Statement List programming

for Festo PLC

Order no.: 18352
Description: MANUAL AWL
Designation: E.HB-AWL-GB
Edition: 1/97
Author: R. Conde, Festo Corporation
Editor: S. Baerwald, YC-ECI

3rd edition

1. Introduction

Target Audience

This manual has been written for use by individuals who are familiar with the basic concepts of industrial controls.

The purpose of this document is to familiarize the reader with programming Festo Programmable Controllers using the Statement List Language (STL).

1. Introduction

Manual Organization

Content Organization
This manual is divided into several major sections:

Chapter 1 — **Introduction** 4
Provides a brief introduction into the organization and content of this document.

Chapter 2 — **FST Programming Environment** 7
Introduces the Festo FST family of programming software and defines some common terms that will be used in this manual.

Chapter 3 — **Using FST Software** 11
Provides an overview of the process required to create, edit, load and run Statement List (STL) programs in the FST programming environment.

Chapter 4 — **Operands of Festo PLC's** 15
Describes various addressable Operands (internal PLC elements) of Festo programmable controllers in summary format. The STL language operators are presented as well as the concept of Local and Global Operands.

Chapter 5 — **STL Program Structure** 23
Addresses the various elements and instructions of the STL language as well as factors influencing program flow.

Chapter 6 — **STL Instruction Summary** 35
A brief introduction of each STL instruction is provided in alphabetical format.

Chapter 7 — **STL Instruction Reference** 39
This section provides a detailed description of each STL command instruction including its purpose, the proper syntax and several examples of usage. Commands are listed in alphabetical order for quick access.

1. Introduction

Content Organization (continued)

Chapter 8 — **Accessing Digital Inputs and Outputs** 75
Provides in depth information of how to address digital PLC Inputs and Outputs.

Chapter 9 — **Using Timers** 81
Describes how to use timers in STL. Also details the various system elements that encompass timing functions.

Chapter 10 — **Using Counters** 89
This chapter shows how to implement counters using the STL language.

Chapter 11 — **Using Registers** 99
Explains the structure and uses of Registers using STL in Festo controllers.

Chapter 12 — **Flags and Flag Words** 103
Provides important information regarding the various uses and structure of Flags and Flag Words.

Chapter 13 — **Applying Specialized PLC functions** 109
Includes basic information regarding application of Analog I/O, Networking, Field Bus and Positioning functions.

Appendix A — **Festo Controller Operands** 117
A listing of the available operands for all current Festo programmable controllers in tabular format.

Appendix B — **Sample STL Programs** 121
Several sample control tasks are presented, along with sample STL language solutions.

Appendix C — **Multiple Programs, Multiprocessing and Multitasking** 137
Explains the meaning of these terms, how they are implemented and which controller models support these features.

Appendix D — **Understanding Binary numbers** 143
Offers a basic presentation of how to convert between binary and decimal numbers.

Index 149

1. Introduction

Physical Organization

Many sections of this manual are further divided to provide the following organizational structure:

BRIEF, where appropriate, is located at the beginning of each section and describes the key points covered in the section. Experienced programmers and those who have studied the section can refer to the Brief to get, in condensed form, the information they need. Novice programmers can use the Brief as an introduction and guide to the important ideas and concepts that will be covered in the section.

DETAILS is the heart of each section. It contains a thorough explanation of the topic, which may include theory, purpose and typical examples.

2. FST Programming Environment

2. FST Programming Environment

Contents

BRIEF 9

DETAILS 9
Languages 9
Organization and Definitions 9

2. FST Programming Environment

BRIEF

This section provides basic information regarding Festo FST (**F**esto **S**oftware **T**ools) programming software. FST provides a complete environment integrating programming and documentation as well as on-line facilities.

DETAILS

Languages

FST software is available for operation on IBM XT/AT compatible computers using the PC/MS-DOS operating system. FST software is available to support the following programming languages:

- Matrix
- Statement List
- Ladder Diagram
- BASIC

Additional programming languages are under development.

Please contact your local Festo office for further information regarding availability.

Organization and Definitions

Before beginning to describe the STL language itself, it is useful to provide a larger context in which the overall organization of Festo PLC's can be viewed.

In describing the form and function of the STL language, the following organization and definitions will be applied:

FST Project

An FST **project** includes **all of the CPU's** (1 or more) within a system which are connected by means of the CPU's primary bus. FST software organizes all of its activities at the **project** level. Larger control systems may consist of multiple FST projects connected together via a network.

If a control system includes a Festo Field Bus system, all of the Field Bus Slave Stations would generally belong to the FST Project which included the Field Bus Master.

2. FST Programming Environment

CPU

The next lower level of organization is the CPU. Depending upon the model, Festo PLC's may allow from 1 to 5 or more CPU's to be interconnected at the project level. The acronyms CPU (**C**entral **P**rocessing **U**nit) and CCU (**C**entral **C**ontrol **U**nit) will be used interchangeably in this document.

Program

Each CPU may contain one or more user application programs. The maximum number of programs which can be stored, as well as the number of programs that may be processed concurrently, varies according to the controller model.

This manual will concentrate on the structure and implementation at the Program level.

3. Using FST Software

3. Using FST Software

Contents

BRIEF 13

DETAILS 14
Preliminary steps 13
Creating a program 13
Program writing 14
Loading programs 14

3. Using FST Software

BRIEF

This chapter describes the organization and principle functions of FST software and how it is used with Festo programmable controllers. The reader is directed to consult the manual that accompanies each FST software product for more detailed information.

In particular, this section presents a summary of the steps required to begin a new STL program.

DETAILS

Preliminary Steps

1. Installation & configuration: After obtaining the required FST software package, it should be installed on your computer following the installation instructions provided.

2. FST Project: Select an existing or Create a new project name using the FST menu.

Creating a Program

3. Program Creation: The STL Editor is used to create new or modify existing STL programs. When you select the STL program editor, and if you are creating a new program, (vs. modifying an existing program); FST will prompt for information regarding the program to be created.

Depending upon the controller model being used, one or more of the following entries may be required:

FST Editor Prompt:	Definition:
CCU	allows specification of CPU# for FPC405 models
Prgm./Module [P/B]	Enter „P“ for program or „B“ for program module
Program No.	defines the number of the program or module to be created. The range varies by controller model.
Version No.	Multiple versions of the same program no. can be stored. Specify a single digit number 1-9.
Description	Enter an optional program description.

3. Using FST Software

Program Writing

4. Using the STL Editor: The STL editor allows off-line entry and modification of programs using program-defined function keys for ease of program entry and formatting. The off-line feature provides the ability to edit programs without being connected to the programmable controller.

Help is always available by pressing the F9 function key. The **F8 File Menu** provides several variations for saving your work. It is also possible to perform a **Syntax Check** of your program.

By selecting a syntax check, a program can be tested for proper command formation (syntax). Any discrepancies will be displayed and must be corrected before the program can be loaded into the controller.

Loading Programs

5. Transferring programs to the controller: After you have completed editing your program(s), they must be transferred (loaded) from the personal computer to the programmable controller.

FST software, in conjunction with the RS232 serial port of your personal computer is used to perform this transfer. Depending on the controller model being used, a special cable and/or adaptor may be required. Please refer to the FST product brochure, manual or your local Festo office for the proper parts for your configuration.

FST software provides the ability to **Load Programs** or **Load Projects**. Until you become familiar with using FST software, it is suggested that you select the **Load Project** option from the FST Menu. This selection will assure that all of the data required for proper controller operation will be transferred.

6. On-Line operation: The On-Line facility of FST allows monitoring of the programmable controller at any time.

This feature allows easy monitoring of all important controller information including I/O, Timers, Counters and Registers, etc. In the case of STL programs, debugging is enhanced as it is possible to check which program Step is being executed.

Some versions of FST also allow displaying an STL program in 'dynamic source' mode. This mode displays the program's source code (as created in the STL editor) as well as the Step number being processed and the Status of all single and multibit operands used in the program.

4. Operands of Festo PLC’s

4. Operands of Festo PLC's

Contents

BRIEF 17

DETAILS 17

Single vs Multibit Operands 17
Single bit Operands 18
Multibit Operands 19

Local vs Global Operands 20
Global Operands 20
Local Operands 20

Operators 21

4. Operands of Festo PLC's

BRIEF

This chapter will introduce the identifiers used with Festo programmable controllers to refer to various system elements (both hardware and software).

These system identifiers (e.g. Timers, Inputs, Outputs etc.) will be referred to as **Operands**. Operands are elements within the controller that can be interrogated or manipulated using program instructions and operators. The concept of **Local** and **Global** operands will also be discussed.

DETAILS

FST software allows programs to be written using both **Absolute operands** (e.g. T1 is the absolute operand for Timer number 1) as well as **Symbolic operands** (e.g. MOTOR could be assigned to Output 1.6). In order to provide the highest degree of clarity, this document will only use absolute operands.

Before proceeding with using the STL language, it is necessary to become familiar with the various operands of the controllers and how they are addressed using the STL language.

Depending on the controller model, there may be differences in the type and scope of operands that are available. The reader should refer to Appendix A and the appropriate controller manual for more information.

Single vs Multibit Operands

A distinction should be made between Single and Multibit operands. Single bit operands (**SBO**) can be evaluated as true/false in the conditional part of a program sentence and can be Set/Reset in the executive part of a program sentence. During interrogation and loading operations, SBO's are stored in the Single Bit Accumulator (**SBA**) of the CPU.

Multibit operands (**MBO**) can be tested for value (<,>, =, etc.), (range 0-255, 0-65535, +/- 32767 etc.) or compared to other multibit operands in the conditional part of a sentence. In the executive part of a program sentence, multibit operands can be loaded with a value, decremented and incremented or manipulated via a rich set of arithmetic and logic operators. During interrogation and loading operations, MBO's are loaded into the MultiBit Accumulator (**MBA**) of the CPU.

Complete information on the use of Single and Multibit operands is described later in this document.

4. Operands of Festo PLC's

The next section presents a short summary of the various Single and Multibit PLC operands available in Festo programmable controllers. A complete description, along with example usage, is presented later in this document.

Depending upon the specific operand, it may be possible to use the operand in either the Conditional part, the Executive part, or both parts of a program sentence.

Single Bit Operands

The following table provides general information regarding Single Bit Operands, how they are abbreviated in the STL language, as well as a brief example. The **part** column indicates whether the respective example is valid for the Conditional (c) or Executive (e) section of a sentence.

A detailed explanation of each Operand and STL instruction can be found later in this document.

Operand	STL Form	Syntax	Part	Typical Example
Input	I	In.n	c	IF I2.0
Output Output	O O	On.n On.n	c e	IF O2.6 SET O2.3
Flag Flag	F F	Fn.n Fn.n	c e	IF F7.16 (note: called 'internal coils' by some competitors) RESET F9.3
Counter Counter	C C	Cn Cn	c e	IF C3 SET C5
Timer Timer	T T	Tn Tn	c e	IF T7 SET T4
Program Program	P P	Pn Pn	c e	* IF P2 * SET P3
Processor Processor	Y Y	Yn Yn	c e	* IF Y2 * RESET Y1
Error Status	E	E	c	* IF E
Auto Restart	ARU	ARU	c	* IF ARU

NOTE: ***Operands which are marked by '*' may differ or not be available in all controller models.***

4. Operands of Festo PLC's

Multibit Operands

The following table provides general information regarding the use of typical Multibit Operands. Detailed information is provided later in this document.

Operand	STL Form	Syntax	Part	Typical Example
Input Word	IW	IWn	c	IF (IW3=V 255)
Output Word Output Word	OW OW	OWn OWn	c e	IF (OW2=V80) LOAD V128 TO OW3
Flag Word Flag Word	FW FW	FWn FWn	c e	IF (FW3=V220) LOAD V21000 TO FW1
Function Unit Function Unit	FU FU	FUn FUn	c e	IF (FU32=V16) LOAD FU34 TO R60
Timer Word Timer Word	TW TW	TWn TWn	c e	IF (TW2 < V2000) LOAD V1345 TO TW6
Timer Preselect Timer Preselect	TP TP	TPn TPn	c e	IF (TP0 < V20) * THEN LOAD V500 TO TP4
Counter Word Counter Word	CW CW	CWn CWn	c e	IF (CW3 <> V50) THEN INC CW5
Count. Preselect Count. Preselect	CP CP	CPn CPn	c e	IF (CP3 = V555) LOAD V67 TO CP5
Register Register	R R	Rn Rn	c e	IF (R60 = V21113) LOAD (R53 + R45) TO R32
Error Word Error Word	EW EW	EW EW	c e	IF (EW AND V15) LOAD V0 TO EW

NOTE: ***Operands which are marked by '*' may differ or not be available in all controller models.***

4. Operands of Festo PLC's

Local vs. Global Operands

Some controller models allow multiple CPU's within a system (see Appendices A & C). When such systems are constructed, some operands are designated as **local**, while others are **global**.

Global Operands

Global operands are parts of a system which can be accessed by **any program in any CPU**. Typical examples of global elements include Inputs, Outputs and Flags.

In order for such **global accesses** to be possible, global operands must be **unique** in their naming conventions.

Local Operands

Local operands are parts of a system which can only be accessed **by programs in a particular single CPU.** Generally these operands reside within the local CPU and do not have unique global names.

If the controller model being used does **not** allow inclusion of a **CPU number or module number** when referencing an operand, then the operand is typically classified as being local.

For example, if a system contained multiple CPU's, each CPU might have Timers 0-31 which are referenced as T0-T31 in the STL programs.

Further, we might have a program running in CPU 0 which referred to Timer 6 (T6) and also have a program in CPU 1 which referred to Timer 6 (T6). In this situation, our system actually contains **two (2) totally independent** timers, both of which are referenced as T6.... one in each CPU.

Although you should refer to the manual for the controller model being used, the following Operands are generally local:

- Registers
- Timers
- Counters
- Function Units
- Programs
- Processors

4. Operands of Festo PLC's

Operators

The STL language uses the following operators and notations to be used in the construction of sentences.

Symbol	Purpose
N	NOT (negation)
V V$ V%	VALUE assignment for Multibit operands (decimal) VALUE assignment for Multibit operands (hexadecimal) VALUE assignment for Multibit operands (binary)
+	Addition of Multibit operands and constants
-	Subtraction of Multibit operands and constants
*	Multiplication of Multibit operands and constants
/	Division of Multibit operands and constants
<	Multibit comparison...Less Than
>	Multibit comparison...Greater Than
=	Multibit comparison...Equal To
<>	Multibit comparison...Not Equal To
<=	Multibit comparison...Less Than or Equal To
>=	Multibit comparison...Greater Than or Equal To
( )	Opening/Closing parenthesis used to qualify or specify the Order of Precedence for Logic and Arithmetic operations.

