTPUT_1:    REAL;
OP_INPUT_OUTPUT_2:    REAL;
END_VAR

(*Apply the filter on the inputs *)
OP_INPUT_OUTPUT_0:=FILTER(INPUT_0, OP_INPUT_OUTPUT_0);
OP_INPUT_OUTPUT_1:=FILTER(INPUT_1, OP_INPUT_OUTPUT_1);
OP_INPUT_OUTPUT_2:=FILTER(INPUT_2, OP_INPUT_OUTPUT_2);

END_PROGRAM
```

Diagnostic Operands

The diagnostic operands are used to report for user any error in execution time. The operands can be configured through menu *Module / Operands / Diagnostic*. If the operands were not configured an error is reported in verification time. The errors code can be seen in section **Errors in Execution Time**.

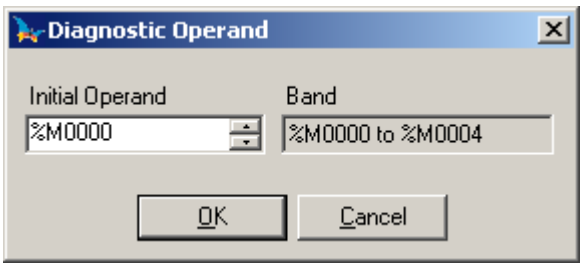

Figure 3-3. Window of configuration of the diagnostics operands

The diagnostic operands can be the same for all ST modules.

Temporary Operands

The temporary operands are used to solve some operations that use variables of type DINT and REAL. It is an optional configuration, but if the compiler needs it will be reported on verification time. The operands can be configured through menu *Module / Operands / Temporary*. The maximum quantity of the operands, if required, is 4.

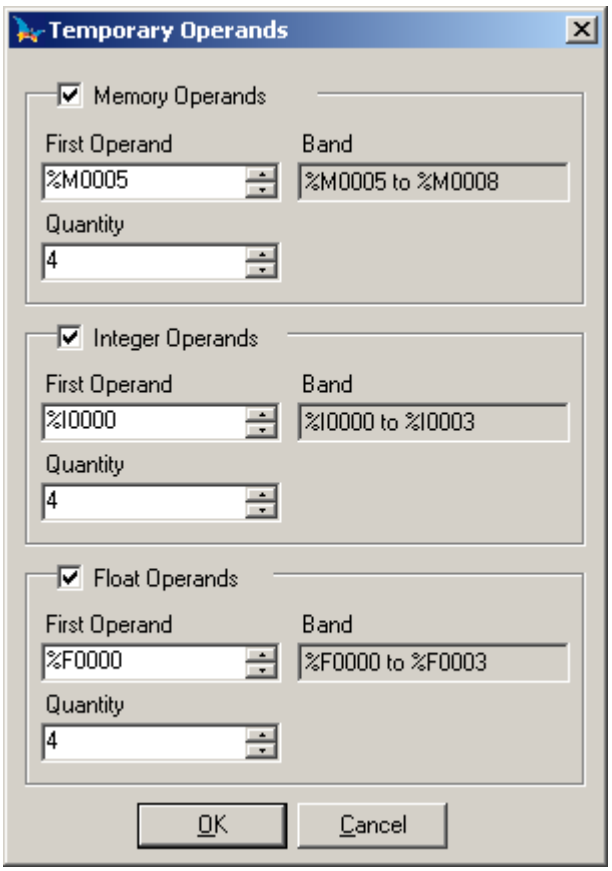

Figure 3-4. Window of configuration of the temporary operands

The temporary operands can be the same for all ST modules.

Verifying the Code

Before sending the module to the PLC, the user should verify the program, it is, make sure that there are not many programming errors on the module. For that, use the command verify of the menu module. If there are errors, they will be shown on the bottom of the window.

Saving the Code

Whenever a code written on ST language is saved in disk, it will be verified. The CÓDIGO FONTE is always saved on the module, but if the verification shows errors no executable code is added to the module.

WARNING: MasterTool XE only allows sending modules without errors to the PLC. However, the previous versions of MasterTool XE do not execute it, so in this case a module with error can be sent to the PLC. But due to the verification error, the module will not execute it and an error in execution time will happen.

Using a Module in ST

The ST module is used as modules in Ladder. It must be called through another ladder module using the CHP instructions (to P modules) or the instructions CHF (to F modules).

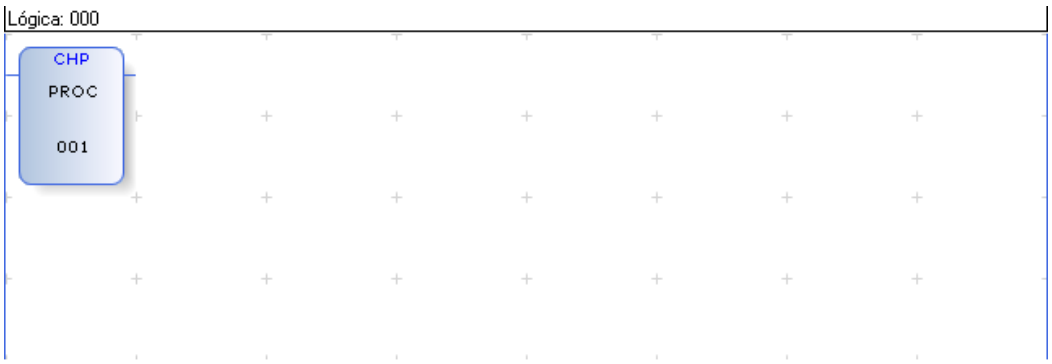

Figure 3-5. Calling of P module written in ST

4. Programming

This chapter describes the rules to write a program in ST language, presenting its syntax, semantic rules and the structure of the source code.

Structure of a Module in ST Language

A module in ST language is composed by:

- Global Variables (not mandatory)
- Functions, (not mandatory)
- Main program

Its basic structure must be as following:

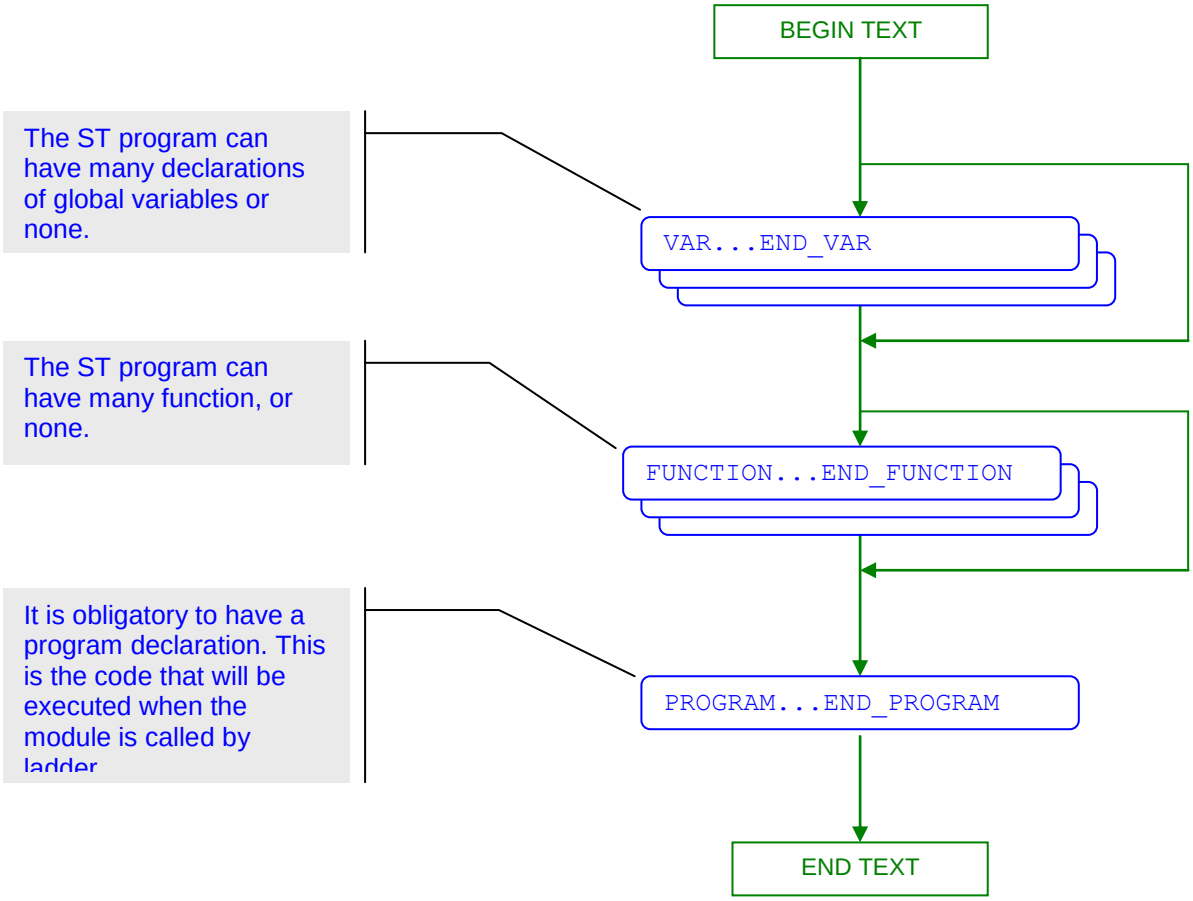

Figure 4-1. Structure of a program in ST language

Elements of ST Language

The elements of ST Language are all the symbols, numbers, operators and other punctuation used by the language and that cannot be textually divided. The ST language elements are:

- Identifiers
- Numerical constants

- Boolean constants
- Empty space
- Comment
- Operators and other punctuation signs

The following table shows some definitions of the elements:

Char	Any char from ‘a’ to ‘z’, capital or small
Digit	Any digit from ‘0’ to ‘9’
Binary digit	Any digit from ‘0’ to ‘1’
Octal digit	Any digit from ‘0’ to ‘7’
Hexadecimal digit	Any digit from ‘0’ to ‘9’, or any char from ‘a’ to ‘f’, capital or small

Table 4-1. Notations to represent values and data

Identifiers

An identifier is used to name different elements in the language, being an unique element in its scope. It is formed by letters, numbers and by the character subscribed (“_”). It must start by letter or by the subscribed character, but it cannot have two consecutive subscribed characters and cannot end with this character.

Only the first 32 characters of the sequence. The others will not have meaning and will be ignored. The use of capital or small letters does not have meaning on the identifier, it means, Level_Vase1, level_vase1 or Level_vase1 are distinct identifiers

The following table has some example of valid and non-valid identifiers:

Valid Identifiers	• `PumpStatus` • `ALM_PressureHigh` • `B1_ACTIVE` • `_1001_1` • `pressureTest`
Invalid Identifiers	• `__Test` • `Pump__B3214` • `Valve-125201` • `154_TurnOn` • `Pressure_`

Table 4-2. Identifiers

WARNING:
An identifier can have up to 32 characters. The characters up this limit are not considered. The identifier name is not case sensitive.

Empty Space

The space characters, tab and break line are considered empty spaces, it means, they can be used freely among the other elements of ST language.

Comments

Comments on the source code must be done between “(*” and “*)”.

Many lines can be commented with the same block. However it is not allowed the use of comments nested as: "(* ... (* ... *) ... *)".

Numerical Constants:

There are two types of constants: integer and real. In both cases, a number can have several characters of subscribed inside it. This character does not have meaning, it serves only to make a number more legible. However the subscribed character cannot start or finish a number, as well as it is not allowed the use of two subscribed characters together.

An integer number can also be expressed in binary, octal or hexadecimal bases. For this, one of the prefixes **2#, 8#** and **16#** must be used before the number, as can see below.