4. Operands of Festo PLC’s

5. STL Program Structure

5. STL Program Structure

Contents

BRIEF 25

DETAILS 25

STL Element Hierarchy 26
Step Instruction 26
Sentences 26
Typical Sentences 27
Further Examples 27
Comparison to Ladder Diagram 28

Step Instruction 29

Execution rules 30

Influencing Program Flow 32
NOP Instruction 32
JuMP Instruction 33
OTHRW Instruction 34

5. STL Program Structure

BRIEF

This chapter presents the basic architecture of the STL language and introduces the major elements of the language.

Many sample program fragments are included to illustrate key points. Although some of these programs may use terms with which the reader is not yet familiar, the comments included should provide sufficient understanding.

The STL language allows the programmer to solve control tasks using simple English statements to describe the desired operation of the controller. The modular nature of the language allows the programmer to solve complex tasks in an efficient and self-documenting manner.

The STL language as described herein applies to the Festo FPC100B/AF, FPC405, FEC and IPC programmable controllers. The structure of the STL language remains consistent across all models.

Hardware dependent features which are only available in specific models will not be discussed in detail in this document. Additional information regarding such features can be found in the respective controller manuals.

Information contained in this publication reflects the STL language as implemented in FST Software Version 3.X.

DETAILS

Statement List programs are constructed using several important elements. Not all of the available elements are required, and the way in which the elements are combined greatly influences how the program will operate.

This section will introduce each of the elements and how they work together in a program. After this brief introduction, a more detailed presentation of each element will be provided.

5. STL Program Structure

STL Element Hierarchy

PROGRAM

STEP

SENTENCE

CONDITIONAL PART

EXECUTIVE PART

Step Instruction

Although the use of the keyword STEP is optional, most STL programs will use the STEP instruction. The STEP instruction is used to mark the beginning of a logical block of program code.

Each STL program may contain up to 255 discrete STEPS and each Step may contain one or more Sentences. Each Step may be assigned an optional label or name.

A Step label is only required if the respective Step will later be assigned as the destination of a Jump instruction. A more complete description of the STEP instruction is presented after the introduction of Sentences.

Sentences

The Sentence forms the most basic level of program organization. Each Sentence consists of a **Conditional Part** and an **Executive Part**.

The Conditional Part serves to list one or more conditions which are to be evaluated at run time as being either true or false. The Conditional part always begins with the **IF** keyword and continues with one or more statements that describe the conditions to be evaluated.

If the programmed conditions evaluate as **true**, then any instructions programmed in the Executive part of the sentence will be performed. The beginning of the Executive part is marked by the **THEN** keyword.

5. STL Program Structure

Typical Sentences

The following section presents several typical simple STL sentences without any use of the Step instruction.

IF			I1.0	If Input 1.0 is active
THEN	SET		O1.2	then switch on Output 1.2
IF		N	I2.0	If Input 2.0 is NOT active
THEN	SET		O2.3	then switch on Output 2.3
IF			I6.0	If Input 6.0 is active
	AND	N	I2.1	and Input 2.1 is not active
	AND		O3.1	and Output 3.1 is ON
THEN	RESET		O2.1	then turn off Output 2.1
	RESET		T6	and reset Timer 6

In the last sample sentence, the principle of **compound** conditions has been introduced. That is, **all** of the stated conditions in the current sentence must be true for the actions following the THEN keyword to be executed.

Further Examples

IF			I3.2	If Input 3.2 is Active
	OR	N	T6	or Timer 6 is NOT running
THEN	INC		CW1	then increment Counter 1
	SET		T4	and start Timer 4 with pre-existing parameters

This example shows the use of the **OR** structure within the conditional part of a sentence. That is, the sentence will evaluate as being true (and therefore Counter 1 would be incremented and Timer 4 started) if **either or both** of the stated conditions are true.

The next sentence introduces the use of parentheses within the conditional part of a sentence to influence the manner in which conditions are evaluated.

IF		(	I1.1		If Input 1.1 is Active AND
	AND		T4	)	Timer 4 is running
	OR	(	I1.3		OR if Input 1.3 is Active
	AND		I1.2	)	and Input 1.2 is Active

We have utilized the OR instruction to combine two compound conditions by means of the parenthesis operator.

5. STL Program Structure

The previous examples have just briefly introduced the use of sentences in the Statement List language. It is possible to create entire programs that consist only of multiple sentences without ever using the **STEP** instruction.

Programs constructed in this manner are often described as being parallel programs. These programs react much in the same manner as programs written in the Ladder Diagram language.

That is, without using the Step instruction, such programs would be processed in a 'scanning-like' manner. In order for such programs to be processed continuously, it is necessary to include the PSE instruction (see Chapters 7 & 8).

Comparison to Ladder Diagram

For those readers who are familiar with the Ladder Diagram PLC language, a comparison between an STL Sentence and a Ladder Diagram rung can be made.

For example, a Ladder Diagram rung to switch ON an Output whenever an Input is Active and switch OFF the Output whenever the Input is Inactive would appear as:

```
    I1.0                        O2.6
|----] [---------------------------(  )--------|
```

While the equivalent STL sentence would be:

IF		I1.0	If Input 1.0 is active
THEN	SET	O2.6	then switch on Output 2.6
	PSE		end of program
OTHRW	RESET	O2.6	else turn off Output 2.6
	PSE		end of program

You will notice that the previous example also introduced the OTHRW command. The STL language requires explicit instructions to alter the state of any operand (e.g. Output, Timer, Counter).

The PSE instruction is placed at the **end** of a parallel program section to cause the program to be executed continuously by returning to the first Sentence of the current Step or the first Sentence of the program if no Steps are used. See chapter 7.

5. STL Program Structure

Step Instruction

Programs that do not use the STEP instruction can be processed in a parallel (scanning) fashion. Although this type of program execution may be well suited for solving certain types of control tasks, the STL language provides the **STEP** instruction which allows programs to be divided into discrete sections (STEPS) which will be executed independently.

In its simplest form, a STEP includes at least one sentence and takes the form:

```
STEP    (label)
IF                  I1.0    If Input 1.0 is Active
THEN    SET         O2.4    then turn on Output 2.4 and
                            proceed to the next step
```

It is important to understand that the program will WAIT at this Step until the conditions are true at which time the actions will be performed and only then will the program proceed to the next Step.

The optional Step label is only required if a Step will be the target of a JUMP instruction. It should be noted that when FST software loads STL programs into the programmable controller, it assigns relative Step numbers to each program Step. These assigned step numbers are also reproduced in all program listings and can be quite helpful in monitoring program execution for On-Line debugging purposes.

Program Steps can, of course, include multiple sentences:

```
STEP
IF                  I2.2    If Input 2.2 is Active
THEN    SET         O4.4    Switch on Output 4.4

IF                  I1.6    If Input 1.6 is Active
THEN    RESET       O2.5    Switch off Output 2.5
        SET         O3.3    and Switch on Output 3.3
```

In the previous example, we have introduced the concept of multiple Sentences within a Single Step. When the program reaches this Step, it will process the first sentence (in this case, turning on Output 4.4 if Input 2.2 is active) and then move to the second sentence **regardless** of whether the Conditions in the first sentence were true.

When the **last** (in this case the second) sentence of a Step is processed, if the Conditional part is **true,** then the Executive part will be carried out **and** the program will proceed to the next Step.

If the Conditional part of the **last** sentence is **not true**, then the program will return to the **first** sentence of the Current Step.

5. STL Program Structure

Execution Rules

The following guidelines can be applied to determine how Steps and Sentences will be processed:

- If the Conditions of a sentence are met, then the programmed Actions are executed.
- If the Conditions of the **last** (or only) sentence within a Step are met, then the programmed Actions are executed **and** the program proceeds to the next Step.
- If the Conditions of a sentence are **not** met, then the program will move to the next sentence in the current Step.
- If the Conditions of the **last** (or only) sentence within a Step are **not** met, then the program will return to the **first sentence of the current Step**.

Note:
It is important to understand when constructing Programs or Steps that contain multiple Sentences that will be processed in a parallel (scanning) manner; that **every time the conditional part of a Sentence evaluates as true, the instructions programmed in the executive part will be performed.** This **must** be considered in order to avoid uncontrolled multiple executions of instructions such as SET TIMER or INC/DEC counter.

The STL language does not use 'edge triggering'...**conditions are evaluated for truth each time they are processed without regard as to their prior status.**

This situation is easily handled by either using Steps, Flags or other means of control. See Appendix B for examples.

5. STL Program Structure

Figure 5-1 illustrates the processing structure of an STL Step. By using various combinations of Steps containing single or multiple sentences, the STL language provides a wide range of facilities for solving even very complex tasks.

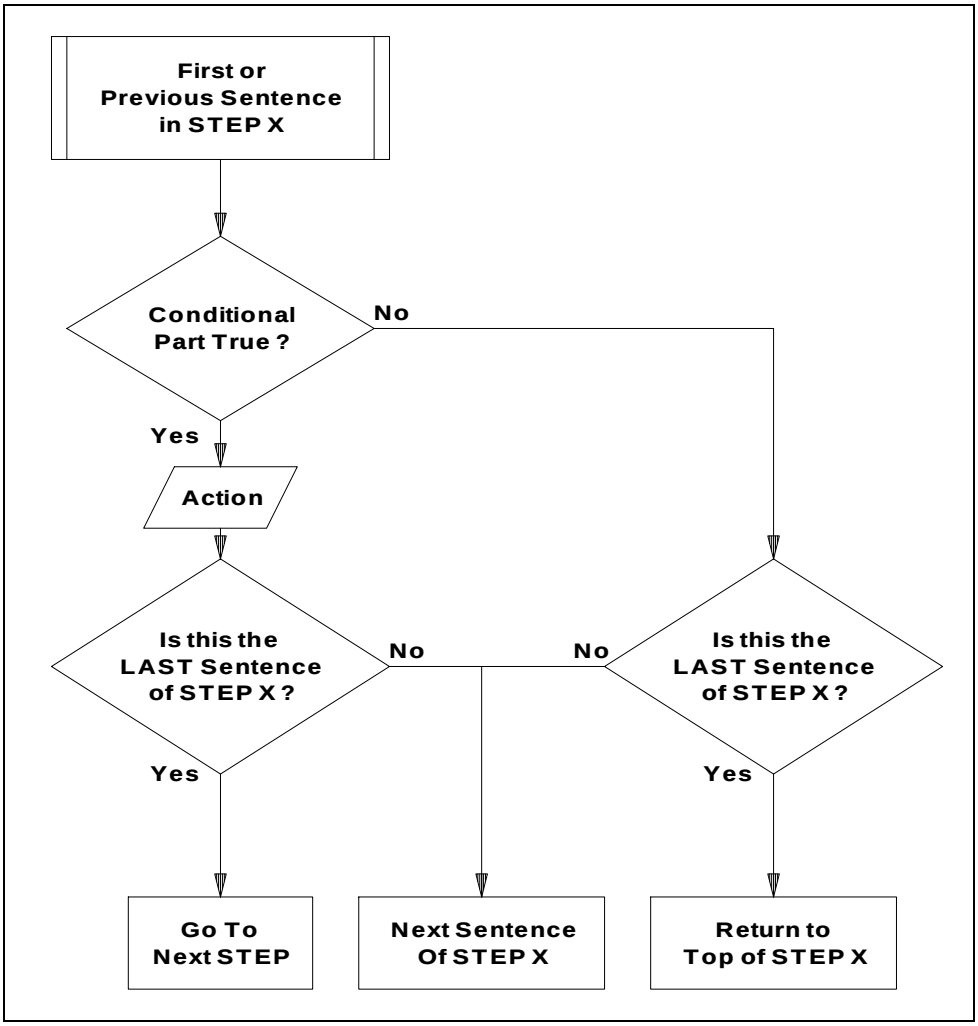


Figure 5-1 STL Basic Step Execution Rules

5. STL Program Structure

Influencing Program Flow

In addition to the control structures inherent within the Step instruction, several additional STL instructions are available which can be used to influence the execution criteria of program Steps and Sentences.

NOP Instruction

The NOP instruction may be used in either the Conditional or Executive part of a sentence.

When used in the Conditional part, the NOP instruction is **always** evaluated as true. The NOP instruction can be used to cause unconditional execution of a sentence.

IF		NOP	this is always true
THEN	SET	O1.0	so Output 1.0 will always be turned on when the program reaches here.

A typical use can be seen in the following example in which the author desired that when program execution reached Step 50 several conditions were to be checked and if they were true the appropriate actions were executed.

However, regardless of whether any or all of the conditions were true, after being checked **exactly** one time the program would turn on Output 3.6 and proceed to the next Step...because we have forced the **last** sentence to be true via the NOP instruction.

STEP 50				
IF			I1.0	If Input 1.0 is Active
THEN	SET		O2.2	then turn on Output 2.2
IF		N	I3.5	If Input 3.5 is NOT Active
	AND		I4.4	and Input 4.4 is Active
THEN	RESET		O1.2	then turn off Output 1.2
IF			T3	If Timer 3 is running
THEN	SET		F0.0	then set Flag 0.0
IF	NOP			in any case...we make certain that the LAST sentence will ALWAYS be true.
THEN	SET		O3.6	turn on Output 3.6 , **exit this Step and go to Next Step.**

5. STL Program Structure

The NOP instruction may also be used in the Executive part of a sentence. When used in the Executive part, a NOP is equivalent to 'do nothing'. It is often used when the program is to wait for certain conditions and then proceed to the next Step.

IF		I3.2	If Input 3.2 is Active
THEN	NOP		do nothing & go to the next Step.

JuMP Instruction

Another STL instruction which can be used to influence the flow of program execution is the JMP instruction.

The JMP instruction adds the ability of program branching to the STL language. By modifying the previous example it is possible to test the conditions of each sentence and if true perform the programmed action and then JuMP to a designated program Step.

STEP 50				
IF			I1.0	If Input 1.0 is Active
THEN	SET		O2.2	turn on Output 2.2
	JMP		70	and jump to Step label 70
	TO			
IF		N	I3.5	If Input 3.5 is NOT Active
	AND		I4.4	and Input 4.4 is Active
THEN	RESET		O1.2	turn off Output 1.2
	JMP		6	and jump to Step label 6
	TO			
IF			T3	if Timer 3 is running
THEN	SET		F0.0	then set Flag 0.0
IF			NOP	Always true, so...
THEN	SET		O3.6	turn on Output 3.6 and go to the next step.

It can be seen that not only have we altered the program flow, but in addition have established **priorities** between the sentences.

For example, sentences 2, 3 and 4 will only have the possibility to be processed if sentence 1 is false and therefore not executed; because if sentence 1 is executed, the program will Jump to Step 70 without ever processing any subsequent sentences in Step 50.