```
2#0000 1001  (* Binary constant equal to 9   *)
2#1111 1111  (* Binary constant equal to 255 *)
8#457        (* Octal constant equal to 303 *)
16#00AA      (* Hexa constant equal to 170 *)
16#8000      (* Hexa constant equal to 32768 *)
```

The following table brings some examples of numerical literal valid and non-valid:

	Example	Value in decimal
Valid numerical literal	• 12547 • 10_531 • -1_532 • • 6978 • 2#0000_1001 • 2#01001100 • 2#001 • 8#457 • 8#6254 • 16#4576_ab4f • 16#980ABE4D • 16#FFFF_ffff	• 12547 • 10531 • -1532 • • 6978 • 9 • 76 • 1 • 303 • 3244 • 1165405007 • 2550840909 • 4294967295
Invalid numeric literal	• _1546 • 4578_ • -_5447 • 10__135 • #0010001 • 2#4577_0158#00159 • 16#_1789 • 16#4587_	

Table 4-3. Numerical literal

Real numerical literal has the decimal point "." between the integer and the fraction part. The exponent of the base 10 can be specified using the prefixes 'E' or 'e'. In case it is not informed, the value 0(zero) will be considered. The table bellow shows examples:

Valid numeric literal	• -1.34E-2 • 1.0e+6 • 1.234E6 • 1_254.4879
Invalid numeric literal	• 1.34E-2.10 • e +6 • _254.4879

Table 4-4. Numeric literal with decimal base

Boolean constants

For boolean constants the words TRUE or FALSE can be used; or its numerical equivalents 1 or 0.

Types of Data

The type of data defines the manner in which the data can be stored on the PLC memory. This section defines the possible types of data as well as the conversion functions applicable from a type to another.

Basic Types of Data

ST language used the following types of data:

Type	Description	Bytes	Band
BOOL	Boolean	1	FALSE, TRUE, 1 or 0.
BYTE	8 bits sequence	1	Note 1
WORD	16 bits sequence	2	Note 1
DWORD	32 bits sequence	4	Note 1
USINT	Short integer	1	0 .. 255
INT	Integer	2	- 32.768 .. 32.767
DINT	Double integer	4	-2.147.483.648 .. 2.147.483.647
REAL	Real	4	$\pm 10^{\pm 38}$ Note 2

Table 4-5. Data basic types

Note 1: Numerical band is not applicable to the types BYTE, WORD and DWORD.

Note 2: the floating point with simple precision is defined by the norm IEC 559 (IEEE 754).

Classifying Data

The several types of data are classified in groups according to a hierarchy. Afterwards, it will be seen that the use of operators usually is limited to determined types of a same group.

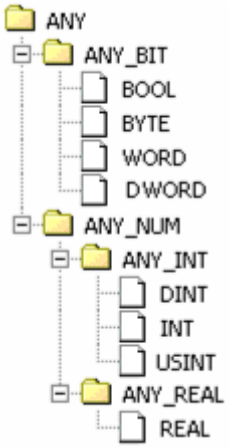

Figure 4-2. Data classification

Types Conversion

The variable can be converted to another format through the converting type functions. The converting of a type to another can occur on an implicit manner, when the verifier inserts a conversion function automatically; or in an explicit manner, when the user must insert the conversion function.

The conversion functions use the format:

```
<source type>_TO_<target type>( <value to convert> )
```

The implicit conversions among types are possible since there is no risk of losing information, and that there is compatibility among types. The following figure presents all the possible implicit conversions between two types:

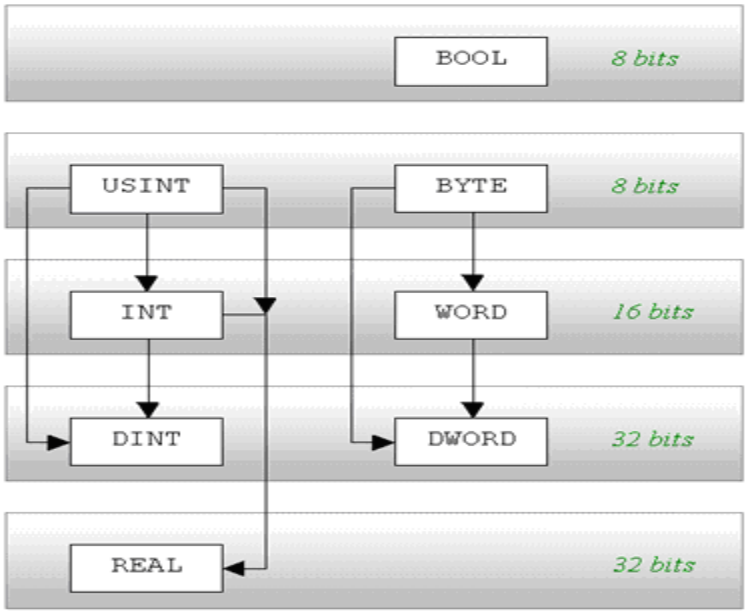

Figure 4-3. Implicit conversions

The following table shows the operations carried out on the conversion of types. It can be observed, that even the implicit conversions have conversion functions to the case of conversion before a mathematics operation whose result requires more numerical resolution.

Source	Target	Comment	Conversion	Function
BOOL	BYTE	If the source is TRUE the target will be 1. If the source is FALSE the target will be 0.	Explicit	BOOL_TO_BYTE
	WORD		Explicit	BOOL_TO_WORD
	DWORD		Explicit	BOOL_TO_DWORD
	USINT		Explicit	BOOL_TO_USINT
	INT		Explicit	BOOL_TO_INT
	DINT		Explicit	BOOL_TO_DINT
	REAL		Explicit	BOOL_TO_REAL
BYTE	BOOL	If the source is zero the target will be FALSE, on contrary case will be TRUE.	Explicit	BYTE_TO_BOOL
	USINT	The target will receive a value from 0 to 255.	Explicit	BYTE_TO_USINT
	WORD	Convert the absolute value from source to the target format. The values will remain unchanged.	Implicit	BYTE_TO_WORD
	DWORD		Implicit	BYTE_TO_DWORD
	INT		Explicit	BYTE_TO_INT
	DINT		Explicit	BYTE_TO_DINT
	REAL		Explicit	BYTE_TO_REAL

Source	Target	Comment	Conversion	Function
WORD	BOOL	If the source is zero the source will be FALSE, on contrary case will be TRUE.	Explicit	WORD_TO_BOOL
	BYTE	The target will receive the low byte of the source.	Explicit	WORD_TO_BYTE
	USINT		Explicit	WORD_TO_USINT
	INT	The target will receive a value from –32768 to 32767, and can assume negative values.	Explicit	WORD_TO_INT
	DWORD	Convert the absolute value from source to target. The values will remain unchanged.	Implicit	WORD_TO_DWORD
	DINT		Explicit	WORD_TO_DINT
	REAL		Explicit	WORD_TO_REAL
DWORD	BOOL	If the source is zero the target will be FALSE, on contrary case will be TRUE.	Explicit	DWORD_TO_BOOL
	BYTE	The target will receive the low byte of the source.	Explicit	DWORD_TO_BYTE
	USINT		Explicit	DWORD_TO_USINT
	WORD	The target will receive the low word of the source.	Explicit	DWORD_TO_WORD
	INT	The target will receive the binary form of the source, and it can have negative values.	Explicit	DWORD_TO_INT
	DINT		Explicit	DWORD_TO_DINT
	REAL	The conversion to REAL accur with the lost of numerical resolution (only to values with modules greater than 16777215)	Explicit	DWORD_TO_REAL
USINT	BOOL	If the source is zero the target will be FALSE, on the contrary case will be TRUE.	Explicit	USINT_TO_BOOL
	BYTE	The target receives the binary form of the source.	Explicit	USINT_TO_BYTE
	WORD		Explicit	USINT_TO_WORD
	DWORD		Explicit	USINT_TO_DWORD
	INT	Convert the absolute value of the source to the target format. The values will remain the same.	Implicit	USINT_TO_INT
	DINT		Implicit	USINT_TO_DINT
	REAL		Explicit	USINT_TO_REAL

Source	Target	Comment	Conversion	Function
INT	BOOL	If the source is zero the target will be FALSE, on the contrary case will be TRUE.	Explicit	INT_TO_BOOL
	BYTE	The target will receive the low byte of the source.	Explicit	INT_TO_BYTE
	USINT		Explicit	INT_TO_USINT
	WORD	The target receives the binary form of the source.	Explicit	INT_TO_WORD
	DWORD	The target receives the binary form of the source. If the source is a negative number the word more significative will receive the value 0xFFFF.	Explicit	INT_TO_DWORD
	DINT	Convert the absolute value of the source to the target format. The values will remain the same.	Implicit	INT_TO_DINT
	REAL		Implicit	INT_TO_REAL

Source	Target	Comment	Conversion	Function
DINT	BOOL	If the source is zero, the target will be FALSE, on the contrary case will be TRUE.	Explicit	DINT_TO_BOOL
	BYTE	The target will receive the low byte of the source.	Explicit	DINT_TO_BYTE
	USINT		Explicit	DINT_TO_USINT
	WORD	The target receives the binary form of the source low word.	Explicit	DINT_TO_WORD
	INT		Explicit	DINT_TO_INT
	DWORD	The target receives the source binary form.	Explicit	DINT_TO_DWORD
	REAL	The target will receive the integer source value. The conversion to REAL occur with the lost of the numerical resolution to values with module greater than 16777215.	Explicit	DINT_TO_REAL
REAL	BOOL	If the source is zero, the target will be FALSE, on the contrary case will be TRUE.	Explicit	REAL_TO_BOOL
	WORD	The source is first converted to DINT. See DINT conversion to the other types.	Explicit	REAL _TO_WORD
	INT	Convert the absolute value of the source to the target. Values out of the numerical band will be saturated on the numerical limits of INT (-32768 to 32767).	Explicit	REAL _TO_INT
	DINT	The target will receive the source integer value. Values out of the numerical range will be sautéed on the DINT numerical limits (-2147483648 to 2147483647).	Explicit	REAL _TO_DINT

Table 4-6. Operations on type conversion

Parallel to those conversion operations there are functions able to convert any type of input to a specific type. Those functions of type conversion use the format:

```
ANY_TO_<target type>( <value to convert> )
```

WARNING:
The conversion of variables type REAL to any variable type ANY_INT using the functions of conversion type, will always be passed the integer part of the REAL variable. Those functions do not carry out rounding.

Variables

A variable is a memory area that stores a type of language data. The data are defined as Basic Types of Data. All the variables must be declared before being used.

Declaring de Variables

Any variable declaration must be done between the words VAR and END_VAR. Each declaration can have many variables, separated by comma. In this case, all will be of a same type.

```
VAR
       <Var1>, <Var2>, ... , <VarM> : <tipo1>;
       <VarN>, <VarO>, ... , <VarZ> : <tipoN>;
END_VAR
```

The variables are always initiated automatically with the standard value, according to the following table. Only the variable placed in PLC operands are not initiated automatically.

Data Type	Default start value
BOOL	FALSE or 0
USINT, INT, DINT	0
BYTE, WORD, DWORD	0
REAL	0.0

Table 4-7. Start value of the variables

Only-Reading Variables

A variable declared as only reading only accepts value attribution in its declaration. Any other attribution during the code will cause error in verifying time.

The declaration of only-reading variables, also known as constant variables, is done through the clause CONSTANT, according to what is shown bellow:

```
VAR CONSTANT
       <Var1>, <Var2>, ... , <VarM> : <type1>;
       <VarN>, <VarP>, ... , <VarZ> : <typeN>;
END_VAR
```

The application of those variables occurs when it is necessary to map a variable of the PLC inside the ST module and that cannot have writing operations on them. Other application occurs when it is desired to substitute symbols for numerical constants and group in an only place the values associated to them.

The following example shows a simple application of variables type CONSTANT:

```
VAR CONSTANT
       INI : INT := 1;
       END : INT := 100;
END_VAR

VAR
       VECTOR                : ARRAY[1..100] OF INT;
       LEVEL AT %F0030       : REAL;
       I                     : INT;
END_VAR

FOR I := INI TO END DO
       VECTOR[ I ] := REAL TO INT( LEVEL * 100);
END_FOR;
```

Declaring Vectors

Vectors are declared using the clause ARRAY. The inferior and superior limits can be specified freely. That is, it is possible to define a vector of 10 elements, being the first element accessed by the index 14 and the last one by 23, for example. Those vectors can be indexed through any expression of type USINT, INT or DINT, since it is positive. The maximum number of elements is limited only by the size of the data area (3kbytes).

```
VAR
       <vetor1> : ARRAY [ <inferior_limit> .. <superior_limit > ] OF <type>;
       <vetor2> : ARRAY [ <inferior_limit> .. <superior_limit> ] OF <type>;

END_VAR
```

A declaration of vectors where the superior limit is less than the inferior limit or where the inferior limit is less than zero will cause a verification error.

Starting the Variables

Variables can be started with a value different from the standard, placing this value after the type as is following shown:

```
VAR
       <name of the variable > : <type> := <start value >;
END_VAR
```

A vector can also be started on the declaration. In this case, the values are written in sequence and must be separated by comma.

```
VAR
       Vector : ARRAY [1..10] OF INT := [ 11, 12, 13, 0, 0, 0, 0, 18, 19, 20 ];
END_VAR
```

It is equivalent to:

```
Vector[ 1 ] := 11;
Vector[ 2 ] := 12;
Vector[ 3 ] := 13;
Vector[ 4 ] := 0;
Vector[ 5 ] := 0;
Vector[ 6 ] := 0;
Vector[ 7 ] := 0;
Vector[ 8 ] := 18;
Vector[ 9 ] := 19;
Vector[ 10 ] := 20;
```

Mapping Variables in Simple Operands and Table

All the variables are placed by the verifier in a data area reserved to the use of modules P and F. As this memory area is destroyed in the end of the module call all the variable values are also destroyed.

The mapping of variables in operands %M allows keeping the variable value between the module calls. It also allows accessing the operands inside the PLC used by other modules. It is possible map operands %E, %S, %A, %M, %F, %I, %TM, %TF e %TI.

All the mapping is done in the variables declaration with the clause **AT**. As it MAPEA a global address for all the programming modules of the PLC, this operation has the following restrictions:

- It can only be used inside VAR.
- Declared variable with operands mapping are not automatically started with its standard value, but can be started explicitly.
- The use of the word CONSTANT is allowed, indicates that the variable will not be modified during the program.
- The signal “%” is mandatory before the operand.

Bellow, the declaration syntax:

```
VAR
       <variable> AT <operand_CP> : <type> ;
END_VAR
```

The types allowed on the declaration must be compatible with the operands. The following table shows the possible associations among operands and types:

Operand	Allowed Types
%M.x, %A.x, %E.x and %S.x	BOOL
%A, %E and %S	USINT or BYTE
%M or %TM	INT, or WORD
%I or %TI	DINT, or DWORD
%F or %TF	REAL

Table 4-8. Related types with operands on PLC

It is also possible to associate operand bit to a variable type BOOL. The allowed operands are: %E, %S, %A e %M. In this case, the mapping is done to an operand type BOOL.

Some examples of MAPEAMENTOS allowed are shown in the following figures:

Mapping to BOOL:

```
VAR
       HIGH_PRESSURE  AT %M0006.6 : BOOL;
       START_ECR      AT %A0006.6 : BOOL := FALSE;
       INPUT          AT %E0010.5 : BOOL;
       OUTPUT         AT %S0008.7 : BOOL;
END_VAR
```

Mapping to INT and WORD:

```
VAR
       LEVEL           AT %M0123:     INT;
       TEMPERATURE     AT %TM010[15]: INT;
       ALARMS          AT %M1432:     WORD;
END_VAR
```

All the operands MAPEAMENTOS are verified during the module execution so that it is possible to be sure that this operand was declared on C module. In case it not exists, an error in execution time occurs, according the description of the Table 5-2.

Mapping Vectors in Simple Operands and Table

It is also possible to map vectors to PLC operands using the clauses AT and ARRAY. Blocks of simple operands can be mapped, table operands or parts of table operands.

As in the mapping of variables, in the vectors mapping the vector type must be compatible with the operand type according to the table 4-5.

Bellow some examples of different mapping of vectors in operands.

```
VAR
       (*Declaring a vector of 150 positions on the operands block of operands M1000 to
M1149*)
       ALARMS AT %M1000 : ARRAY[ 1..150 ] OF WORD;

       (*Declaring a vector of 100 positions on the operands block
        %TM200[000] to %TM200[099]. It is obligatory to define the first position of
the table. *)
       PRESSURES AT %TM200[000] : ARRAY[ 1..100 ] OF INT;
```

```
        (*Declaring a vector of 50 positions on the operands block
        %TM030[050] to %TM030[099]. *)
        SILOS AT %TM030[050] : ARRAY[ 1..50 ] OF INT;
END_VAR
```

In operands mapping, every block must be declared on module C of the CP. A mapped vector in table operand must be totally in the table operand mapped. In case any of these conditions is false, an error in execution time occurs.

Mapped vectors in operands cannot be BOOL type.

Functions

A function is a routine that must be executed several times. The main use of the functions is allow a better modularity to a program. A function is declared in the following manner:

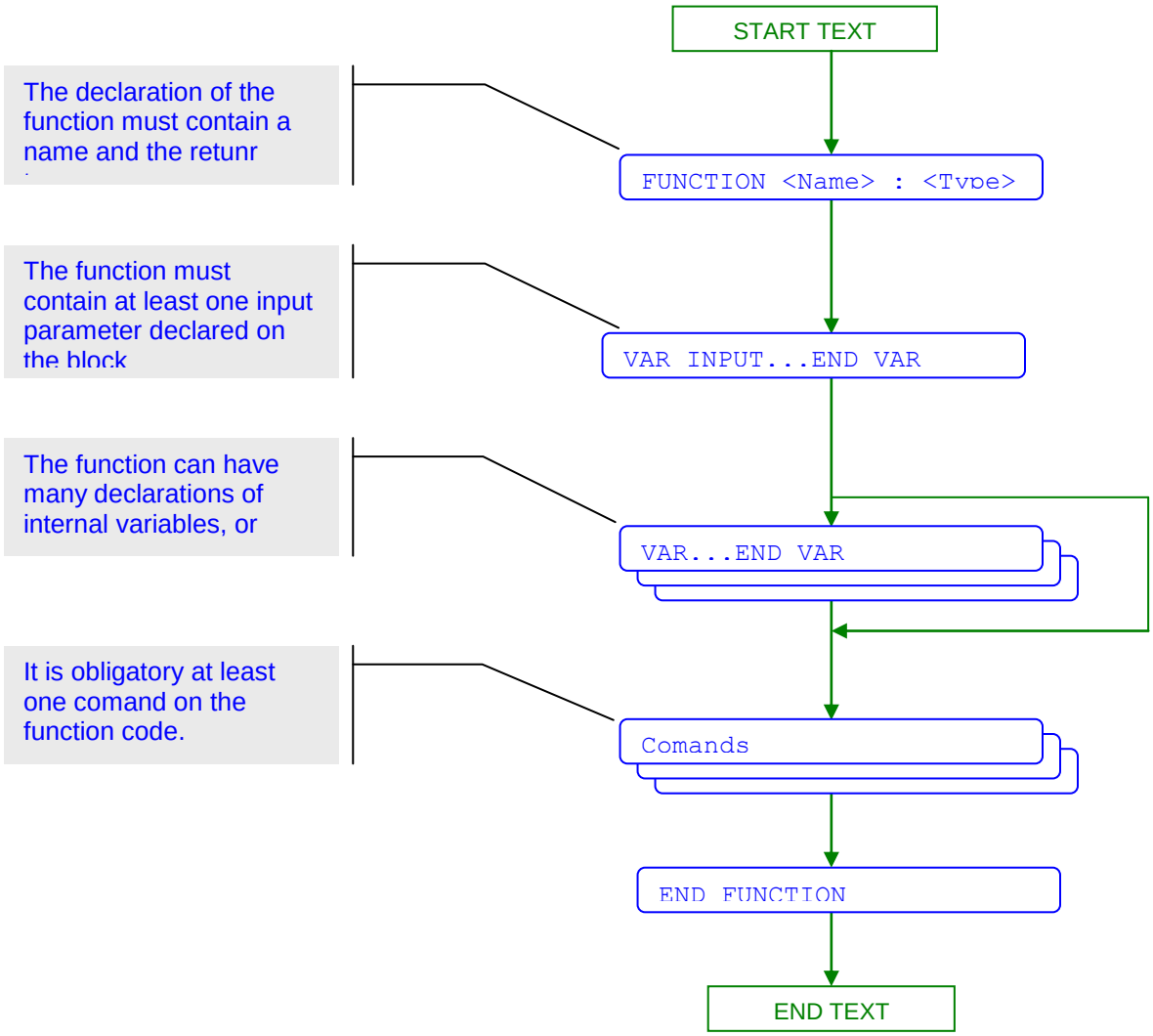

Figure 4-4. Text structure to define a function

Before the function returns to the routine which called it, a return value must be configured. This operation can be done attributing a value to the function name. The return value can be attributed in several parts of the commands block and can be done more than once.

```
FUNCTION FUNCTION1 : INT
        VAR_INPUT
                A : INT;
```

```
        END_VAR

        FUNCTION1 := A * 2;

END_FUNCTION
```

The input parameters must be specified between the words VAR_INPUT and END_VAR. The order in which the parameters are specified determines the order in which they must pass to this function.

The input parameters cannot be mapped in PLC operands, or be defined as vectors.

A function can call other functions defined by the user. However, the call of the same function is not allowed and an error in verification time occurs. The limit of calls nested function is 16 calls, that is, from the main program, can only be called 16 functions according to the following picture:

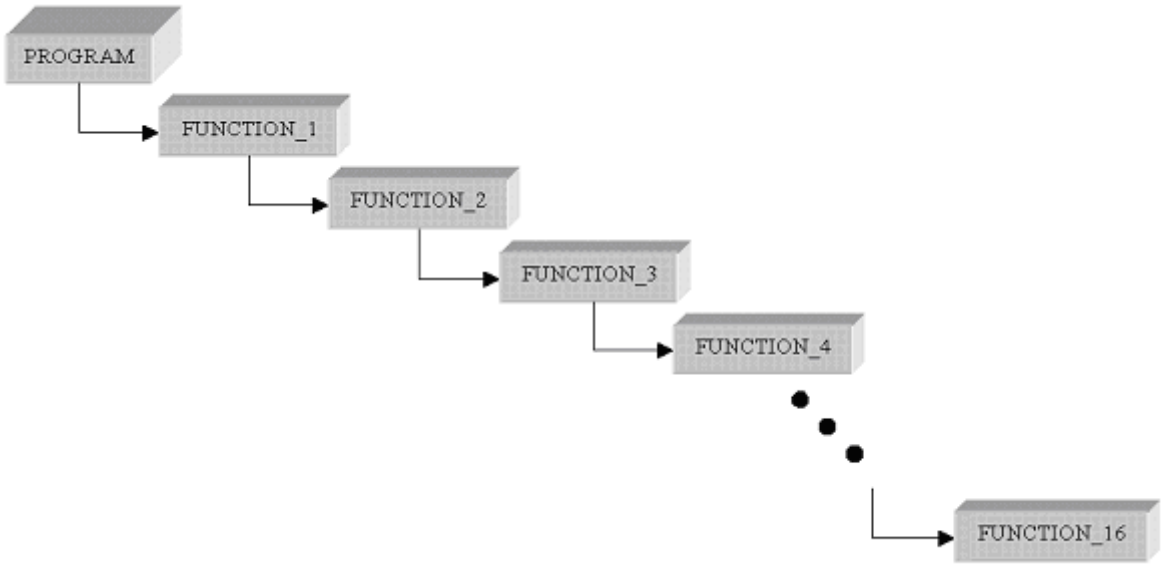

Figure 4-5. Calling function limits

Program

Program is the code routine where the execution of module starts. A program can access all the global variable and functions defined on the module. It is declared as in the following figure:

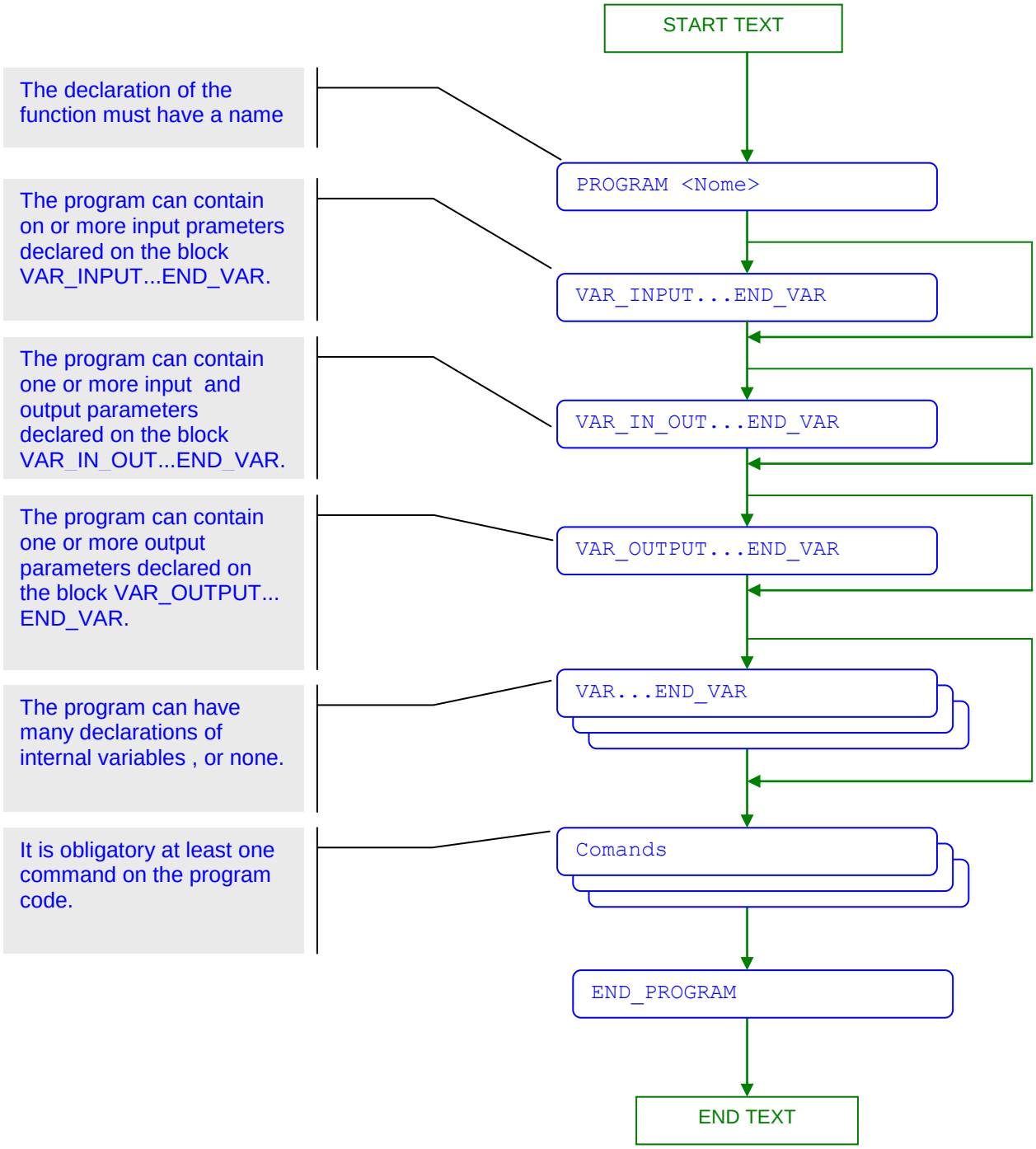

Figure 4-6. Text structure to define a function

Parameters Passing

Whenever a module F is being programmed, the passing of input and output parameters is possible. They are passed through the ladder instruction and are limited to the amount of parameters allowed by the instruction, being a total of 10 input parameters to input and/or output and 10 more output parameters.

Declaration	Description	CHF – Calling F Modules
VAR_INPUT	Input parameters. Read only.	The first n input parameters.
VAR_IN_OUT	Input and Output parameters. Read and Write.	The last n Input parameters.
VAR_OUTPUT	Output Parameters. Write only.	The output parameters.

Table 4-9. Parameters types

The following figure shows the association between the input and output variables of a CHF with the types of variable used in the ST module.

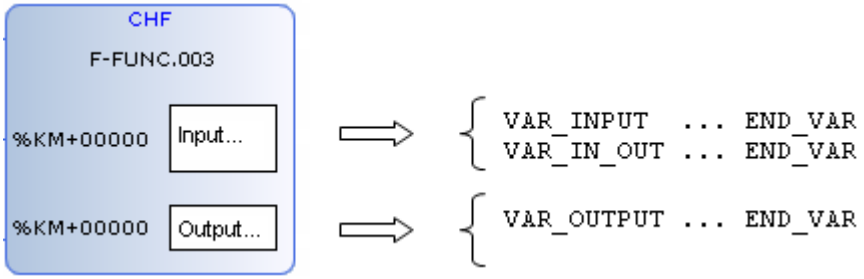

Figure 4-7. Relations between inputs and outputs of CHF with ST module operands

All the parameters are passed by value. They can be table operands, blocks of simple operands. The consistence rules to the passing of parameters are shown bellow, and any violation will cause an error in execution time:

- The parameters must be passed to equivalent types according to the table 4-8;
- Constant operands, %KM, %KI and %KF, can only be related to VAR_INPUT and only to simple variable, they cannot be vector
- Can not be passed part of the operand, like bit, nible, byte or word.
- Table operands can be related only to vector;
- A VECTOR must be related to a table operand or to a block of simple operands, where the first VECTOR position is the operand passed on the CHF;
- All the operands passed by parameters must be declared;
- All the operands related to vector must be declared, and it includes the complete vector size.

Table 4-9 represents the input and output operands declaration to the following declaration of variables in a ST module.