5. STL Program Structure

OTHRW Instruction

The OTHRW (otherwise) instruction can also be used to influence program flow. The OTHRW instruction is executed when the last encountered IF clause evaluated as **not true.**

IF		I2.0	If Input 2.0 is Active
THEN	SET	O3.3	turn on Output 3.3
OTHRW	SET	O4.5	otherwise turn on Output 4.5

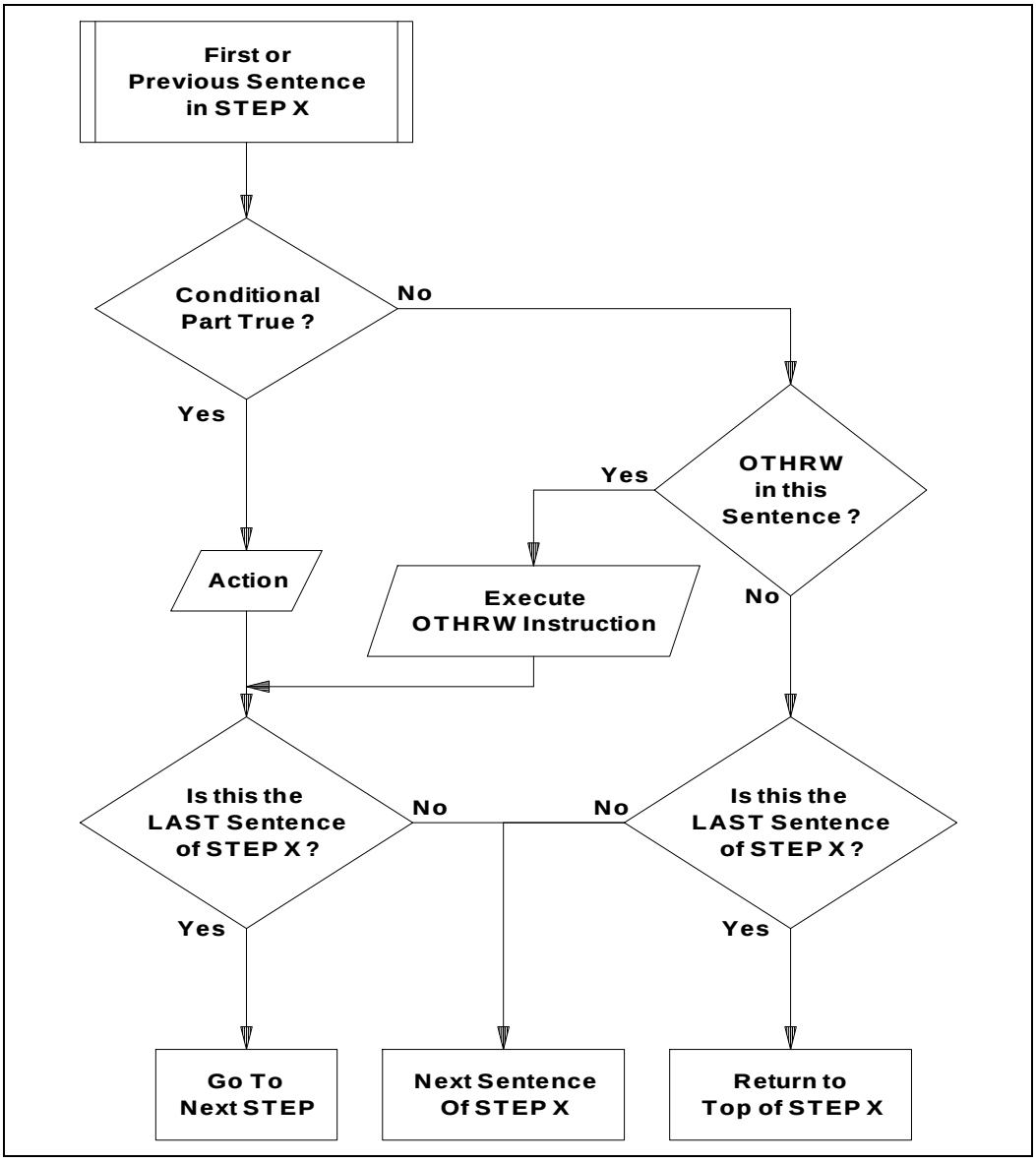


Figure 5-2 Step Execution with OTHRW instruction

6. STL Instruction Summary

6. STL Instruction Summary

The STL language provides the following instructions which allow both simple and complex control tasks to be solved quickly and easily. Detailed information and examples of usage appear in the next section.

Instruction	Purpose
AND	Performs a logical AND operation on single or multibit operands and constants.
BID	Converts the contents of the Multibit Accumulator from Binary to BCD format.
CFM n	Begin execution or Initialization of a Function Module.
CMP n	Begin execution of a Program Module.
CPL	Produces the two's compliment of the contents of the multibit accumulator.
DEC	Decrements a Multibit Operand/Accumulator
DEB	Converts the contents of the Multibit Accumulator from BCD to Binary format.
EXOR	Performs a logical EXOR operation on single or multibit operands and constants.
IF	Keyword marking the beginning of the Conditional part of a sentence
INC	Increments a Multibit Operand/Accumulator
INV	Produces the one's compliment of the contents of the multibit accumulator.
JMP TO (Step label)	Causes program to continue execution at the specified Step.
LOAD	Allows loading specified operands (single or multibit) and constants to either the single or multibit accumulator.
NOP	A special instruction which is **always** true in the Conditional Part of a sentence. In the Executive Part it is equivalent to '**do nothing**'.
OR	Performs a logical OR operation on single or multibit operands and constants.

6. STL Instruction Summary

Instruction	Purpose
OTHRW	Provides the ability to continue program execution if the Conditional Part of a sentence is **false**.
PSE	The PSE (**P**rogram **S**ection **E**nd) instruction.
RESET	The Reset instruction is used to change single bit operands to a logical '0' status.
ROL	Rotates Left all bits contained in the Multibit Accumulator by one position. The most significant bit is moved to the least significant bit.
ROR	Rotates Right all bits contained in the Multibit Accumulator by one position. The least significant bit is moved to the most significant bit.
SET	The Set instruction is used to change single bit operands to a logical '1' status.
SHIFT	Performs a Single Bit Swap between a Single Bit Operand and the Single Bit Accumulator.
SHL	Shifts Left all bits contained in the Multibit Accumulator by one position. The most significant bit is lost, and the least significant bit is filled with a zero (0).
SHR	Shifts Right all bits contained in the Multibit Accumulator by one position. The least significant bit is lost, and the most significant bit is filled with a zero (0).
SWAP	Exchanges the high and low order bytes of the Multibit Accumulator.
TO	Used with the LOAD instruction to specify a destination operand.
THEN	Keyword marking the beginning of the Executive part of a sentence.
WITH	Used to pass parameters with some CFM/CMP instructions. Also used to specify timer clock rates for some PLC models.

6. STL Instruction Summary

7. STL Instruction Reference

7. STL Instruction Reference

Contents

AND 41
BID 43
CFM 44
CMP 46
CPL 48
DEB 49
DEC 50
EXOR 51
INC 53
INV 54
JMP TO 55
LOAD...TO 58
NOP 61
OR 63
PSE 65
RESET 66
ROL 67
ROR 68
SET 69
SHIFT 70
SHL 71
SHR 73
SWAP 74

7. STL Instruction Reference

AND

Purpose

1. To combine two or more single or multibit operands in the Conditional part of a Sentence using the logical AND operation.

2. To perform logical AND'ing of two multibit operands or values in either the Conditional or Executive parts of a sentence.

Examples

Single Bit

IF		I1.1	If Input 1.1 is Active
	AND	T6	and Timer 6 is running
THEN	SET	O1.5	turn on Output 1.5

Multibit

The following illustrates the logical bit-wise AND operation applied to two 8 bit operands:

0	0	1	0	1	1	0	1	operand 1 = 45 decimal
1	1	1	0	1	1	0	0	AND operand 2 = 236 decimal
0	0	1	0	1	1	0	0	result = 44 decimal

The AND function can be used with Multibit operands in both the Conditional as well as the Executive parts of a Sentence.

When used in the Conditional part of a Sentence this function allows the result of a Logical AND function of two Multibit operands to be compared to a third Multibit operand or constant.

IF		(	R6		the contents of Register 6
	AND		R7	)	are AND'd to the contents of Register 7. Next the result is
		=	V34		compared to the constant 34 decimal. If equality is found,
THEN	...				any programmed actions will be performed.

7. STL Instruction Reference

It is important to understand that the above sentence is **not** to be interpreted as:

"If R6 =34 and R7=34 then ..."

This sentence in STL would be written as:

```
IF                  (   R6      )   if this is true...
                    =   V34     )
          AND
                    (   R7      )   if this is true...
                    =   V34     )
THEN      ...                       then...
```

The next example shows how to use the multibit performance of Festo controllers to Read an entire group (Word) of Inputs. Next the result is logically AND'ed with 15 decimal (00001111 binary). By comparing the result of this operation to see if it is greater than 0, we are able to test if any of Inputs 0.0 through 0.3 are active.

```
IF                  (   IW0     )   the contents of Input Word 0
          AND           V15     )   are AND'ed with the decimal
                                    constant 15, and the result is
                                    compared...
                    >   V0          as being greater than value 0
THEN      ....                      the sentence will be true
```

The next example shows using the AND function with multibit operands in the Executive part of a sentence.

```
IF        ...                       if the conditions are true
THEN      LOAD      (   R38         then transfer the contents of
                                    Register 38 to the Multibit
                                    Accumulator.
          AND           R45     )   logically AND'ing to Register
                                    45 and placing the result
          TO            R17         in Register 17
```

7. STL Instruction Reference

BID

Purpose

To convert the contents of the multibit accumulator from Binary to BCD format.

This instruction is often used in conjunction with a device connected to the PLC's outputs (e.g. canned message displays, motor controls etc.)

These devices often expect input commands in BCD format.

Refer to the DEB instruction for conversion from BCD to Binary format.

Examples

The value to be converted must first be loaded into the multibit accumulator.

IF		I1.0	Start Servo motor button
THEN	LOAD	R26	Register 26 contains the new position information
	BID		Convert to BCD format
	AND	V15	mask all bits except 0-3
	TO	OW2	and transfer the results to Output Word 2 (connected to servo controller)

Please note that the maximum allowed range is 0-9999.

7. STL Instruction Reference

CFM

Purpose

The CFM (Call Function Module) instruction is used to request execution of a standard system routine which is resident within the System memory of the controller.

You should refer to the appropriate controller manual to see which CFM calls are available for your particular hardware configuration. These standard routines cannot be written by the user as they are integral sections of the controller's operating system.

Some Function Modules may use Function Units (FU) to pass information to/from user programs and Function Modules.

Examples

Depending upon the specific controller model, as well as the particular CFM routine being called, it may be necessary to provide several parameters when programming a CFM.

Example 1:
FPC100
This system routine can be used to unconditionally clear or reset a variety of operands. The call to this CFM accepts a single numeric parameter. If we use a Value of 2, the Function Module will Reset ALL Flags to 0's.

IF		I1.2	Reset button pressed
THEN	CFM	0	Call Function Module 0
	WITH	V2	pass a value of 2 to parameter number 1, which here results in ALL FLAGS being placed in a RESET state.

7. STL Instruction Reference

Example 2:
FPC405
This system routine can be used to enable interrupt driven high speed event counting on Input 0 of the 405 Interrupt-CPU. This CFM requires that several parameters accompany the system call.

The first parameter specifies the number of the program we want to execute when our final count is reached. The second parameter allows us to specify whether we want to recognize the rising or falling signal edge. Parameter 3 allows specification of the preselected number of pulses we want to count before executing the program number specified in parameter 1.

```
IF                      I2.2    Motor start button
        AND     N       O2.1    & motor not already running
THEN    SET             O2.1    Start the Motor
        CFM             2       Call Function Module 2,
                                enabling the interrupt function
                                for CPU Input 0
        WITH            V6      Program number to run when
                                final count is reached
        WITH            V0      specify watch for rising edge
                                of signal
        WITH            V200    final count is 200
```

7. STL Instruction Reference

CMP

Purpose

The CMP (Call Program Module) instruction is used to request execution of an external program routine. Program modules may be considered similar to subroutines.

NOTE: It is NOT permissible to use the CMP instruction from WITHIN a Program Module.

Program modules may be written in one of several languages including STL and Assembler. Festo is able to supply a number of optimized program modules for handling specialized tasks such as:

- **Text I/O**
- **High Speed Counting**
- **32 bit arithmetic functions**

If you have a task that you are unable to handle using standard language facilities, please contact your Festo office...we may have already solved your problem!

Some Program Modules may use Function Units (FU) to pass information to/from user programs and Program Modules.

Please refer to the CFM instruction for calling standard Festo Rom-resident routines.

Examples

Depending upon the specific controller model, as well as the particular Program module being called, it may be necessary to provide several parameters when using a CMP.

Example: FPC100
This program module can be used to transmit text. The call to this particular CMP accepts several parameters depending upon the function desired.

```
IF                          I1.5        if tank high level sensor
THEN    CMP                 7           Call Program Module 7
        WITH                V0          specify text string output
        WITH                'Tank #1 is Over-Full'
```

7. STL Instruction Reference

The previous example merely serves to provide a general understanding of the way in which program modules are called. The actual calling procedures vary greatly, so the user must always refer to the appropriate documentation.

Simple Modules:

In the situation where the user merely writes a subroutine as a program module, it is not necessary to pass any parameters. In such cases a simplified call may be made:

IF... THEN	CMP	4	call program module that does not require any parameters

7. STL Instruction Reference

CPL

Purpose

This command complements the contents of the multibit accumulator using the two's complement method.

In principle, the effect of using the CPL instruction is the same as multiplying a number by -1 when applied to signed integers.

Examples

The following illustrates applying the CPL instruction to a 16 bit number which has been loaded to the multibit accumulator:

0	0	0	1	0	0	1	0	0	1	1	0	0	1	1	1	4711
1	1	1	0	1	1	0	1	1	0	0	1	1	0	0	1	CPL = -4711

The value to be operated on must first be loaded into the multibit accumulator. In the following example, the program will check to see if Register 32 contains a negative number, and if so will convert the number to a positive number and store it in Register 22.

```
IF              (   R32     )   test to see if Register 32 is
                <   V0          less than 0....a negative
                                value
THEN    LOAD        R32         and if it is negative load it to
                                the multibit accumulator
        CPL                     apply the compliment
                                instruction and copy the
        TO          R22         to Register 22.
```

7. STL Instruction Reference

DEB

Purpose

To convert the contents of the multibit accumulator from BCD to Binary format.

It is common that various peripheral equipment may report information (values etc.) to a PLC via standard PLC Inputs. In order to minimize the number of Inputs required, the peripheral device may use BCD encoding.

Since the DEB instruction operates on the contents of the multibit accumulator, the value to be converted must first be loaded into the Multibit accumulator.

Examples

For example, if we used two BCD thumbwheel switches to allow the entry for the number of cycles a machine should run, the following instructions might be used.

We have connected the BCD switches to Inputs 0-7 of Input Word 1 and Input 0.3 is used to actually enter the current settings, which are then stored in Counter Word 2.

IF			I0.3		When Input 0.3 is activated,
THEN	LOAD	(	IW1		copy the COMPLETE Input Word to the Multibit Accumulator, and then use
	AND		V255	)	the AND function to MASK off Inputs 8-15. Whether or not Inputs 0.8-15 exist or not, this ensures that we only have the true value of the BCD switches lin the accumulator.
	DEB				Perform the actual BCD to Decimal conversion and then
	TO		CW2		copy the result to Counter Word 2.

Please note that the maximum allowed range is 0-9999.

7. STL Instruction Reference

DEC

Purpose

The DECrement instruction reduces the value of any multibit operand by 1. Unlike other arithmetic instructions, the DECrement operation may be carried out directly without the need to first load the operand to be DECremented to the multibit accumulator.

While the DECrement instruction can be used with any multibit operand, it is most often used in conjunction with Counters.

Chapter 10 contains specific, detailed information on how to work with counters.

Examples

In the following example we will assume that on a bottle filling line, Input 1.3 is activated each time a bottle passes by a counting station. The total number of bottles is to be stored in Register 9.

However, sometimes a bottle is not completely filled and this is tested further on in the production process. If a partially filled bottle is sensed, the existing total count is to be reduced by 1.

```
IF                          I1.3    Input 1.3 senses all bottles
                                    which we want to totalize
THEN    INC                 R9      so add 1 to the existing count

IF                          I2.2    When a bottle arrives at the
                                    level test station
        AND     N           I3.6    and it's not properly filled
THEN    DEC                 R9      subtract 1 from the total
```

The above DECrement instruction is equivalent to:

```
IF...
THEN    LOAD                R9      load R9 to the Multibit
                                    accumulator
                -           V1      subtract 1
        TO                  R9      & copy the result back to R9
```

7. STL Instruction Reference

EXOR

Purpose

To combine two or more single or multibit operands in the Conditional or Executive part of a Sentence using the logical EXOR (EXclusive OR) operation.

Examples

Single Bit

In the following example, Output 1.3 will be switched on if either I1.1 or I1.2 is active, but not if both are active.

IF		I1.1	If either 1.1 or 1.2 is active
	EXOR	I1.2	(BUT NOT BOTH!)
THEN	SET	O1.3	switch on Output 1.3

Multibit

The following illustrates the logical bit-wise EXOR operation applied to two 8 bit operands:

0	0	1	0	1	1	0	1	operand 1 = 45 decimal
1	1	1	0	1	1	0	0	EXOR with operand 2 =236 decimal
1	1	0	0	0	0	0	1	result = 193 decimal

When used in the Conditional part of a Sentence, this function allows the result of a Logical EXOR function of two Multibit operands to be compared to a third Multibit operand or constant.

In the following example, we will use the power of the EXOR function to control an 8 bottle filling station. The 8 filling positions are located on part of a bottle conveying system. As bottles pass by they must be checked. At any of the 8 positions it is possible that a bottle may or may not be present. Any unfilled bottles shall be filled and when all present bottles are filled the bottling line shall again continue moving, searching for the next group to fill.

7. STL Instruction Reference

Input word 0 (I0.0-I0.7) is connected to the bottle-present sensors and Input word 1 (I1.0-I1.7) is connected to the bottle filled sensors.

Output word 0 (O0.0-O0.7) controls the fluid dispensers, while Output 1.0, when set closes a gate, thus holding the bottles in place for filling.

```
STEP 10
IF                  N   O1.0      bottles are not stopped
        AND             I0.7      & bottle exists at position 7
THEN    SET             O1.0      stop bottles from moving
        LOAD        (   IW0       see which bottles are present
        EXOR            IW1   )   and are NOT filled.  Then turn
        TO              OW0       on the outputs to fill bottles

IF                  (   OW0       If ALL Outputs are Off
                    =   V0    )
        AND             O1.0      and the bottles were stopped
                                  for filling...
THEN    RESET           O1.0      let bottles move again
        JMP TO          10        goto Step 10
```

7. STL Instruction Reference

INC

Purpose

The INCrement instruction increases the value of any multibit operand by 1. Unlike other arithmetic instructions, the INCrement operation may be carried out directly without the need to first load the operand to be INCremented to the multibit accumulator.

While the INCrement instruction can be used with any multibit operand, it is most often used in conjunction with Counters.

Chapter 10 contains specific, detailed information on how to work with counters.

Examples

In the following example we will assume that on a bottle filling line, Input 1.3 is activated each time a bottle passes by a counting station. The total number of bottles is to be stored in Register 9.

However, sometimes a bottle is not completely filled and this is tested further on in the production process. If a partially filled bottle is sensed, the existing total count is to be reduced by 1.

```
IF                          I1.3    Input 1.3 senses all bottles
                                    which we want to totalize
THEN    INC                 R9      so add 1 to the existing count

IF                          I2.2    When a bottle arrives at the
                                    level test station
        AND     N           I3.6    and it's not properly filled
THEN    DEC                 R9      subtract 1 from the total
```

The above INCrement instruction is equivalent to:

```
IF...
THEN    LOAD                R9      load R9 to the Multibit
                                    accumulator
                +           V1      add 1
        TO                  R9      & copy the result back to R9
```

7. STL Instruction Reference

INV

Purpose

This command complements (INVerts) the contents of the multibit accumulator using the one's complement method.

When applied to signed integers, this is equivalent of multiplying a number by -1 and then adding -1.

Examples

The following illustrates applying the INV instruction to a 16 bit number which has been loaded into the multibit accumulator.

0	0	0	1	0	0	1	0	0	1	1	0	0	1	1	1
1	1	1	0	1	1	0	1	1	0	0	1	1	0	0	0

INV =

The INV instruction can be of use when it is desired to 'flip' (invert) each and every bit as contained in the multibit accumulator.

In the following example, a mixing machine has 16 stations. The mixing cycle consists of alternating time periods of shaking and settling.

During normal operation workers add or remove containers randomly. Only those stations that have containers in placed are to be activated. Sensors are provided to see which stations are to be activated.

```
STEP 10
IF                    N     T1      No Time cycle in progress
THEN      LOAD              OW1     Current status Outputs 0-15 for
                                    each station shakers is copied
                                    to the multibit accumulator
          INV               OW1     now we 'flip' the status of each
                                    output...those that are On go
                                    Off etc. (but this is only done
                                    within the MBA!)
          AND               IW1     now correct for any stations
                                    that were off, but were turned
                                    on and have no container.
          TO                OW1     finally actually switch on the
                                    appropriate Outputs.
          SET               T1      Start the timer

STEP 20                             wait until process done
IF                    N     T1      time period done
THEN      JMP TO            10      back to Step 10
```

7. STL Instruction Reference

JMP TO

Purpose

To provide a means to influence the flow of program execution based upon programmable criteria. Analogous to the BASIC language instruction GOTO.

Please note that use of the JMP TO instruction can remove the requirement that the LAST sentence of a STEP be true for program execution to continue.

The JMP TO instruction can also be used to prioritize the execution of sentences within a Step.

Examples

In the first example, the JuMP instruction is used within a parallel program to detect and then react to an ESTOP condition. Step 20 contains all of the sentences that are processed in a parallel manner.
Note that ESTOP is a normally-closed button.

```
STEP 20
...prior sentences in Step 20

IF                    N   I1.1      see is ESTOP was pressed
THEN    LOAD              V0        if so, turn off ALL Outputs as a
                                    group in
        TO                OW0       Output Word 0
        TO                OW1       and Output Word 1
        JMP TO            80        then goto special routine

...remaining sentences in Step 20

STEP 80                             ESTOP Routine
IF                        I1.1      wait here until the ESTOP
                                    signal is no longer sensed
        AND               I2.1      and the RESET button is
                                    pressed.
THEN    JMP TO            20        continue at Step 20
```

The following example uses multiple jumps within a Step and illustrates a situation whereby a machine operator **must** select 1 of 3 possible choices.

7. STL Instruction Reference

```
STEP 40                                     Operator MUST select only 1
                                            of 3 possible choices
IF                          I1.1            sequence 1 selected
        AND         N       I1.2            and not sequence 2
        AND         N       I1.3            and not sequence 3
THEN    JMP TO              100             section for sequence 1

IF                          I1.2            sequence 2 selected
        AND         N       I1.1            and not sequence 1
        AND         N       I1.3            and not sequence 3
THEN    JMP TO              150             section for sequence 2

IF                          I1.3            sequence 3 selected
        AND         N       I1.2            and not sequence 2
        AND         N       I1.1            and not sequence 1
THEN    JMP TO              200             section for sequence 3
```

By carefully ordering multiple sentences within a Step, along with proper use of the JuMP instruction, it is easy to prioritize operational sequences.

The next example assumes that Steps up through 50 contain instructions for machine processing, and that upon reaching STEP 60 the machine is to check Inputs 1.1, 1.2 and 1.3 and wait until the FIRST input appears and then process only (1) of these inputs with Input 1.1 having the highest priority and Input 1.3 having the lowest priority.

```
STEP 60
IF                  N       I1.1        wait until at least one of the
        AND         N       I1.2        required input becomes true
        AND         N       I1.3
THEN    JMP TO              60

IF                          I1.1        it's the highest priority input
THEN    JMP TO              100         step 100

IF                  N       I1.1        make sure no higher priority
        AND                 I1.2        request exists
THEN    JMP TO              150         step 150
IF                  N       I1.1        make sure no higher priority
        AND         N       I1.2        request exists
        AND                 I1.3
THEN    NOP                             ok to just proceed to the next
                                        program step
```

7. STL Instruction Reference

LOAD... TO

Purpose

The **LOAD** instruction allows copying (loading) Single and Multibit operands to the Single Bit Accumulator and Multibit Accumulator (respectively) in preparation for:

1. performing logical and/or mathematical operations.
2. or as a required intermediate step for transferring information between operands.

The ...**TO** part of the instruction allows specifying the destination operand.

The LOAD...TO instruction is most often used with Multibit operands.

Examples

Single Bit Loads

Single Bit Syntax

Source	Optional Operation	Destination
LOAD SBO	none	TO SBO
LOAD I1.0		TO O1.0
LOAD SBO	AND SBO	TO SBO
LOAD I1.0	AND N I1.1	TO O1.0

Note: SBO = any **S**ingle **B**it **O**perand

While the above examples are valid STL instructions, they are not typically used. They are, however, equivalent to:

```
IF                      I1.0    If Input 1.0 is Active then
THEN    SET             O1.0    switch on Output 1.0  else
OTHRW   RESET           O1.0    turn off Output 1.0


IF                      I1.0    If Input 1.0 is Active and
        AND     N       I1.1    and Input 1.1 is NOT Active
THEN    SET             O1.0    switch on Output 1.0  else
OTHRW   RESET           O1.0    turn off Output 1.0
```

7. STL Instruction Reference

Multibit Loads

Multibit Bit Syntax

Source	Optional Operation	Destination
LOAD MBO/V	none	TO MBO
LOAD R6		TO TW1
LOAD MBO/V	AND MBO/V	TO MBO
LOAD R11	SHL	TO CW4
LOAD CW2	+ V3199	TO R28

Note: MBO/V = any **M**ulti **B**it **O**perand or Value

The use of the LOAD instruction with Multibit operands and values, when used in conjunction with the available mathematical or logical operations, provides very powerful processing capabilities.

The following examples illustrate some of the diverse functions which can be accomplished using the LOAD instruction.

Switching off ALL Outputs of a system

We will assume our system contains 64 Outputs organized as 4 x16 bit words. Using the typical RESET instruction would require program logic such as:

```
IF                      I1.0    eg.:. a Reset Button
THEN    RESET           O1.0    turn off 1 Output
        RESET           O1.1    and another
        ....                    until we repeated this
                                command for each of the 64
                                Outputs.
```

By using the LOAD instruction the same result can be accomplished by:

```
IF                      I1.0    eg.:. a Reset Button
THEN    LOAD            V0      put zero in Multibit
                                Accumulator
        TO              OW1     turn off Outputs 1.0 - 1.15
        TO              OW2     turn off Outputs 2.0 - 2.15
        TO              OW3     turn off Outputs 3.0 - 3.15
        TO              OW4     turn off Outputs 4.0 - 4.15
```

Note that once a Value (in this case 0) has been loaded into the Multibit Accumulator, it can be copied (using **TO**) to multiple destinations without having to be reloaded.

7. STL Instruction Reference

Summary

The LOAD instruction may well be one of the most powerful instructions in the STL language.

It is important to remember that the LOAD instruction merely prepares the system for the instructions that follow.

Note:
When a LOAD instruction is executed, the specified Multibit Operand or Value is loaded into the Multibit Accumulator (MBA).

The MBA is 16 bits wide. If the Multibit Operand specified as the source (e.g. LOAD MBO) is only 8 bits wide (e.g. I/O module with only 8 discrete points) then the upper byte of the MBA will be filled with 0's.

In the same way, if the MBA is transferred (via the TO instruction) to an 8 bit wide destination, the upper 8 bits will be lost.

Additional examples that include the LOAD instruction can be found throughout this chapter as part of the explanations and examples provided for many of the STL instructions including:

SHL, SHR, ROR, ROL, SWAP, WITH, AND, OR, EXOR etc.

7. STL Instruction Reference

NOP

Purpose

The NOP (**N**o **Op**eration) instruction, which at first may seem to be of little value, is quite often helpful when programming.

The actual consequence of using the NOP instruction depends on whether it is used in the Conditional or Executive part of a sentence.