```
PROGRAM TEST
VAR_INPUT
        OP_INPUT0:          ARRAY[ 1..100 ] OF INT;
        OP_INPUT1:          INT;
        OP_INPUT2:          INT;
END_VAR

VAR_IN_OUT
        OP_IN_OUT0:         REAL;
        OP_IN_OUT1:         REAL;
        OP_IN_OUT2:         REAL;
END_VAR

VAR_OUTPUT
        OP_OUTPUT0:         ARRAY[ 1..10 ] OF INT;
        OP_OUTPUT1:         INT;
        OP_OUTPUT2:         INT;
        OP_OUTPUT3:         REAL;
        OP_OUTPUT4:         REAL;
        OP_OUTPUT5:         REAL;
END_VAR
;
END_PROGRAM
```

Parameters Passing to F Module

A module F programmed in ST language allows the passing of parameters type VAR_INPUT, VAR_IN_OUT e VAR_OUT. The variables type VAR_INPUT e VAR_IN_OUT are declared on "Input…", while the variables type VAR_OUT are declared on "Output…", of the CHF. The distinction between input and output variables is made at the moment of the variables declaration on ST module. The following example describes the variable declaration of a module F with three input variables, two input and output variables and two output variables. In this case five variable must be declared on "Input" and two on "Output" on the CHF.

```
PROGRAM <program name>
VAR_INPUT
        VAR_0:  INT;
        VAR_1:  INT;
        VAR_2:  INT;
END_VAR

VAR_IN_OUT
        VAR_IN_OUT_0:       REAL;
        VAR_IN_OUT_1:       REAL;
END_VAR

VAR_OUTPUT
        VAR_OUT_0:          INT;
        VAR_OUT_1:          INT;
END_VAR

        <program body>

END_PROGRAM
```

At the end of the execution of the ST module the operands type VAR_IN_OUT are copied to their respective source operands.

ATTENTION: Only parameters type VAR_INPUT can be associated to constant operands (%KM, %KI, %KF) through the input parameters of the CHF.

Signs of Module Input and Output

Inside the PROGRAM scope, there are up to six variables type BOOL pre-declared referring to the input and output signs of the CHF and CHP instructions. The following figure presents this association:

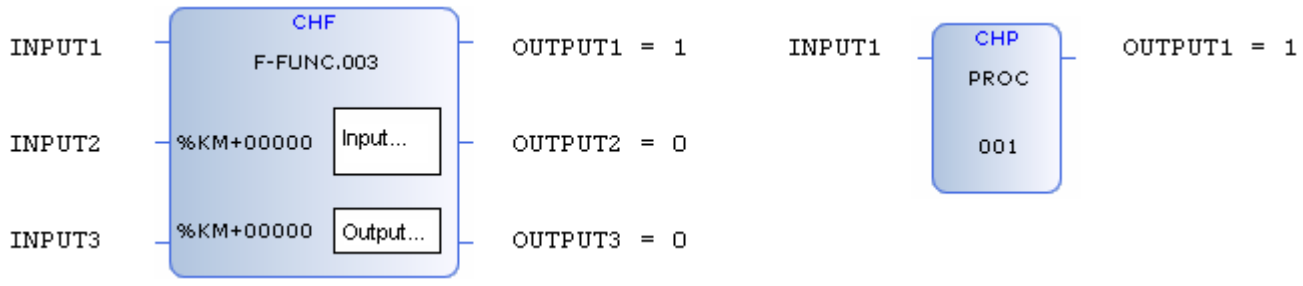

Figure 4-8. Modules outputs and inputs signals

The variable INPUT1, INPUT2 and INPUT3 are only for reading. The value of each variable corresponds to the value of the input instruction CHF or CHP on ladder.

ATTENTION: For the execution of the program it is necessary that the first input of the CHF and CHP instruction is enabled.

The variable OUTPU1, OUTPUT2 and OUTPUT3 are only writing. The variable OUPUT1 is started with TRUE while the others are started with FALSE. In case there is an error in execution time, the variable OUTPUT1 is placed in FALSE, independently of what the user have already written on it.

```
PROGRAM SUM
VAR_INPUT
        I ,J : INT;
END_VAR
VAR
        SUM : INT;
END_VAR
        (* Input 2 of the function defines if the sum must be times 2 or times 4*)
        IF INPUT2 THEN
                SUM := (I + J) * 2;
        ELSE
                SUM := (I + J) * 4;
        END_IF;

        (* The output of the function will be unpowered if an overflow occur *)
        IF INTERNAL_OVERFLOW THEN
                OUTPUT1 := FALSE;
        ELSE
                OUTPUT1 := TRUE;
        END_IF;

END_PROGRAM
```

Internal Variable of Control

INTERNAL_OVERFLOW

It indicates that an overflow or underflow in last arithmetic operation of kind addition, subtraction, multiplication, negation or in EXPT function.

```
PROGRAM SUM

VAR_INPUT
        I ,J, K : INT;
END_VAR

VAR
        TEMP : INT;
END_VAR

        TEMP := I + J;

        IF INTERNAL_OVERFLOW THEN
                OUTPUT1 := FALSE;
                RETURN;
        END_IF;

        TEMP := TEMP + K;

        IF INTERNAL_OVERFLOW THEN
                OUTPUT1 := FALSE;
        END_IF;

END_PROGRAM
```

When an overflow or underflow happen, the result of operation will be limited on type used in operation. For example, add two variables of type INT, whose values are 15_000 and 25_000, the result will be 32_767.

Scope and Life Time Rules

The names used to identify variables, functions and the program can be declared in global or local scope.

In the global scope, the name is visible to all the functions and to the program. All the functions and the program are declared on the global scope, but the variables, are only global the ones which are declared out of the functions and the program, in the beginning of the code.

The local scope is the scope inside the functions and the program. The declared variables inside this scope are visible only to the functions in which they were declared.

It is not allowed to declare the same name twice inside the same scope. However, when a local variable name coincides with a global variable name, the name declared in the global scope will always be used.

The lifetime of variables will depend on the local where they were declared. To variables inside the functions, the value of those variables is destroyed at the end of the function call. The variable values declared on the global scope are destroyed at the end of the module call, when the program returns to ladder. The variables mapped in operands, which keep its value between the modules call, are exceptions.

Commands

A program written in ST is composed by a sequence of "commands". The types of commands are:

- Commands of attribution
- Commands of function call
- Commands of program control
- Commands of selection
- Commands of repetition

Besides the commands, the verifier or the ST Language is able to evaluate mathematics expressions to value calculus.

Expressions

Expressions are used to calculate or evaluate values. An expression is composed by several operands and operators. They can be variable, literal or function call.

Operators can use one or two operands. When they use only one operator they are called "unary". In this case, they will always be located before the operand. When they use two operands, they are called binary. In this case, the operator must be between the operands.

Both operators used in binary operations, in most of the operations, must be the of the same type.

When operators of different types are used, a conversion function must be used, according to the description of the section **Types Conversion .**

Mathematics Operators

Those operators carry through mathematics operations between two operands. The operands can be any ANY_NUM, but they cannot be different among them. The mathematics operator always returns the same type of used operands.

Operator	Description	Application	Change INTERNAL_OVERFLOW
+	Addition	ANY_NUM **+** ANY_NUM	Yes
-	Subtraction	ANY_NUM **–** ANY_NUM	Yes
-	Denied	- REAL - DINT - INT	Yes
*****	Multiplication	ANY_NUM ***** ANY_NUM	Yes
/	Division	ANY_NUM **/** ANY_NUM	No
MOD	Modulo of division	ANY_INT MOD ANY_INT	No

Table 4-10. Mathematics operands basic

The operation `Var1 MOD 0` will return 0 (zero). This operation will not create error of division by zero.

Relation Operators

Relation operands execute a comparison between two numerical types according to the description on the Table 4-11. The operands must be of the same type and the operation returns always in a type BOOL.

Operator	Description	Application
<	Less than	ANY_NUM **<** ANY_NUM
>	Grater than	ANY_NUM **>** ANY_NUM
<=	Lesser or equal	ANY_NUM **<=** ANY_NUM
>=	Greater or equal	ANY_NUM **>=** ANY_NUM
=	Equality	ANY **=** ANY
<>	Inequality	ANY **<>** ANY

Table 4-11. Related operator

Operators Logic and Bit-to-Bit

Those operators execute two different operations: Boolean logic and bit-to-bit logic. The operation selection is done according to the types of operands used.

Boolean logic operations are executed among operand type BOOL. The following table represents the result of an operation BOOL. The result will always be BOOL type.

Operand A	Operand B	AND, &	OR	XOR
FALSE	FALSE	FALSE	FALSE	FALSE
FALSE	TRUE	FALSE	TRUE	TRUE
TRUE	FALSE	FALSE	TRUE	TRUE
TRUE	TRUE	TRUE	TRUE	FALSE

Table 4-12. Logic operators

Logic operations bit-to-bit are executed when the operands are BYTE, WORD and DWORD, being the two operands of the same type. The bit-to-bit operation carries through a Boolean operation for each bit of the operands, according to the description on the Table 4-12. Those operations return to the same type of the operands used.