Examples

Conditional Part

When used in the Conditional part of a sentence, the NOP instruction can be used to construct a sentence which will **always** be evaluated as true and any programmed instructions in the Executive part will be performed.

```
STEP 45
IF                      NOP     always true
THEN    SET             T6      start timer 6
        SET             O1.2    switch on Output 1.2
```

Parallel Processing

When a program step contains multiple sentences which are to be processed (scanned) continuously, the NOP instruction may be used to control program flow.

```
STEP 11
IF                      I1.4    If Input 1.4 is active
THEN    SET             T4      start Timer 4

IF                      I3.0    Manual Start Input
THEN    SET             O1.6    Start Motor
OTHRW   RESET           O1.6    else Stop Motor

IF                      T4      Timer 4 running
        AND             O1.6    motor is running
THEN    INC             CW3     increment cycle count

IF                      I2.2    Emergency Button
THEN    JMP TO          90      Exit this scan ...

IF      NOP                     always...
THEN    JMP TO          11      continue this scan...

STEP 90                         special routine
IF                  N   I2.2    Emergency Button released
        AND             I3.3    Reset Button
THEN    JMP TO          11      go back to Step 11, else wait
```

7. STL Instruction Reference

Executive Part

When used in the Executive part of a sentence, the NOP instruction is evaluated as a 'do nothing' instruction. Although this may appear to have little practical value, it is often quite useful when the programmer wishes to wait until the programmed conditions become true before proceeding with program execution.

```
STEP 60
IF                          I1.5    Input 1.5 is active
          AND               T7      Timer 7 is running
          AND       N       C2      Counter 2 is not active
THEN                        NOP     after the above conditions are
                                    all satisfied...proceed.

STEP 70   ....
```

7. STL Instruction Reference

OR

Purpose

1. To combine two or more single or multibit operands in the Conditional part of a Sentence using the logical OR operation.

2. To perform logical OR 'ing of two multibit operands (or values) in either the Conditional or Executive part of a sentence.

Examples

Single Bit

IF		I1.1	If Input 1.1 is Active
	OR	T6	or Timer 6 is running
THEN	SET	O1.5	turn on Output 1.5

Multibit

The following illustrates the logical bit-wise OR operation applied to two 8 bit operands:

0	0	1	0	1	1	0	1	operand 1 = 45 decimal
1	1	1	0	1	1	0	0	OR operand 2 = 236 decimal
1	1	1	0	1	1	0	1	result = 237 decimal

The OR function can be used with Multibit operands and values in both the Conditional as well as the Executive parts of a Sentence.

When used in the Conditional part of a Sentence, this function allows the result of a logical OR function of two Multibit operands or values to be compared to a third Multibit operand or value.

IF		(	R43		the contents of Register 43
	OR		R7	)	are OR'd to the contents of Register 7. Next the result is
		=	V100		equal to 100
THEN	...				if so, then perform any instructions provided.

7. STL Instruction Reference

The above sentence is **not** to be interpreted as:

"If R43 =100 or R7=100 then ..."

which could be written as:

```
IF              (   R43     if this is true...
                =   V100  )
          OR                or
                (   R7      if this is true...
                =   V100  )
THEN      ...               then...
```

The next example is a machine which consists of 16 parallel conveyors, each one of which transports component parts to an assembly area.

Component parts are loaded by hand at one or more of 3 possible locations on each conveyor. Each conveyor includes 3 sensors that check if parts have been loaded.

When all 16 conveyors have at least one component loaded, then each conveyor shall start running. As a part reaches the end position of each conveyor, that conveyor shall stop. Each conveyor contains a sensor to sense when a part is present at the end position.

```
STEP 50                                 The criteria to start
IF              (   OW1                 all conveyors are now
                                        stopped (outputs 1.0 - 1.15 )
          OR        IW4                 AND all 16 end positions are
                                        clear
                =   V0          )
          AND   (   IW1                 all load station 1 sensors for
                                        conveyors 1,2,3
          OR        IW2                 all load station 2 sensors for
                                        conveyors 1,2,3
          OR        IW3                 all load station 3 sensors for
                                        conveyors 1,2,3
                =   V65535      )       all 16 conveyors have at
                                        least 1 component loaded
THEN      LOAD      V65535              so turn-on all 16 conveyors
          TO        OW1                 which are controlled by
                                        Outputs 1.0 - 1.15
STEP 60                                 turn off each conveyor
THEN      LOAD      IW4                 as a component reaches the
          TO        OW4                 end position
IF              (   OW4                 when all conveyors are
                =   V0          )       stopped
THEN      JMP TO    50                  then start again
```

7. STL Instruction Reference

PSE

Purpose

To mark the end of a program (**P**rogram **S**ection **E**nd) and cause a program change. Will also result in a Virtual Processor Swap for controller models that support multi-tasking (see Appendix C). This instruction is not available in the FEC and IPC.

Upon returning to the program which executed the PSE instruction, the program will continue processing:

- at the first sentence of the current Step **or**
- at the first sentence in the program when no Steps exist

Examples

If an STL program merely ends with a normal sentence, and no further instructions are given, the program will cease running.

Typical programs or program sections are terminated using either the PSE instruction or the JUMP TO instruction.

Example 1

```
STEP 10
IF                          I1.1        Start Button
THEN        SET             O2.1        Extend Cylinder

STEP 20
IF                          I3.1        Cylinder is extended
THEN        RESET           O2.1        so retract cylinder
            PSE                         goto first sentence
OTHRW       PSE                         goto first Sentence
```

When a program has been constructed without Step labels, and the program is to be processed continuously in a scanning manner; the program should end with a PSE instruction.

Example 3

```
...                                     prior sentences
...                                     "
...                                     "
IF                          NOP         always true
THEN                        PSE         program section end ....go to
                                        top sentence in program
```

7. STL Instruction Reference

RESET

Purpose

The RESET instruction is used to change the status of Single Bit operands to a logical 0 (zero).

RESETting an operand that is **already** reset has no effect.

The actual effect of issuing a RESET instruction varies according to the operand addressed. The following table provides a summary of using the RESET instruction.

Detailed information on using the RESET instruction can be found in chapters 8, 9, 10 and 12.

Examples

Operand	Syntax	Effect
Output	RESET O1.6	Switches Output 1.6 off.
Flag	RESET F2.12	Forces the status of Flag 12.2 to be '0'.
Counter	RESET C6	The status of Counter 6 is changed to inactive.
Timer	RESET T4	The status of Timer 4 is changed to inactive.
Program	RESET P2	Program 2 is stopped
Program status	RESET PS2	Program 2 is suspended
Error Status	RESET E	Clears the Error Status Bit

7. STL Instruction Reference

ROL

Purpose

The **RO**tate **L**eft instruction rotates the contents of the Multibit Accumulator to the Left by one position.

The most significant bit (bit 15) is transferred to the least significant bit position. Also see the ROR, SHR and SHL instructions.

It should be remembered that the LOAD...TO instruction is normally used first to prepare the Multibit Accumulator and again after the ROR instruction to copy the results to the desired MBO.

Examples

The following illustrates the effect of using the ROL instruction.

0	1	0	1	0	1	1	0	0	0	0	1	1	1	0	1	LOAD MBO
1	0	1	0	1	1	0	0	0	0	1	1	1	0	1	0	1st ROL
0	1	0	1	1	0	0	0	0	1	1	1	0	1	0	1	2nd ROL
0	1	0	1	1	0	0	0	0	1	1	1	0	1	0	1	TO MBO

```
IF                    N   T6      If Timer 6 is not running
THEN    LOAD              OW1     load all 16 bits of Output
                                  Word 1 to the MBA
        ROL               rotate left the first time
        ROL               rotate left a second time
        TO                OW1     and copy the result back to
                                  the same place....could be an
                                  MBO!
```

This instruction may also find good use when applied to machinery that use various types of rotary tables or conveyors to track the status of production as the machinery indexes.

7. STL Instruction Reference

ROR

Purpose

The **RO**tate **R**ight instruction rotates the contents of the Multibit Accumulator to the Right by one position.

The least significant bit (bit 0) is transferred to the most significant bit position. Also see the ROL, SHR and SHL instructions.

It should be remembered that the LOAD...TO instruction is normally used first to prepare the Multibit Accumulator and again after the ROR instruction to copy the results to the desired MBO.

Examples

The following illustrates the effect of using the ROR instruction.

0	1	0	1	0	1	1	0	0	0	0	1	1	1	0	1	LOAD MBO
1	0	1	0	1	0	1	1	0	0	0	0	1	1	1	0	1st ROR
0	1	0	1	0	1	0	1	1	0	0	0	0	1	1	1	2nd ROR
0	1	0	1	0	1	0	1	1	0	0	0	0	1	1	1	TO MBO

```
IF                    N   T6      If Timer 6 is not running
THEN    LOAD              OW1     all 16 bits of Output Word
        ROR               ROR     rotate right the first time
        ROR               ROR     rotate right a second time
        TO                OW1     and copy the result back to
                                  the same place....could be an
                                  MBO!
```

This instruction may also find good use when applied to machinery that use various types of rotary tables or conveyors to track the status of production as the machinery indexes.

7. STL Instruction Reference

SET

Purpose

The SET instruction is used to change the status of Single Bit operands to a logical 1 (one).

The actual effect of issuing a SET instruction varies according to the operand addressed. The following table provides a summary of using the SET instruction.

Detailed information on using the SET instruction can be found in chapters 8, 9, 10, and 12.

Examples

Operand	Syntax	Effect
Output	SET O1.6	Switches Output 1.6 ON.
Flag	SET F2.12	forces the status of Flag 12.2 to be '1'.
Counter	SET C6	1. Counter Word 6 is loaded with a value of 0. 2. The status bit of Counter 6 (C6) is set to active (=1).
Timer	SET T4	1. The contents of Timer Pre-select 4 is copied to Timer Word 4. 2. The status bit of Timer 4 (T4) is set to active (=1).
Program	SET P2	Program 2 is started from the beginning.
Program status	SET PS2	Program 2 will be continued from where it was suspended by the instruction RESET PS2

7. STL Instruction Reference

SHIFT

Purpose

The SHIFT instruction executes a swap between the Single Bit Accumulator (SBA) and a Single Bit Operand (SBO).

This instruction can be used to construct Shift Registers of varying lengths...longer or shorter than the 16 bit manipulations performed by the SHL and SHR instructions.

To operate properly, the SBA must first be loaded and then any number of single bit SHIFT's can be programmed.

Examples

In the following example, each time Input 1.0 is activated, the status of Outputs 1.1 through 1.4 are to be updated:

Output 1.4 shall take on the previous status of Output 1.3

Output 1.3 shall take on the previous status of Output 1.2

Output 1.2 shall take on the previous status of Output 1.1

Output 1.1 shall take on the status of Input 1.1

```
STEP 10
IF                          I1.0    Input Activated
THEN    LOAD                I1.1
        TO                  F0.0    a Flag is used here to avoid
                                    'writing' to an Input, which
                                    would otherwise occur.
        SHIFT               O1.1    SWAP F0.0 <-> O1.1
        SHIFT               O1.2    SWAP O1.1<-> O1.2
        SHIFT               O1.3    SWAP O1.2<-> O1.3
        SHIFT               O1.4    SWAP O1.3<-> O1.4

STEP 20
IF                      N   I1.0    wait for Input to go away
THEN    JMP TO              10      repeat
```

See section 12 (Flags and Flag Words) for an alternative method of constructing Shift Registers.

7. STL Instruction Reference

SHL

Purpose

The **SH**ift **L**eft instruction moves (shifts) the contents of the Multibit Accumulator to the Left by one position.

The most significant bit (bit 15) is discarded and the least significant bit position is filled with a 0. Also see the ROL, ROR, SHR instructions.

A typical use of the SHL instruction is to emulate a Shift Register.

The SHL instruction may also be used to multiply any MBO or value by 2. The programmer must check for any possible overflow.

It should be remembered that the LOAD...TO instruction is normally used first to prepare the Multibit Accumulator and again after the SHL instruction to copy the results to the desired MBO.

Examples

1	1	0	1	0	1	1	0	0	0	0	1	1	1	0	1	LOAD MBO
1	0	1	0	1	1	0	0	0	0	1	1	1	0	1	0	SHL
1	0	1	0	1	1	0	0	0	0	1	1	1	0	1	0	TO MBO

Shift Register

The following example demonstrates using SHL in combination with a MBO to emulate a shift register. While any multibit operand can be used, we have chosen to use a Flag Word, since Flag Words may be addressed on both a bit or word basis (see Chapter 12).

We will assume that we are controlling a machine that assembles ribbon cartridges for computer printers. The process begins at station 1 where empty lower cartridge shells are placed on the assembly line; through station 10 where completed assemblies are off-loaded to a packing machine.

At each station (1-10), after the respective assembly operation is completed, a quality check is made. Defective assemblies are removed immediately.

In addition, when the machine is first started in the morning, and later shut down at night, only stations which contain valid components are to be processed.

7. STL Instruction Reference

While each station includes sensors to make certain all parts are properly positioned prior to operation, the use of a shift register will greatly simplify our processing needs.

To simplify this example, we will only consider stations 1-3.

```
STEP 40                                    Process valid stations...FW1
                                           contains a bit pattern of where
                                           valid assemblies exist
IF                          F1.1           valid part was at Station 1
        AND         N       T1             operation done
        AND         N       I2.1           QC BAD at Station 1
THEN    RESET               F1.1           BAD PART REMOVED

IF                          F1.2           valid part was at Station 2
        AND         N       T1             operation done
        AND         N       I2.2           QC BAD at Station 2
THEN    RESET               F1.2           BAD PART REMOVED

IF                          F1.3           valid part was at Station 3
        AND         N       I1.3           QC BAD at Station 3
        AND         N       T1             operation done
THEN    RESET               F1.3           BAD PART REMOVED

IF                          T1             operations done
THEN    SET                 O1.1           Index assembly Line

STEP 50
IF                  N       I2.0           Line is indexing...
THEN    LOAD                FW1            Get shift register
        SHL                                update it
        TO                  FW1            and store it

STEP 60
IF                          I2.0           Indexing done
THEN    RESET               O1.1
        JMP TO              20             resume processing...
```

Multiplication

The SHL instruction can also be used to multiply the contents of the MBA by 2.

```
IF                          I1.0           Parts sensor
THEN    LOAD                R6             Register 6
        SHL                                multiply by 2
        SHL                                again, so actually x4
        TO                  R6             and store the result
```

7. STL Instruction Reference

SHR

Purpose

The **SH**ift **R**ight instruction moves (shifts) the contents of the Multibit Accumulator to the Right by one position.

The least significant bit (bit 0) is discarded and the most significant bit position is filled with a 0. Also see the ROL, ROR, SHL instructions.

The SHR instruction may also be used to divide any MBO or value by 2. The programmer must check for any possible overflow/underflow or if the dividend is an odd number, in which case the result will be incorrect as only integers (whole numbers) are supported.

It should be remembered that the LOAD...TO instruction is normally used first to prepare the Multibit Accumulator and again after the SHR instruction to copy the results to the desired MBO.

Examples

Division

1	1	0	1	0	1	1	0	0	0	0	1	1	1	0	1
0	1	1	0	1	0	1	1	0	0	0	0	1	1	1	0
0	1	1	0	1	0	1	1	0	0	0	0	1	1	1	0

LOAD MBO
SHL
TO MBO

7. STL Instruction Reference

SWAP

Purposes

Provides the means of exchanging (swapping) the high order byte (bits 8-15) and the low order byte (bits 0-7) of the Multibit accumulator.

The Multibit Accumulator must be loaded with the appropriate MBO or value before executing the SWAP instruction.

Examples

1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	LOAD MBO /V
0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	SWAP
0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	TO MBO

8. Accessing Inputs and Outputs

8. Accessing Inputs and Outputs

Contents

BRIEF 77

DETAILS 77

I/O Organization 77
I/O Words 77
Discrete I/O Stages 78

Using Inputs in Programs 78
Discrete Inputs 78
Input Words 79

Using Outputs in Programs 80
Discrete Outputs 80
Output Words 80

8. Accessing Inputs and Outputs

BRIEF

This chapter details how to access standard digital Inputs and Outputs using the STL language. Inputs and Outputs that are connected to the CPU (in which the controlling STL program is stored) by means of:

- Field Bus System
- Network System

as well as Analog I/O are not covered in this section. Please refer to chapter 13 and the documentation covering the specific system components for more information.

DETAILS

I/O Organization

Festo programmable controllers organize **I**nputs and **O**utputs (I/O) on a **word (group)** basis. Depending on the particular controller model (or I/O module for modular systems) each I/O group usually consists of either 8 or 16 discrete Inputs or Outputs.

I/O Words

These complete groups or words are referenced by their type (**I**nput or **O**utput and the Word address number (**n**). This address number is generally fixed in smaller controllers and configurable (via switches) in modular systems.

Input Words are referenced as **IWn** while Output Words as declared as **OWn**. Examples include:

IW1	Input Word 1
IW7	Input Word 7
OW0	Output Word 0
OW2	Output Word 2

It should be noted that every Input and Output within a system must have a unique address number. For example: It is not permissible for a system to have I/O's with duplicate addresses.

However, it **is** generally acceptable for a system to include an Input Word with the same address number as an Output Word (e.g. IW1 and OW1). Please refer to the respective hardware manual for your specific controller model.

8. Accessing Inputs and Outputs

Discrete I/O Stages

The individual Inputs or Outputs contained within the I/O group are referenced by specifying:

the I/O type (**I** or **O**) +

the Word address number (**n**) +

"**.**" followed by the particular I/O Stage number (**Sn**).

Stage numbers are 0 - 7 or 0-15 depending on the I/O group size. For example:

I3.2	Input stage 2 of Input Word 3
I0.15	Input stage 15 of Input Word 0
O2.7	Output stage 7 of Output Word 2
O0.0	Output stage 0 of Output Word 0

Using Inputs in Programs

Inputs are elements of the control system that are designed only to be read or queried. That is to say, they are connected to external devices such as sensors, switches, etc., which may or may not supply a signal to an individual input.

Discrete Inputs

By executing the appropriate STL instructions within the Conditional part of a sentence, the controller is able to determine the current status of a discrete Input.

IF		I1.1	Test for a **valid** signal at Input 1.1
IF	N	I3.3	Test for a **false** signal at Input 3.3

Multiple Inputs as well as other conditions can be combined in various logical combinations. Examples can be found in chapter 7.

8. Accessing Inputs and Outputs

Input Words

Sometimes it may be desirable or necessary to check the status of entire Input Words. To determine the status of a complete Input Word, it is necessary to read the value of the entire word and determine if it meets the desired criteria.

Readers who are not familiar with how to determine the value of binary numbers can refer to Appendix D.

For example, to test whether **all 8** Inputs of Input Word 2 are receiving valid signals we could logically AND each Input:

```
IF      I2.0    here we check to see if all 8 inputs
  AND   I2.1    of an 8 bit Input Word are receiving
  AND   I2.2    valid signals
  AND   I2.3
  AND   I2.4
  AND   I2.5
  AND   I2.6
  AND   I2.7
```

or by using the STL language's ability to evaluate complete Words, we can use the program sequence:

```
IF      (   IW2       merely test if all 8 inputs are
        =   V255  )   on...11111111 (binary) = 255
```

More complex tests, which would require long sequences if programmed bit by bit, are also easily accomplished using entire Input Words combined with other logical instructions.

To test if **one or more** of Inputs 1.5, 1.6 1.7 are valid can be done by:

```
IF      (   IW1       first get the entire word
  AND       V224  )   = 11100000 binary
        >   V31       if the result is greater than
                      here we have
```

Which is equivalent to:

```
IF      I1.5
  OR    I1.6
  OR    I1.7
```

8. Accessing Inputs and Outputs

Using Outputs in Programs

The Outputs of a programmable controller can be used to control various types of electrical devices by means of program instructions which switch on (SET) or switch off (RESET) the required Output.

Note: While Inputs may only be read (queried), Outputs may be written to (SET or RESET) and may also be queried in the same manner as Inputs. Therefore, references to Outputs may appear in both the Conditional as well as the Executive parts of an STL sentence.

Discrete Outputs

By executing the appropriate STL instructions within the Executive part of a sentence, the controller can switch a particular Output ON or OFF.

The **SET** instruction is used to switch **on** an Output, while the **RESET** instruction will turn an Output **off**.

```
IF...                                   whatever conditions are needed
THEN    SET     O1.2                    switch on Output 1.2
        RESET   O3.3                    turn off Output 3.3
```

SETting an Output which is already SET or RESETting an Output which is already RESET will have no effect.

As noted, Outputs may also be queried in the Conditional part...The following sentence checks if Input 2.4 is receiving a valid signal and if Output 2.2 is currently switched on:

```
IF              I2.4        input 2.4 active
        AND     O2.2        and Output 2.2 is ON
THEN    ....                desired actions
```

Output Words

Sometimes it may be desirable or necessary to test or alter the status of entire Output Words. In the same manner as Inputs can be manipulated on a group or Word basis, the same principles apply to Outputs.

For example, the STL sentence:

```
THEN    LOAD    V0
        TO      OW2
```

will result in **all** of the Outputs associated with Output Word 2 being switched off.

9. Using Timers

9. Using Timers

Contents

BRIEF 83

DETAILS 83
General information 83

Using a timer 83
Initializing a Timer Preselect 84
Example: Initializing a Timer Preselect with a clock rate 84
Example: Initializing a Timer Preselect without a clock rate 84

Starting a Timer 85

Checking the Status of a Timer 85

Stopping a Timer 85

Examples 87
Avoiding unwanted restarting by use of the STL step structure 87
Avoiding continuous restarting of Timers in Parallel processing 87

9. Using Timers

BRIEF

This chapter discusses how Timers are programmed using the STL language. In addition, an understanding regarding the internal functioning of STL Timers is presented.

The reader is directed to Appendix A of this document which provides information regarding the number of timers available in each controller model.

DETAILS

General Information

Each Timer as implemented in the STL language consists of several elements:

Element/Operand	Ref.	Function
Timer Status Bit	Tn	allows a program to test if a timer is active (running). This bit is changed to active when the timer is started (SET). When the programmed time period is complete or if the timer is stopped (RESET) the status bit becomes inactive.
Timer Preselect	TPn	a 16 bit operand that contains the value that defines the time period for Timer n.
Timer Word	TWn	a 16 bit operand to which the TP is transferred automatically when the Timer is started (SET). The contents are automatically decremented by the system at regular intervals.

Note: Controller models which incorporate back-up batteries maintain the contents of Timer Preselects during power-off periods.

Using a Timer

Several basic steps are required to use a timer in an STL program:

- a valid Timer Preselect must be established
- an instruction to start the Timer must be issued
- the status of the Timer (active/stopped) can be tested

9. Using Timers

Initializing a Timer Preselect

> **Note:**
> Depending upon which controller model is being used, it may or may not be required to specify a **clock rate** as well as a time value. Please refer to the hardware manual for the controller model you are programming.

Before any Timer can be used, the respective Timer Preselect must first be initialized with a value corresponding to the desired time period.

This initialization only needs to be performed again if the time value needs to be changed. It is not necessary to reload the Timer Preselect each time the Timer is started. Timer Preselects may be loaded with either a value or with the contents of any MBO (e.g. Register, Input Word, Flag Word etc.)

Example: Initializing a Timer Preselect with a clock rate

```
STEP 1                                  we do this first !
IF                      NOP             unconditionally
        LOAD            V10             value 10
        TO              TP4             to Timer Preselect 4
        WITH            SEC             clock rate=seconds ...Timer 4
                                        will now be a 10 second timer
```

The available clock rates are:

HSC	hundredths of seconds
TSC	tenths of seconds
SEC	seconds
MIN	minutes

Example: Initializing a Timer Preselect without a clock rate

```
STEP 1                                  we do this first !
IF                      NOP             unconditionally
        LOAD            V100            value 100....the unspecified
                                        clock rate will be in 1/100th of
                                        a second increments.
        TO              TP0             to Timer Preselect 0 = 1 sec.
```

The preceding example has initialized Timer 0 to have a duration of 1 second (100 x 1/100th second). The allowable range is 0-65535 which provides timer periods from 0.01s to 655.35 s (approx. 10 minutes).

9. Using Timers

Starting a Timer

Starting a timer only requires issuing a **SET** instruction and specifying which timer is to be started:

IF		I1.0	any condition to start
THEN	SET	T6	so start timer 6

Whenever the **SET Tn** instruction is executed, the following occurs:

1. The value stored in TPn (Timer Preselect n) is copied to TWn (Timer Word n).

2. Tn (Timer Status n) becomes '1' (active/running).

3. The controller automatically decrements the value stored in TWn at regular intervals.

4. When the value stored in TWn reaches 0 (zero), Tn (Timer Status) becomes '0' (inactive/stopped).

Note:
If an instruction to SET a Timer is executed, AND the timer specified is ALREADY active, the timer will be RESTARTED and a NEW timing period will be begin.

Checking the Status of a Timer

In order for timers to be useful in controlling processes, it is necessary to know when a programmed time is complete. The STL language provides the means to check whether a timer is active in the same manner as checking if an Input is active.

IF			T5	test if Timer 5 is **active** (running)
IF		N	T3	test if Timer 3 is **not active** (stopped)

Stopping a Timer

Stopping a timer only requires issuing a **RESET** instruction and specifying which timer is to be stopped:

IF		I2.0	Input to stop the timer
THEN	RESET	T5	Stop Timer 5

When the RESET Tn instruction is issued the Timer Status Bit (Tn) becomes 0 (inactive). If the timer was already inactive, there is no effect.

9. Using Timers

Figure 9-1 illustrates the relationship between the Timer Status Bit (Tn), the SET Tn, and RESET Tn instructions and the normal timing period.

The solid line represents the a normal timing sequence in which the status of the timer becomes active when the SET Tn instruction is executed and the status returns to inactive when the programmed time period is complete.

The broken line indicates that issuing a RESET Tn instruction will immediately return the timer status to inactive.

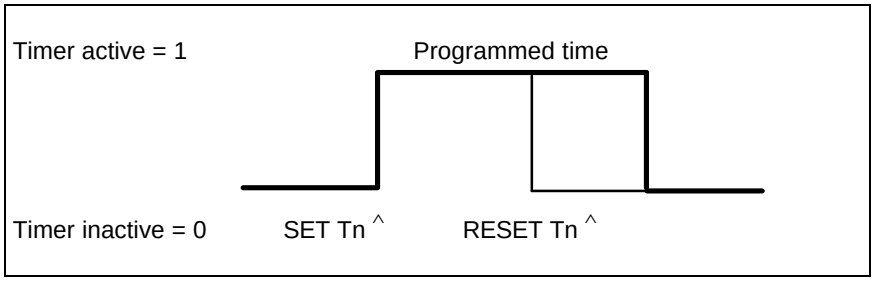


Figure 9-1 Timer Status Bit Tn

Caution:
It is important to understand when constructing Programs or Steps that contain multiple Sentences that will be processed in a parallel (scanning) manner; that **every time the conditional part of a Sentence evaluates as true, the instructions programmed in the executive part will be performed.** This **must** be considered in order to avoid uncontrolled multiple executions of most instructions including SET TIMER or INC/DEC Counter Word, SHL, etc.

The STL language does not use 'edge triggering'...**conditions are evaluated for truth each time they are processed without regard as to their prior status.**

This situation is easily handled by either using Steps, Flags or other means of control. The following examples show two possible ways in which this effect can be minimized.

9. Using Timers

Examples

Avoiding unwanted restarting by use of the STL Step structure

The next example shows a program section in which it is desired to turn on a motor for 3 seconds each time a button is pressed if the motor is not already running and at least 9 seconds time has passed since the motor was last run.

In this program the potential situation of timers being continually re-started is eliminated by combining the STL Step keyword with the **N Timer** instruction.

```
Step 1
IF                          NOP     initialize on power up
THEN    LOAD                V900    900 * .01sec unit of time
        TO                  TP0     Timer 0 is 2 sec pause time
        LOAD                V300    300 * .01sec unit of time
        TO                  TP2     Timer 2 is motor time
        SET                 T0      run the pause timer

STEP 10
IF                      N   T0      Timer 0 has finished
        AND             N   T2      Timer 2 is not running
        AND             N   O1.0    Motor not running
        AND                 I1.2    Button pressed
THEN    SET                 T2      Start Timer
        SET                 O1.0    Start motor

STEP 20
IF                      N   T2      motor time done
THEN    RESET               O1.0    stop the motor
        SET                 T0      start the pause timer
        JMP TO              10      start again
```

Avoiding continuous restarting of Timers in Parallel processing

It is important that the STL programmer understand that a Timer status bit (e.g. T2) can be tested using the instructions:

IF		T2	This test is true if Timer 2 is currently active and timing
IF	N	T2	This test is true if Timer 2 is **not currently active**

It is vital to understand that neither of these instructions allow testing whether Timer 2 has been started **and** is complete. Therefore, when STL programs are constructed in a manner allowing program sentences to be processed multiple times, measures must be taken to avoid unexpected results.

9. Using Timers

The following example presents a program section in which a pushbutton is used to extend a cylinder for a preset timer period. The program logic used will avoid the following problems which might otherwise arise:

Holding the pushbutton or pressing and releasing the button multiple times within the defined time period will not alter the programmed time.

STEP 1				**initialization first time only**
THEN	LOAD		V0	
	TO		OW0	all outputs off
	RESET		F3.0	clear Flag 3.0
	LOAD		V100	initialize timer
	TO		TP0	Make timer T0 1 second
STEP 2				**main scanning section**
IF			I1.0	Button 1 is pressed
	AND	N	T0	and Timer 0 is not running
	AND	N	F3.0	form edge detection
THEN	SET		T0	start Timer 0
	SET		O1.0	extend cylinder 1
	SET		F3.0	memorize rising edge of P.B.
IF		N	T0	Timer 0 is not running
	AND		O1.0	and cylinder is extended
THEN	RESET		O1.0	then retract the cylinder
IF		N	T0	Timer 0 is not active
	AND		F3.0	and we previously had a rising edge
	AND	N	I1.0	and the pushbutton is released...falling edge found!
THEN	RESET		F3.0	so get ready for next edge
IF			NOP	just keep scanning
THEN	JMP TO		2	the currrent step.