```
Inputs := 2#0011 1001;        (* Inputs, type BYTE *)

Filters  := 2#0001_1100;      (* Filters,  type BYTE *)

Alarms := Inputs AND Filters; (* Alarms,  type BYTE *)

                              (*Alarms is equal to 2#0001_1000 *)
```

Operator	Description	Application
AND, &	Operation “AND”	ANY_BIT **AND** ANY_BIT ANY_BIT **&** ANY_BIT
XOR	Operation “OR” exclusive boolean	ANY_BIT **XOR** ANY_BIT
OR	Operation “OR” boolean	ANY_BIT **OR** ANY_BIT
NOT	Boolean Complement	**NOT** ANY_BIT

Table 4-13. Operators bit-to-bit

Operators Precedence

The evaluation of the expression is done according to the preference of the operators, as shown on Table 4-14. Operators of bigger precedence are evaluated first. If the operators have the same precedence, the one which is more on the left will be evaluated first.

Precedence	Operator	Description
0 (greater)	(...)	Expression between parentheses
	function (...)	Function
1	-	Negation
	NOT	Complement
2	*	Multiplication
	/	Division
	MOD	Rest
3	+	Addition
	-	Subtraction
4	< , > , <= , >=	Comparing
5	=	Equality
	<>	Inequality
6	AND, &	Operation “AND” boolean
7	XOR	Operation “OR” exclusive boolean
8 (less)	OR	Operation “OR” boolean

Table 4-14. Operations precedence

Function Calls

A function can be called inside an expression. The value to be passed for each parameter is written inside the parenthesis and separated by comma. The order in which the value must be written must be the same as the parameters were declared in the function.

```
 (* Function calling: first form *)
function_name ( expression_1, expression_2, ... , expression_n )
```

In an expression that has more than one function, it is not possible to determine what function will be executed first.

EXPT Function

The EXPT function return the result of the operation (*base* ^{exponent}), where the *base* can be ANY_INT or REAL and the *exponent* can be USINT. The type of result will be the same type of the *base.* This function change the value of INTERNAL_OVERFLOW.

```
VAR
       base, result   : INT;
       exponent       : USINT;
END_VAR

base           :=     4;
exponent       :=     2;
result         :=     EXPT( base, exponent );

(* result is equal to 16 *)
```

Integer Constants

Integer constants can be used in operation with ANY_INT since the literal value does not pass the limit of the type of the other operand.

Band	Types compatible
0 to 255	USINT, INT, DINT, BYTE, WORD and DWORD
-1 to –32.768	INT and DINT
0 to 32.767	INT, DINT, WORD and DWORD
0 to 65.535	DINT, WORD and DWORD
-1 to -2.147.483.648	DINT
0 to 2.147.483.647	DINT and DWORD
0 to 4.294.967.296	DWORD

Table 4-15. Integer constants

The real numerical literal can only be used in operations with variables that are also type REAL.

Attribution Command

Attribution is used to write a given value in a variable.

```
<variable> := <expression>;
```

Command of Program control

Command RETURN

A function always returns to the routine which called it after the execution of the last affirmation. However, it is possible return in the middle of the code, through the use of the reserved word **RETURN**.

In case the word RETURN is used in the main program, the program will be stopped, returning the execution control to ladder program.

Commands of selection

A command of selection executes one among several affirmation blocks. The chosen of the block is defined by an evaluation function expressed by non-terminal <*boolean_expression*>.There are two kinds of affirmation of selection: the command **IF** and the command **CASE**.

Command IF

IF command executes the affirmations after **THEN** if the test <*boolean_expression*> is true. Optionally, it is possible insert other test conditions with the **ELSIF** clause, being only the affirmation group where the first test is true executed.

Optionally, it is possible to specify an affirmation block to be executed, in case all the tests fail, through the clause **ELSE**.

```
IF <boolean_expression> THEN <commands>

{ ELSEIF <boolean_expression>  THEN <commands> }

[ ELSE <commands> ]

END_IF;
```

Example:

```
IF A = B THEN
       TEST := 10;

ELSEIF A = C THEN
       TEST := 11;

ELSEIF A = D THEN
       TEST := 12;
```

```
ELSEIF A = E THEN
       TEST := 13;

ELSE
       TEST := 0;

END_IF;
```

Command CASE

Command CASE also executes only one affirmation block. The block selection is done by comparing the integer value of <*integer_expression*> with the values written on <*cases*>.

Optionally it is possible to specify an affirmation block to be executed if all the tests fail through the clause **ELSE.**

```
CASE <integer_expression> OF

<cases> :  <commands>

{ <cases> :  <commands> }

[ ELSE <commands> ]

END_CASE;
```

<*cases*> indicates value list or sub-band separated by comma.

Example:

```
CASE Temperature OF

       0 :  Bomb1 := 10;

       1, 2, 5..10 :  Bomb1 := 10;

END_CASE;
```

The values tested must be compatibles with <*expression integer* >.

Commands de Repetition or Iteration

An iteration command executes repeatedly an affirmation block. The number of times it is executed depends on the iteration type that can be: command **WHILE**, command **FOR** and command **REPEAT.**

For all the commands, it is possible to stop the iteration loop prematurely through the **EXIT** command. This command can only be used inside the iteration loop. The use of the command EXIT out of an iteration affirmation will cause a verification error.

Command WHILE

The command WHILE executes a block of commands while the evaluation function <*boolean_expression*> is true. The command WHILE always test the evaluation function before executing the block. So, if in the first iteration the result of the test is false, the affirmation block will not be executed.

```
WHILE <boolean_expression> DO

       <commands>

END_WHILE;
```

Example:

```
WHILE I <= END_TABLE DO

       ACUMULATOR := ACUMULATOR + TABLE[ I ];
       I := I + 1;

END_WHILE;
```

Command REPEAT

The command REPEAT executes the commands block until the evaluation function <*boolean_expression*> is true. Differently of the command WHILE, the command REPEAT executes first the affirmation block and after tests the evaluation function. So, the affirmation block is executed at least once.

```
REPEAT

       <comands>

UNTIL <boolean_expression>  END_REPEAT;
```

Example:

```
REPEAT

       ACUMULATOR := ACUMULATOR + TABLE[ I ];
       I := I + 1;

UNTIL I > END_TABLE  END_REPEAT;
```

Command FOR

The command FOR allows executing an statement block repeatedly. The number of repetitions is controled by a <*control_variable*>. This variable must be type USINT or INT and cannot be an operand of the CPU of the programmable controller (%M, for example).

First, <*control_variable*> is initialized with the value of <*start_exp*>. In the beginning of each repetition, it is verified if the value of <*control_variable*> exceeds the value defined by <*end_exp*>. If it did not exceed, the statement block is executed. If not, the FOR command is stopped. In the end of the block execution, <*control_variable*> is incremented in 1, or by the value defined by <*inc_exp*><*inc_exp*>. The <*control_variable*> and the expressions <*start_exp*>, <*expr_inicial*> and <*end_exp*> must be data of the same type (USINT or INT).

```
FOR      <control_variable>      :=      <start_exp>      TO      <end_exp>      [      BY
<inc_exp><inc_exp><inc_exp><inc_exp> ] DO

       <commands>

END_FOR;
```

or

```
FOR <control_variable> := <expr_start> TO <expr_end> DO

       <commands>

END_FOR;
```

<*control_variable*>, inside the scope of the loop FOR, can be read but cannot be written.

During the iterations, the value <*end_exp*>evaluated in the beginning of FOR. That is, this expression is not evaluated again during the command.

Example:

```
FOR I := START_TABLE TO END_TABLE DO

       ACUMULATOR := ACUMULATOR + TABLE[ I ];
```

```
END_FOR;
```

5. Debug

Debug methods

This chapter describes as the debug of a module created in language ST, according to the orientations on Chapter 4 – Programming. Besides forcing and monitoring variables, there are other resources very useful when it is desired to debug an application, in ladder or ST language.

Afterwards, two methods are described. The first one uses execution in cycled mode of the programmable controller, and the second one uses status machines to implement the debug.

Cycled Mode

When in cycled mode, the programmable controller does not execute periodically the module E001, and remains waiting for commands of MasterTool programmer. To further details about mode cycled check the Programming Manual of MasterTool XE.

Using the depuration in cycled mode, a verification of the variable values used can be done between a cycle and other, verifying the variable values used and even force values to verify the behavior of the application that is being debugged.

Status Machines

This debug method consists in creating a defined sequence of actions related to an index or status. This manner, the code can be executed step by step, and each step can be a simple code line or a part of any code.

A simples implementation of this status machine can be obtained through the IF command, according to what is shown in the following example:

```
IF STATUS = 1 THEN
< commands block 1>
STATUS := 0;
END_IF;

IF STATUS = 2 THEN
< commands block 2>
STATUS := 0;
END_IF;
.
.
.
IF STATUS = n THEN
< commands block n>
STATUS := 0;
END_IF;
```

Each command block of the code presented is executed once, as the status index is zeroed at the end of the block execution. Incrementing the variable of the status, different parts of the code can be executed in a controlled manner. Between a status and another, different values can be forced to the verification of the code behavior and the values obtained in the involved variables, for example.

Errors in Verification Time

An useful information about the level of debug of source codes are the errors returned by the verification, upon checking the written code. Those errors can be typing problems, invalid

associations and inappropriate use of instructions, easing the process of development of the ST module.

The language ST verification can generate the following errors in a module:

Description	**Probably cause**
Invalid Char	The verifier did not recognize the typed text or char.
Invalid Symbol	Indicates that the lexical notation used is incorrect. The possible notations are listed below: Identifier: - Ended with "_" - Have two consecutive "_". Numeric Literal: - Ended with "_" - Have two consecutive "_". - Floating point with exponent without value. - Invalid digit to the numerical base.
Commentary not closed with *)	End of file was found before the end of comment.
Was expected <*token 1*> instead of <*text 2*>	Syntax Error. Probably was typed a wrong text, or a text is missing.
It was not expected <*text 1*> after <*text 2*>	Syntax Error. Probably was typed a wrong text, or a text is missing.
It was not expected <*text 1*>	Syntax Error. Probably was typed a wrong text, or a text is missing.
The Identifier <*name*> was already declared on this scope.	The Identifier was already declared. Use another name.
Identifier	Identifier
Variable <*name*> was not declared.	The variable was not declared. Declare the variable before use it.
Vector<*name*> was not declared.	The vector was not declared. Declare the vector before use it.
Function <*name*> was not declared.	The function was not declared. Declare the function before use it.
Was expected an integer expression.	The commands CASE expect integer expressions to test. The expression used is not integer.
Impossible to convert <type 1> to <type 2>	The conversion between the types is not allowed. Try using an explicit function conversion.
Is not possible execute the operation <operation> between the types <left type> and the <right type>.	The operation is not valid to the types. Use de conversion function to convert to correct type.
Is not possible execute the operation <operation> with <right type>.	The operation is not valid to the type. Use de conversion function to convert to correct type.
Command EXIT can not be executed out of a loop WHILE, FOR or REPEAT.	The command EXIT can not be executed out of a loop WHILE, FOR or REPEAT.
This code will never be executed.	The code was written after a command of RETURN or EXIT and will never be executed.
Recursive calling of the function <name>	A function can not be called recursively.
One part of the function <name> does not return a value.	Exist a part of the code that do not return a value through the command: Function := value;
Function <*name*> was called with more parameters than declared.	
Function <*name*> was called with less parameters than declared.	
Symbol <name> is not a function	Was expected that the symbol was a function.
Symbol <name> is not a vector	Was expected that the symbol was a vector.
Symbol <name> is not a variable	Was expected that the symbol was a variable.
Symbol <name> is not a constant	Was expected that the symbol was a constant.

Symbol <name> do not allow reading	
Symbol <name> do not allow writing	
Constant <value> is already used on other case.	The integer constant has already been used on other CASE command.
FOR control variable <name> can not be related to an operand on the PLC.	
FOR control variable <name> can not be written inside a FOR	
Incorrect number of elements to start the vector	
PLC operand <operand type> is incompatible with <variable type>	
<Type> can not be used as parameter of <name function or program>	The declared type can not be used as parameter.
The F module can not have more than 10 parameters VAR_INPUT and VAR_IN_OUT	The number of declared parameters in VAR_INPUT and VAR_IN_OUT can not be more than 10.
The F module can not have more than 10 parameters VAR_OUTPUT.	The number of declared parameters in VAR_OUTPUT can not be more than 10.
Procedure module do not allow parameters	
Function not allow parameters of the type VAR_IN_OUT or VAR_OUTPUT	Was declared one or more variables of the type VAR_IN_OUT or VAR_OUTPUT on the FUNCTION scope.
Value <value> out of the limits	
Minimal value greater than maximum value	
Operand <operand> is not valid.	The index used on operand name is out of limits.
The number of variables exceed the limit.	
CPU <CPU name> is not supported by ST	Only CPUs PO3x42, PO3x47, AL-2004, PX2004 and PX2014 are support by ST
Quantity the <type operand> operands temporary is not sufficient. Minimal <minimal quantity> operands.	Configure more temporary operands.
The data area exceeds limit of the CPU <CPU name> . Byte used <bytes>. Limit <limit of bytes>.	Use less variables.
Invalid size array. Maximus <quantity> elements.	
Fail on assembly module.	Error on assembly module. Please contact Altus support.

Table 5-1. Errors on verification time

Errors in Execution Time

Whenever the ST module executes an illegal operation, as for example a division by zero or even an access to operands non-declared, the occurrence of an error in execution time is defined. This type of information indicates that, although correct according to ST language, the module ST is receiving values or variable types for each it is not prepared. The operands are configured on the menu Modules/Operands/Diagnosis.

Five %M operands are used to indicate the reason for the error, according to the following table:

Operand	Description
%M+0	Line that the error occurred: Line = 0: shows that an error was found on the beginning of the module, before the first line of the code. Line = -32.768 or bit15 with 1: no error was found, the other operands are zeroed.
%M+1	Error code. See the following table.
%M+2	First error complement information.
%M+3	Second error complement information.
%M+4	Error complement information.

Table 5-2. Errors on execution time

The following table presents a detailed description of the possible errors in execution time:

Code	Description	Probably Cause	Comp 1	Comp 2	Comp 3	Correction
2000	Access to a not declared operand.	The operand was not declared on C module.	Operand Type: 0: %M 8: %E/S 9: %A 4: %F 1: %I	Operand Address	Not used	Declare the operand on C module.
2001	Table operand not declared	The table operand was not declared on C module.	Operand Type: 0: %M 4: %F 1: %I	Table Address	Not used	Declare the operand on C module.
2002	Table position not declared	The table was not declared on C module with the number of positions used by the program.	Operand Type: 0: %M 4: %F 1: %I	Table Address	Table Position	Declare the operand on C module.
2003	Incorrect input parameter.	The operand used as input parameter on CHF is not compatible with the declared type.	Number of the input parameter.	Not used	Not used	Correct the operand on the CHF instruction.
2004	Incorrect output parameter.	The parameter used as output parameter on CHF is not compatible with the declared type.	Number of the output parameter.	Not used	Not used	Correct the operand on the CHF instruction.
2005	Operand of the input parameter is not declared.	The operand used as input parameter on CHF is not declared on C module of the PLC. Or the operands of the vector are not declared on the C module of the PLC.	Number of the input parameter.	Not used	Not used	Declare the operand on the C module or correct the operand on CHF.
2006	Operand of the output parameter not declared.	The operand used as output parameter on CHF is not declared on C module of the PLC. Or the operands of the vector are not declared on the C module of the PLC.	Number of the output parameter.	Not used	Not used	Declare the operand on the C module or correct the operand on CHF.
2008	Invalid value to constant operand	The used constant value exceed the parameter size of CHF	Number of the input or output parameter.	Not used	Not used	Correct the used constant value.
2009	Incorrect Number of input parameters.	The quantity of input parameters declared on CHF is incorrect.	Not used	Not used	Not used	Verify the correct number of input parameters used on CHF instruction.

Code	Description	Probably Cause	Comp 1	Comp 2	Comp 3	Correction
2010	Incorrect Number of output parameters.	The quantity of output parameters declared on CHF is incorrect.	Not used	Not used	Not used	Verify the correct number of output parameters used on CHF instruction.
2011	Input parameter of array type with insufficient positions.	Table operands with insufficient positions.	Number of the input or output parameter.	Not used	Not used	Verify the positions correct number used on the respective input parameter.
2012	Output parameter of array type with insufficient positions.	Table operands with insufficient positions.	Number of the output parameter.	Not used	Not used	Verify the positions correct number used on the respective output parameter.
2015	Invalid vector index.	The index used to access the vector is less than its minor limit or is less than its major limit.	Not used	Not used	Not used	Verify the possible values that can be used as vector index on the program.
2020	Division by zero.	Occurred a division by zero.	Not used	Not used	Not used	Verify the possible values that can be used on division operation.
2030	Execution time exceeded.	The execution time allowed for the module was exceeded.	Not used	Not used	Not used	Probably a loop instruction as WHILE or REPEAT was executed to infinite.
2031	Execution inside E018	The module is being executed inside E018, what is prohibited	Not used	Not used	Not used	It is not possible to call the module ST from E018.
2032	Firmware version	The version of the firmware is lesser than expected by the function.	Minimal version of the firmware. (decimal)	Not used	Not used	The module generated on ST can not be executed in inferior versions of firmware.
2040	Module saved with error	The module was saved with verification errors generating a blank program.	Not used	Not used	Not used	The module was saved and uploaded to the PLC with verification errors.
2050	Calling limits exceeded.	The number of functions calling in sequence exceeded the limit.	Not used	Not used	Not used	See the limit of callings. See Software.

Table 5-3. Description of errors in execution time

6. Examples of Use

This chapter presents examples of programs written in ST.

Buffer of events

The module F-EVT.030 inserts an event in the event buffer implemented in the operand %TM0010. Each event operand occupies 3 positions in the buffer. The first position stores the minute value in the high byte and the second value in the low byte. The second position stores the time. The third position stores the event code.

```
(*
       Store the events in a TM. Each event is stored in 3 positions.
                     High byte             low byte
       pos 0         minute                second
       pos 1                               hour
       pos 2                               event

       The event code is used as a parameter of CHF.
*)

(* Global Variables ------------------------------------------------------------------ *)
(* Events Buffer *)
VAR
       BUFFER                  AT  %TM0010[000]  :ARRAY[ 1..120 ] OF INT;(* Events Buffer
*)
       BUFFER_IN              AT  %M0000   :INT;    (* Buffer input*)
       BUFFER_OUT             AT  %M0001   :INT;    (* Buffer output*)
       BUFFER_NUM_EVENTS      AT  %M0002   :INT;    (* Number of stored events*)
       BUFFER_OVERFLOW        AT  %A0001.0 :BOOL;   (* Overflow *)
END_VAR

(* Constants *)
VAR CONSTANT
       BUFFER_INF             : INT := 1;      (* First array index *)
       BUFFER_SUP             : INT := 120;    (* Last array index  *)
       BUFFER_LIMIT           : INT := 40;     (* Maximal number of elements on buffer *)
END_VAR


(* Functions-------------------------------------------------------------------------- *)
(*
       Function: InsertValue

       Insert a value on the next position of event buffer;
       Return TRUE if overflow;
 *)

FUNCTION INSERT_VALUE : BOOL
VAR_INPUT
       VALUE : INT;
END_VAR

       (* Insert the position value *)
       BUFFER[ BUFFER_IN ] := VALUE;

       (* Control the buffer limits *)
       IF BUFFER_IN = BUFFER_SUP    THEN
              BUFFER IN := BUFFER SUP;
       ELSE
              BUFFER_IN := BUFFER_IN + 1;
       END_IF;

       (* Control overflow *)
       IF BUFFER_NUM_EVENTS = BUFFER_LIMIT THEN
              INSERT_VALOR := TRUE;
       ELSE
              BUFFER NUM EVENTS := BUFFER NUM EVENTS + 1;
              INSERT_VALUE := FALSE;
       END_IF;
```

```
END_FUNCTION


(* Program ------------------------------------------------------------------------ *)
PROGRAM F_EVT_030
	(*CHF input parameters*)
	VAR_INPUT
		EVENT   : INT;  (* Parameter 1 of CHF*)
		HOUR    : INT;  (* Parameter 2 of CHF*)
		MINUTE  : INT;  (* Parameter 3 of CHF*)
		SECOND  : INT;  (* Parameter 4 of CHF*)
	END_VAR

	(* Insert event *)
	INSERT_VALUE( MINUTE*256 + SECOND );
	INSERT_VALUE( HOUR );
	INSERT_VALUE( EVENT );

END_PROGRAM
```

Conversion of Values

The module P-CONV.040 converts the table degree values Fahrenheit to Celsius, storing the values in another table.