10. Using Counters

10. Using Counters

Contents

BRIEF 91

DETAILS 91

Standard Counters 91
Using Standard Counters 92
Initializing a Counter Preselect 92
Example: Initializing Counter Preselects with an absolute value 92
Example: Initializing Counter Preselects with a MBO 92
Starting a Counter 93
Checking the Status of a Counter 93
Counting Events 93
Stopping a Counter 93

Examples 94

Standard Counter 94
UP/DOWN Counters 97
Example: Using a register as a counter 97

10. Using Counters

BRIEF

This chapter describes how to utilize Counters using the STL language. Information concerning the elements that are associated with each Counter is also provided.

This section does not attempt to describe the operation or implementation of any special high-speed or interrupt driven counters which are available in some controller models. Details regarding the use of such special features can be found in the relevant controller hardware manual.

The reader is directed to Appendix A of the document which provides information regarding the number of counters available in each controller model.

DETAILS

Standard Counters

Each Counter as implemented in the STL language can be programmed in either of two ways. The standard method (sometimes referred to as an INCrementing counter) will be described first.

Element/Operand	Ref.	Function
Counter Status Bit	Cn	allows a program to test if a Counter is active (has not reached it's end value). This bit is changed to active when the counter is started (SET). When the programmed number of counts is reached, or if the counter is stopped (RESET) the status bit becomes inactive.
Counter Preselect	CPn	a 16 bit operand that contains the desired end count.
Counter Word	CWn	a 16 bit operand which contains the current number of counts recorded by means of the DECrement or INCrement instructions. When using standard counters and the SET Cn instruction is executed, the Counter Word is automatically changed to 0 (zero).

Note: Controller models which incorporate back-up batteries maintain the contents of Counter Preselects, Words and Status Bits during power-off periods.

10. Using Counters

Using Standard Counters

A standard counter is suitable for counting defined events and performing a desired action when the pre-defined quantity of events has occurred.

Standard Counters operate in the following manner:

- the value to be counted is stored in the Counter Preselect
- the Counter is Started which in turn:
  - places a 0 (zero) value in the Counter Word (CWn=0)
  - changes the status of the Counter to active (Cn=1)
- the current count can be INCremented or DECremented
- when the current count (CWn) = preselect (CPn) the Counter Status (Cn) changes to inactive (Cn=0)

Initializing a Counter Preselect

Before a standard Counter can be used, the respective Counter Preselect must first be initialized with a value corresponding to the number of events to be counted.

This initialization only needs to be performed again if the value for subsequent counting activities is to be changed. It is not necessary to reload the Counter Preselect each time the Counter is started.

Counter preselects may be loaded with absolute values or with the contents of any MBO (e.g. Register, Input Word, Flag Word etc.)

Example: Initializing Counter Preselects with an absolute value

```
IF                      I1.0    or any desired conditions...
THEN    LOAD            V100    we load an absolute value of
                                100 as the number of events
                                to be counted
        TO              CP4     to the Preselect for Counter 4
```

Example: Initializing a Counter Preselect with a MBO

```
IF                      I1.0    or any desired conditions
THEN    LOAD            IW1     Input Word 1 as a value in
        TO              CP5     to the Preselect for Counter 5
```

By means of the DEB instruction, we could also use external BCD switches to establish the count. See the DEB instruction in the STL Instruction Reference Chapter 7.

10. Using Counters

Starting a Counter

Starting a counter only requires issuing a SET instruction and specifying which counter is to be started:

IF		I1.2	desired conditions
THEN	SET	C2	activate Counter 2

Whenever the Set Cn instruction is executed, the following occurs:

1. The respective Counter Word (CWn) is loaded with a 0 (zero).

2. Cn (Counter Status n) becomes a '1' (active).

Note:
If an instruction to SET a Counter is executed, AND the Counter specified is ALREADY active, the Counter will be RESTARTED and the current count (in CWn) will be returned to 0 (zero).

Checking the Status of a Counter

In order to utilize counters in a meaningful way, it is necessary to be able to determine when the preselected count has been reached.

Counting Events

Once the counter has been activated (SET), the current count is maintained in the respective Counter Word, which can be updated using either the INC CWn or DEC CWn instructions.

Stopping a Counter

A counter can be stopped (deactivated) at any time by issuing the RESET Cn instruction. When the RESET Cn instruction is executed the Counter Status Bit (Cn) becomes 0 (zero). The contents of the Counter Word remain unchanged.

Caution:
In Programs or Steps that contain multiple Sentences that will be processed in a parallel (scanning) manner; **every time the conditional part of a Sentence evaluates as true, the instructions programmed in the executive part will be performed.** This **must** be considered in order to avoid uncontrolled multiple executions of instructions including SET TIMER or INC/DEC Counter Word, SHL, etc.

The STL language does not use 'edge triggering'...**conditions are evaluated for truth each time they are processed without regard as to their prior status.**

10. Using Counters

Examples

Standard Counter

The first example presents using a standard counter in conjunction with the STL Step structure to avoid uncontrolled multiple INCrements of the counter in Steps 10 and 15.

A push button is used to begin a machine cycle. The cycle will switch on a conveyor and count the bottles that pass by a sensor. After 25 bottles have passed the sensor, the conveyor is stopped, and a mechanism positions sealing corks in each bottle. Finally all the corks are pressed into the bottles 2 times for 1 second each.

10. Using Counters

STEP 1				**Power Up**
THEN	RESET		C0	bottle counter
	RESET		C1	press counter
	RESET		O1.0	switch off conveyor
	RESET		O1.1	switch off cork press
	LOAD		V25	how many to count
	TO		CP0	counter 0 preselect
	LOAD		V2	how many presses
	TO		CP2	counter 2 preselect
	LOAD		V100	100 x .01s = 1 second
	TO		TP0	Timer 0 Preselect
STEP 5				**Wait for start button**
IF			I1.0	Start Button
THEN	SET		C0	activate counter
	SET		O1.0	start conveyor
STEP 10				**Start counting bottles**
IF			I1.1	a bottle was sensed
THEN	INC		CW0	increment bottle counter
STEP 15				**25 bottles yet ?**
IF		N	C0	we're all done, so...
THEN	RESET		O1.0	stop conveyor
	SET		C2	activate press counter
	JMP TO		50	exit counting loop
OTHRW				else
IF		N	I1.1	wait for last bottle sensed to move away from sensor.
THEN	JMP TO		10	and continue counting
STEP 50				**25 bottles were counted**
THEN	SET		O1.1	Press corks
	SET		T2	Start Pressing Timer
	INC		CW2	count this pressing
STEP 60				**Wait for 1 second timer**
IF		N	T2	time is done
THEN	RESET		O1.1	stop pressing
STEP 70				**done ?**
IF		N	C2	pressed corks 2 times
THEN	JMP TO		5	back to Step 5
OTHRW	JMP TO		50	else press again

10. Using Counters

The next example details using a standard counter in a scanning program section in which the STL Step structure is **not** used to avoid uncontrolled multiple INCrements of the counter. Therefore an alternative solution using a Flag is provided.

The program waits for a start button and then cycles a cylinder between the fully extended and fully retracted positions 100 times.

Without the use of the Flag, a scanning program would INCrement the counter for each **scan** of the program rather than each time the cylinder was **newly** extended.

```
STEP 1                                    initialization 1 time only
IF                          I1.0          Start button pressed
THEN      LOAD              V0
          TO                OW1           all outputs off
          RESET             F3.0          clear Flag 3.0
          LOAD              V100          initialize timer
          TO                CP0           Make a 100 cycle counter
          SET               CO            start the counter

STEP 2                                    main scanning section
IF        AND               I1.1          Cylinder is retracted
          AND               C0            and Counter 0 is active
          AND           N   F3.0          form edge detection
          AND           N   O1.0          valve to extend cylinder is off
THEN      SET               O1.0          begin extending Cylinder
          SET               F3.0          ok to look for new extend

IF                          I1.2          Cylinder is extended
          AND               F3.0          new edge
THEN      INC               CW0           count the cycle
          RESET             F3.0          update edge control
          RESET             O1.0          start retracting cylinder

IF                        N   C0          100 counts were done
THEN      JMP TO            1             start all over again
```

10. Using Counters

UP/DOWN Counters

In addition to utilizing the previously described (Standard) counters, the STL language, through the use of Multibit Operands, also allows the user to create counters.

So called Up/Down counters can be constructed using any Multibit operand such as Counter Words, Registers etc. Unlike standard counters, there is no need to initialize a counter preselect and no dedicated counter status bit exists. Likewise, the SET/RESET Counter instructions are not applicable.

The following steps are required to use this type of counter:

- Initialize the appropriate **MBO**
- The **MBO** can be INCremented or DECremented
- The **MBO** can be compared to a value or another MBO

Example: Using a Register as a Counter

In the following example a process is started and runs until 100 good parts are produced.

```
STEP 10                                          wait for start
IF                          I1.0                 Start Button
THEN        LOAD            V100                 number to produced
            TO              R50                  Register 50 is the counter
            SET             O1.1                 Switch on the Machine

STEP 20                                          look for any part at QC area
IF                      (   I1.1                 ready to check
            AND             I2.3    )            quality is good
THEN        DEC             R50                  1 less good one to make
            JMP TO          30                   continue at Step 30

IF                      (   I1.1                 ready to check
            AND         N   I2.3    )            QC ok sensor missing=bad
THEN                        NOP                  don't count bad ones

STEP 30                                          see if we have 100 yet
IF                      (   R50
                        =   V0      )            all done
THEN        RESET           O1.1                 Stop machine
            JMP TO          10                   go back to beginning
OTHRW                       NOP                  or if not done, continue

STEP 40                                          wait for last part to move
IF                      N   I1.1                 QC area clear
THEN        JMP TO          20                   continue running/testing
```

10. Using Counters

11. Using Registers

11. Using Registers

Contents

BRIEF 101

DETAILS 101

Examples 101
Using Registers in the Conditional Part of a Sentence 101
Using Registers in the Executive Part of a Sentence 101

11. Using Registers

BRIEF

This chapter explains the concept of Registers as implemented in Festo programmable controllers.

DETAILS

Festo programmable controllers which can be programmed using the STL language all possess a number of 16 bit Registers. The exact quantity of these Registers varies according to the model and can be found in Appendix A.

These Registers are Multibit Operands and can be used to store numbers in the range of:

- 0 - 65535 Unsigned Integers
- +/- 32767 Signed Integers

If the controller model you are using includes a backup battery, then the contents of the Registers will be maintained during power-off periods. Registers which have never been initialized will contain random values.

Registers are most often used in conjunction with the LOAD TO and multibit logic operations. Registers are **not** addressable on a bit by bit basis. If bit access is required, Flag Words may be more suitable (see chapter 12).

Registers may also be used to simplify controlling multiple sequential processes within a single scanned program section (see Appendix B for examples).

Examples

Using Registers in the Conditional Part of a Sentence

IF		(	R51		if the contents of Register 51
		=	V111	)	equals 111
	AND		T7		and Timer 7 is running
	AND	(	R3		and Register is smaller than
		<	R8	)	Register 8
THEN...					do whatever is programmed..

Using Registers in the Executive Part of a Sentence

IF...				programmed conditions
THEN	LOAD		R12	load the contents of Register 12 to the MBA
		+	R50	add the contents of Register 50
	TO		R45	and store the result in Register 45

11. Using Registers

12. Flags and Flag Words

12. Using Flags and Flag Words

Contents

BRIEF 105

DETAILS 105
Similarities to other Multibit Operands 105
Differences compared to other Multibit Operands 105

Examples 106
Conditional Part Examples 106
Executive Part Examples 107
Shift Registers 107

12. Using Flags and Flag Words

BRIEF

This section provides information on the logical construction and use of Flags and Flag Words in Festo programmable controllers. Appendix A provides information regarding the quantity of Flag Words, which varies according to the model of the controller.

DETAILS

Similarities to other Multibit Operands

Flag Words are, in most ways, nearly identical to Registers. Flag Words each contain 16 bits of information. When referenced as complete 16 bit units (Multibit Operands), the term **Flag Word** is applied. Within the STL language, the abbreviation **FW** is used.

Flag Words are able to store numerical data within the range:

- 0 - 65535 Unsigned Integers
- +/- 32767 Signed Integers

If the controller model you are using includes Flash or ZPRAM memory or RAM memory and a backup battery, then the contents of Flags will be maintained during power-off periods. Flags which have never been initialized will contain random values.

Flag Words do differ from other Multibit Operands in several important ways:

Differences compared to other Multibit Operands

1. A major difference between Flags and other Multibit Operands such as Registers, Counter Words, etc., is that each 16 bit Flag Word is also addressable on a per-bit basis. For example, the FPC100 contains 16 Flag Words, addressed as FW0 through FW15.

It is also possible to address individual bits (Flags) of each Flag Word by using the syntax:

F(Flag Word number)**.**Bit number

where Bit Number ranges from 0 to 15.

For example, F7.14 references Bit 14 of Flag 7.
This addressing scheme is quite similar to that used when accessing standard digital I/O points as previously described.

While Flag Words may be used with any STL instructions suitable for Multibit Operands, individual Flags are only accessible using STL instructions designed for Single Bit Operands (see chapter 4).

Single bit Flags are most often used as a convenient means to memorize events. In this respect they are similar to "internal coils or relays" often found in Ladder Diagram.

12. Using Flags and Flag Words

2. The FPC405, which supports Multiple CPU modules (Multi-processing), allows any program in any CPU to access the Flags of FW0 to FW23 (external FW) of any other CPU. That is, each CPU is able to **read from** or **write to** the Flags of any other CPU.

Therefore, Flags can provide a convenient means for implementing inter-CPU communications.

In such multiple CPU systems, each Flag Word is referenced as:

CPU number**.**Flag Word number

For example, FW2.14 references Flag Word 14 in CPU 2.

In the same manner it is also possible to address Single Bit Flags in other CPU's by extending the addressing syntax:

CPU number**.**F(Flag Word number)**.**Bit number

For example, F0.11.9 refers to Flag Bit 9 in Flag Word 11 located in CPU 0.

Examples

Individual Flags (as well as Flag Words) can be programmed in both the Conditional and Executive parts of a Sentence. In the conditional part, Flags can be interrogated as to their status (0=RESET, 1=SET); while Flags Words can be compared to values or other MBO's.