```
(*
	Convert the temperatures from °F to °C
*)

(* Functions----------------------------------------------------------------------- *)
(*
	Function: Convert

	Execute the conversion from one unit to another. Retorns the converted value.
	Makes the operation using the REAL type to obtain precision.
 *)

FUNCTION CONVERT : INT
VAR_INPUT
	INPUT                : REAL;
	MAXIMUM_INPUT        : REAL;
	MINIMUM_INPUT        : REAL;
	MAXIMUM_OUTPUT       : REAL;
	MINIMUM_OUTPUT       : REAL;
END_VAR

	(* Normalize the input value *)
	CONVERT :=     REAL TO INT( INPUT / (MAXIMUM INPUT - MINIMUM INPUT) *
	               (MAXIMUM OUTPUT - MINIMUM OUTPUT) );

END_FUNCTION


(* Program ------------------------------------------------------------------------ *)
PROGRAM P_CONV_040

	(* Variables *)
	VAR
		TEMPERATURES_CELSIUS         AT %TM0010[ 0 ]: ARRAY[ 1..100 ] OF INT;
		TEMPERATURES_FAHRENHEIT      AT %TM0011[ 0 ]: ARRAY[ 1..100 ] OF INT;
		I : INT;
	END_VAR

	(* Constants *)
	VAR CONSTANT
		START   : INT := 1;
		END     : INT := 100;
	END_VAR

	(* Convert the temperatures from Celsius to Fahrenheit *)
	FOR I := START TO END DO
		TEMPERATURES FAHRENHEIT[ I ] :=
			CONVERT ( TEMPERATURES CELSIUS[ I ],
			0, 100,
```

```
                        32, 232 );
        END_FOR;

END_PROGRAM
```

7. Appendix

Keywords

Here are presented a relation of keyworks pertaining to ST language. Not all of them are currently used by ST Language, but they were saved to future implementations.

```
AT
ARRAY OF
CASE OF ELSE END_CASE
INPUT1 INPUT2 INPUT3
OUTPUT1 OUTPUT2 OUTPUT3
INTERNAL_*
EXIT
FALSE TRUE
FOR TO BY DO END_FOR
FUNCTION END_FUNCTION
IF THEN ELSIF ELSE END_IF
REPEAT UNTIL END_REPEAT
RETURN
CONSTANT
VAR END_VAR
VAR_INPUT END_VAR
WHILE DO END_WHILE
BOOL
SINT USINT INT UINT
REAL
AND OR XOR NOT MOD
*_TO__** (conversion types)
EN ENO
F_EDGE
FUNCTION_BLOCK
END_FUNCTION_BLOCK
PROGRAM WITH
PROGRAM END_PROGRAM
R_EDGE
READ_ONLY READ_WRITE
RESOURCE ON END_REPEAT
RETAIN
STRUCT END_STRUCT
TASK
TYPE END_TYPE
VAR_IN_OUT END_VAR
VAR_OUTPUT END_VAR
VAR_EXTERNAL END_VAR
VAR_ACESS END_VAR
WITH
BYTE WORD DWORD LWORD
DINT LINT UDINT U LINT
LREAL TRUNC
TIME DATE TIME_OF_DAY TOD
DATE_AND_TIME DT
ANY ANY_NUM ANY_REAL
 ANY_INT ANY_BIT STRING
 ANY_DATE
ABS SQRT LN LOG EXP SIN COS
TAN ASIN ACOS ATAN
SEL MAX MIM LIMIT MUX
LEFT RIGHT MID CONCAT INSERT
DELETE REPLACE LEN FIND
JMP CAL RET
ADD MUL DIV EXPT MOVE
SHL SHR ROR ROL
GT GE EQ LE LT NE
N R S L D P SD DS SL LD ST
ACTION END_ACTION
INITIAL_STEP END_STEP
STEP END_STEP
TRANSITION FROM TO
END_TRANSITION
```

8. Glossary

Active CPU	In a redundant system is the CPU that is controlling the system – reading the inputs, executing the application program and activating the outputs.
Jumpers	Small connector to shortcut pins located on a circuit board. Used to set addresses or configuration.
Algorithm	Finite and well defined sequence of instructions with the goal to solve problems
Altus Relay and Blocks Language	Set of rules, conventions and syntaxes used when building a application program to run in an Altus PLC.
Application Program	Program downloaded into the PLC and has the instructions that define how the machinery or process will work.
Arrestor	Lightning protection device using inert gases.
Assembly Language	Microprocessor programming language, it is also known as machine language
Backup CPU	In a redundant system, it is the CPU supervising the active CPU. It is not controlling the system, but is ready to take control if the main CPU fails.
Bit	Basic information unit, it may be at 1 or 0 logic level.
BT	Battery test.
Bus	Set of electrical signals that are part of a logic group with the function of transferring data and control between different elements of a subsystem
Byte	Information unit composed by eight bits.
C-Module	See Configuration Module.
Commercial Code	Product code formed by the letters PO and followed by four digits.
Commissioning	Final verification of a control system, when the application programs of all CPUs and remote stations are executed together, after been developed and verified individually.
Configuration Module	Also referred to as C-Module. Unique module in a remote application program that carries several needed parameters for its operation, such as the operands quantity and disposition of I/O modules in the bus
CPU	Central Processing Unit. It controls the data flow, interprets and executes the program instructions as well as monitors the system devices.
Diagnostic	Procedures to detect and isolate failures. It also relates to the data set used for such tasks, and serves for analysis and correction or problems.
E2PROM	Electrically Erasable Programmable Read-Only Memory. Non-volatile memory that may be electrically erased by the electronic circuit.
E-Module	See Execution Module
Encoder	Normally refers to position measurement transducer.
EPROM	Erasable Programmable Read Only Memory. Memory for read only that may be erased and programmed out of the circuit. The memory doesn't loose its contents when powered off.
ER	Acronym used on LEDs to indicate error
ESD	Electrostatic Discharge.
Execution Module	Application program modules. May be one of three types: E000, E001 and E018. The E000 module is executed just once upon system powering or when setting programming into execution mode. The E001 module has the main program that is executed cyclically, while the E018 module is activated by the time interruption.
Firmware	The operating system of a PLC. It controls the PLC basic functions and executes the application programs.
FLASH EPROM	Non volatile memory that may be electrically erased and programmed..
F-Module	See Function Module.
FMS	Fieldbus Message System.
Function Module	Application software module called from the main module (E-module) or from another function module or procedure module. It passes parameters and return values. Works as a subroutine.
Hardkey	Connector normally attached to the parallel port of a microcomputer to avoid the use of illegal software copies
Hardware	Physical equipment used to process data where normally programs (software) are executed
I/O	See Input/Output.
I/O Module	Hardware module that is part of the Input/Output (I/O) subsystem.
I/O Subsystem	Set of digital or analog I/O modules and interfaces of a PLC
IEC 61131	Generic international standard for operation and use of programmable controllers.
IEC Pub. 144 (1963)	International standard for protection of accidental access and sealing the equipment from water, dust and other foreign objects.
IEC-536-1976	International standard for electrical shock protection.

IEC-801-4	International standard for tests of immunity against interference by pulses burst
IEEE C37.90.1 (SWC)	SWC stands for Surge Withstand Capability. This is the international standard for oscillatory wave noises protection.
Input/Output	Also known as I/O. Data input or output devices in a system. In PLCs these are typically the digital or analog modules that monitor or actuate the devices controlled by the system.
Interface	Normally used to refer to a device that adapts electrically or logically the transferring of signals between two equipments.
Interruption	Priority event that temporarily halts the normal execution of a program. The interruptions are divided into two generic types: hardware and software. The former is caused by a signal coming from a peripheral, while the later is caused within a program
ISOL.	Acronym used to indicate isolation or isolated.
kbytes	Memory size unit. Represents 1024 bytes.
LED	Light Emitting Diode. Type of semiconductor diode that emits light when energized. It's used for visual feedback.
Logic	A graphic matrix in Altus Relay and Blocks Language where are inserted the relay diagram language instructions that are part of an application program are inserted. A set of sequentially organized logics makes up a program module.
MasterTool	The Altus WINDOWS® based programming software that allows application software development for PLCs from the Ponto, Grano, Piccolo, AL-2000, AL-3000 and Quarks series. Throughout this manual, this software is referred by its code or as MasterTool Programming.
Menu	Set of available options for a program, they may be selected by the user in order to activate or execute a specific task
Module (hardware)	Basic element of a system with very specific functionality. It's normally connected to the system by connectors and may be easily replaced.
Module (software)	Part of a program capable of performing a specific task. It may be executed independently or in conjunction with other modules through information sharing by parameters.
Module address:	Address used by the CPU in order to access a specific I/O module.
Nibble	Information unit composed of four bits.
Not-operant CPU	In a redundant system this is the CPU that is neither active nor backup. May not take control of the system.
Operands	Elements on which software instructions work. They may represent constants, variables or set of variables.
PA	See Jumpers.
PLC	See Programmable Controller.
P-Module	See Procedure Module.
Procedure Module	PLC application software module called from the main module (E-module) or from another procedure module or function module that does not have parameters.
PROFIBUS PA	Means PROFIBUS Process Automation.
Programmable Controller	Also know as PLC. Equipment controlling a system under the command of an application program. It is composed of a CPU, a power supply and I/O modules.
Programming Language	Set of rules, conventions and syntaxes utilized when writing a program.
RAM	Random Access Memory. Memory where all the addresses may be accessed directly and in random order at the same speed. It is volatile, in other words, its content is erased when powered off, unless there is a battery to keep its contents.
Redundant CPU	The other CPU in a redundant system. For instance, the redundant CPU of CPU2 is CPU1 and vice versa.
Redundant system	System with a backup or double elements to execute specific tasks. Such system may suffer certain failures without stopping the execution of its tasks.
Ripple	Oscillation present in continuous voltages.
RX	Acronym used to indicate serial reception.
Scan Cycle	A complete execution of the PLC application program.
Sockets	Part to plug in integrated circuits or other components, thus facilitating their substitution and maintenance.
Software	Computer programs, procedures and rules related to the operation of a data processing system
Supervisory Station	Equipment connected to a PLC network with the goal of monitoring and controlling the process variables
Tag	Name associated to an operand or to a logic that identifies its content.
Toggle	Element with two stable states that are switched at each activation.
Hot swap	Procedure of replacing modules in a system without powering it off. It is a normal procedure for I/O modules.
TX	Acronym used to indicate serial transmission.
Upload	Reading a program or configuration from the PLC.
Varistor	Protection device against voltage spikes.
Watchdog timer	Electronic circuit that checks the equipment operation integrity.
WD	Acronym for watchdog. See Watchdog timer
Word	Information unit composed by 16 bits.