Conditional Part Examples

IF			F1.1		IF Bit 1 of Flag Word 1 is SET
IF			F2.1		IF Bit 1 of Flag Word 2 is SET
	AND	N	F4.0		and Bit 0 of Flag Word 4 is **not** SET.

Just as with all other Single and Multibit Operands, Flags may be combined with other operands.

IF		(	I3.0		If Input 3.0 is valid
	AND		F0.0	)	and Flag 0.0 is SET
	OR	((	FW3		or the value of all 16 bits of Flag Word 3
		=	V500	)	is equal to 500
	AND	N	T7	)	and Timer 7 is not active

12. Using Flags and Flag Words

Executive Part Examples

IF		I1.1	IF Input 1.1 is valid then
THEN	SET	F2.2	SET Bit 2 of Flag Word 2
IF		T6	IF T6 in local CPU is running
THEN	SET	F3.3	SET Flag 3.3 so another CPU can check T6 status
OTHRW	RESET	F3.3	

In the Executive Part of Sentences, Flag Words may be used as the source or destination of any Multibit instruction.

Shift Registers

The fact that Flags are addressable both on a Word basis as well as on a Bit basis provides a convenient method for constructing shift registers.

As an example, we may need to program a machining line in which raw castings are loaded at station 0 and subsequently various operations are to be performed at the following 15 stations. The complete machine indexes every 2 seconds and during that time a new raw casting may or may not be loaded at station 1...which can be checked by means of a sensor.

Stations 1-15 do not include sensors, but we only want each station to operate if a part is in place.

This presents an ideal situation in which a shift register can be used.

We will use Flag Word 6 to keep track of which stations contain materials to be machined. The Shift Left (SHL) instruction will be used to actually move the individual bits within the Flag Word.

The following I/O's are also used:

Input 1.0 Start Button

Input 1.1 Part Sensor at Station 0

Input 2.2 Transfer Line is indexed

Output 2.0 Indexes machining line

Outputs 1.0 - 1.15 control the machining operation at stations 0 - 15 respectively

12. Using Flags and Flag Words

STEP 10					**Start Up**
IF			I1.0		Start Button
	AND		I2.2		Line is indexed
THEN	LOAD		V200		2 seconds
	TO		TP0		to Timer 0 Preselect
	LOAD		V0		assume new production run
	TO		FW6		no parts at any station
STEP 15					**wait until some parts ready**
IF			I1.1		part was found at station 0
THEN	SET		F6.0		memorize it
IF		(	FW6		any parts to process ?
		>	V0	)	some exist !
THEN	LOAD		FW6		
	TO		OW1		turn on motors at stations with parts
	SET		T0		start process timer
STEP 20					**machining time done ?**
IF		N	T0		timer done
THEN	LOAD		V0		turn off **all** station motors
	SET		O2.0		start indexing line
STEP 25					**wait until index is started**
IF		N	I2.2		started to index
THEN	LOAD		FW6		get all stations status
			SHL		move bits to match parts
	TO		FW6		and store it
STEP 30					**is index complete ?**
IF			I2.2		new index point
THEN	RESET		O2.0		Stop index motor
	JMP TO		15		back to Step 15 for more

13. Specialized Functions

13. Specialized Functions

Contents

BRIEF 111

DETAILS 111

Analog I/O 111
Common Analog Signals 111
Common Analog Functions 112

Networking 113
Network functions 113

Position Controlling 114

Field Bus 115
An Introduction to Field Bus 115

13. Specialized Functions

BRIEF

This chapter provides some basic information concerning the following areas:

- Analog I/O
- Networking
- Position controlling
- Field Bus

Some of these functions may not apply to every controller model and may be handled in different ways depending upon the controller model.

This section is not meant to provide detailed information about these functions, but rather to explain what is available and where specific information can be found.

DETAILS

Analog I/O

As opposed to Digital I/O, in which each signal can only be on or off (1 or 0), analog signals take the form of a continuously variable signal within a pre-defined range.

Since the CPU is only able to function internally using digital signals, interfacing a PLC to either analog inputs or analog outputs requires special hardware components.

These components may be part of the controller's standard equipment (e.g. FPC101AF) or may be an optional component.

Hardware which converts the PLC's internal digital data to analog outputs is called a **D/A converter.**

Hardware which converts external analog input signals to the digital format required by the PLC internally is known as an **A/D converter.**

Common Analog Signals

There are several ranges, or types of analog signals which are popular in industrial control. If we exclude specialized analog signals of the type related to temperature control, the following common ranges remain:

- +/- 10 volts
- 0/4 to 20 milliampere current loop

13. Specialized Functions

Common Analog Functions

In order for analog inputs and outputs to be useful, the programming software must provide the means to carry out the desired functions. Basic analog functions used in industrial control include:

- setting an analog output level based on a digital value
- converting an analog input signal into a digital value

In order to effect these functions from within the PLC programming language being used, specialized CFM or FN procedures are generally used. General information regarding CFM instructions can be found in chapter 7.

Specific information covering the available functions and the proper STL language syntax is located in the appropriate hardware manual or data sheet for the product being used.

13. Specialized Functions

Networking

In the context of this manual and the STL language, Networking refers to the hardware and software that provide the means to interconnect control systems that are otherwise independent units.

Networking is typically employed to connect various elements of a distributed processing system in which each sub-system controls a specific section (physical or logical) of the overall task. Using a network allows these sections to be combined in an orderly manner.

Regardless of the programming language being used, specialized hardware as well as network software is required to implement a network.

Depending upon the controller model, this specialized hardware may take the form of a network processor or module that contains the specialized software routines which can be accessed by the STL language.

Network Functions

Typical functions which must be performed on a network include:

- Initialization of Network stations
- Request another station to execute a command
- Management of Network transmissions

The interface between the STL language and the specialized network software is effected by means of the CFM instruction. General information regarding the CFM instruction can be found in chapter 7.

Specific information explaining the available network software calls for a specific controller model can be found in the appropriate hardware manual or data sheet.

13. Specialized Functions

Position Controlling

It may be necessary to quickly and accurately control the position of mechanical components as part of a control system. Such movements are generally accomplished by using various types of motors.

Depending on the requirements of:

- Speed
- Accuracy
- Cost effectiveness
- Reliability

several types of motors are available. These choices include stepper motors and servo motors as well as multiple speed motors which may incorporate braking components.

Additional variations may or may not incorporate closed loop control.

The more accurate (and expensive) solutions such as servo motors usually incorporate dedicated microprocessors; while less sophisticated solutions may rely fully upon the speed and intelligence of the programmable controller.

Because of the wide variety of positioning sub-systems that may be connected, there are no dedicated STL instructions for positioning. However, Festo can supply specialized programs and program modules that have been optimized and/or customized for position control.

In addition, Festo offers several dedicated positioning controllers. You should refer to the manuals and data sheets for the respective hardware for further information regarding the availability of specific software modules.

13. Specialized Functions

Field Bus

An Introduction to Field Bus

The Festo Field Bus is based upon the RS485 electrical standard which defines the parameters of a high speed serial bus structure.

A differentiation must be made between the elements which form the controller's internal bus structure and the Field Bus system.

Standard I/O's are closely connected, both electrically and physically, to the controller's internal parallel bus structure. While this structure provides very high speed access, its nature places hardware limits upon the number of unique addressable I/O possible.

The Field Bus concept utilizes the aforementioned serial bus to link one Master and Multiple slave stations at transmission rates up to 375,000 bits per second.

Since these stations can be located over relatively long distances (300-1200 meters) they are often generically referred to as "Remote I/O".

The high transmission rates, when combined with the cost savings of using simple twisted-pair cable for bus wiring, makes the Field Bus concept very attractive.

Inputs and Outputs located at Field Bus slave stations can be interrogated and controlled by the Field Bus master station. The STL language allows access to these I/O's using the same **SET** and **RESET** instructions as standard digital I/O (see chapter 8).

To accommodate the extended configuration possible using the Field Bus, the I/O syntax has also been extended. Inputs and Outputs are therefore referenced as follows:

Ipa[.m.s] for Inputs

Opa[.m.s] for Outputs

where: p = System address of Field Bus master processor

a = Field Bus slave station address [1... 99]

m = optional module address [0... 15]

s = optional stage number [0...15]

Please refer to the appropriate hardware manual for specific details.

13. Specialized Functions

Appendix A - Operands

Appendix A - Operands

Contents

BRIEF 119

DETAILS 119

Appendix A - Operands

BRIEF

This section provides an overview of the available range of Operands in each controller model. The reader should be aware that the information provided refers to the operands that are available when programming in the **STL** language.

The number of operands may vary depending upon the programming language used. This listing should only be used as a guide. You should refer to the appropriate hardware and FST manual for any possible changes.

DETAILS

The following table includes those operands most often used when programming in **STL**. For models which allow multiple CPU's the quantities shown are **per CPU.**

	Controller Model			
Operands	**FPC100**	**FPC405**	**FEC**	**IPC**
Counters	16	64	256	256
Timers	32	64	256	256
Flags/ Flag Words	256/ 16 x 16 bit	1024/ 64 x 16 bit	160.000/ 10.000 x 16 bit	160.000/ 10.000 x 16 bit
Registers	64	128	256	256
Multi-tasking (max. tasks)	yes 8	yes 64	yes 64	yes 64

Appendix A - Operands

Appendix B - Sample Programs

Appendix B - Sample Programs

Contents

BRIEF 123

DETAILS 123

Examples 123
Sample 1. Completely Sequential 123
Sample 2. Mostly Sequential with Random events 125
Sample 3. Completely Random events 128
Sample 4. Multiple sequences & Random events 129

Appendix B - Sample Programs

BRIEF

This section presents several sample control problems and solutions using the STL language. The samples presented are generalized to be useful to the reader regardless of which controller model is being used.

If the controller model being programmed supports Multitasking Appendix C should be consulted.

DETAILS

Most control tasks can be divided into 3 categories:

- Completely sequential
- Mostly sequential with some random events
- Completely random

In addition, many situations arise in which it may be necessary to control several sequences simultaneously. The following examples will present solutions for all of the above possibilities.

Examples

Sample 1: Completely Sequential

Tasks which are completely sequential are ideally suited to the STL language because of the implicit Step structure. The sequential task is to control the movement of 3 pneumatic cylinders by means of 3 3/2 solenoid valves in a defined sequence.

When power is applied to the system and the Start Button is pressed, cylinder A is to extend fully for 3 seconds and then retract.

Next cylinder B is to extend fully and retract 4 times and then fully extend and remain extended.

Finally, cylinder C is to extend completely, at which time cylinder A will extend. After Cylinder A is again fully extended, all three cylinders will retract and wait for the Start button.

The following allocations apply:

Input 1.0	Start Button
Input 1.1	Cylinder A retracted
Input 1.2	Cylinder A extended
Input 1.3	Cylinder B retracted
Input 1.4	Cylinder B extended
Input 1.5	Cylinder C retracted
Input 1.6	Cylinder C extended
Output 1.0	Cylinder A extend solenoid
Output 1.1	Cylinder B extend solenoid
Output 1.2	Cylinder C extend solenoid

Appendix B - Sample Programs

STEP 1				**power up initialization**
IF			NOP	always do this
THEN	LOAD		V0	unconditionally switch off all
	TO		OW1	Outputs
	LOAD		V300	Prepare Timer 0 as a 3 second timer
	TO		TP0	units = 0.01 seconds
	LOAD		V4	Prepare Counter 2
	TO		CP2	
STEP 5				**be certain all positions ok**
IF			I1.0	Start Button is pressed
	AND		I1.1	Cylinder A is retracted
	AND		I1.3	Cylinder B is retracted
	AND		I1.5	Cylinder C is retracted
THEN	SET		O1.0	begin extending cylinder A
STEP 10				**Cylinder A fully extended ?**
IF			I1.2	now it's fully extended
THEN	SET		T0	start the 3 second timer
STEP 12				**wait 3 seconds**
IF		N	T0	timer is complete
THEN	RESET		O1.0	begin retracting cylinder A
STEP 15				**Cylinder A fully retracted ?**
IF			I1.1	Cylinder A is retracted
THEN	SET		C2	setup counter 2 - 4 counts
	SET		O1.1	begin extending cylinder B
STEP 20				**Cylinder B fully extended ?**
IF			I1.4	now it's fully extended
THEN	INC		CW2	count this cycle
	RESET		O1.1	begin retracting cylinder B
STEP 22				**is this the 4th extension ?**
IF			I1.3	Cylinder B retracted and 4
	AND		C2	strokes not done
THEN	SET		O1.1	begin extending cylinder B
	JMP TO		20	continue cycles
IF			I1.3	Cylinder B retracted and 4
	AND	N	C2	strokes are done
THEN	SET		O1.1	begin extending cylinder B

Appendix B - Sample Programs

STEP 30			**Cylinder B fully extended ?**
IF		I1.4	Cylinder B fully extended 5 x
THEN	SET	O1.2	begin extending cylinder C
STEP 35			**Cylinder C fully extended ?**
IF		I1.6	Cylinder C fully extended
THEN	SET	O1.0	Begin extending cylinder A
STEP 40			**All cylinders extended ?**
IF		I1.2	cylinder A fully extended too
THEN	RESET	O1.0	retract Cyl. A
	RESET	O1.1	retract Cyl. B
	RESET	O1.2	retract Cyl. C
	JMP TO	5	go back to Step 5

Sample 2: Mostly Sequential with Random events

While some simple machinery may be completely sequential in operation, there may be one or more exceptions which change the classification of the task so that it is no longer totally sequential.

If the majority of the control task is sequential and the controller model allows Multitasking (see Appendix A), a possible solution may be to divide the sequential and random event processing into separate programs (see Appendix C).

However it is also possible to handle such situations with a single STL program. If the random event(s) to be monitored are few and the balance of the program is relatively simple, then it may be possible to handle the requirements by adding a program Sentence in **every** Step.

Other possible solutions include the use of interrupt processing (only supported on some controller models) or by constructing the entire sequence as a parallel (scanning) program section. This method will be demonstrated in samples 3 and 4.

Sample 2 will illustrate inserting a program Sentence in every existing Step of the program presented in Sample 1 as a means of detecting and responding to a simple "**pause"** push button; which when pressed, results in the program being suspended until it is released.

Appendix B - Sample Programs

STEP 1				**power up initialization**
IF			NOP	always do this
THEN	LOAD		V0	unconditionally switch off all
	TO		OW1	Outputs
	LOAD		V300	Prepare Timer 0 as a 3 second timer
	TO		TP0	units = 0.01 seconds
	LOAD		V4	Prepare Counter 2
	TO		CP2	
STEP 5				**be certain all positions ok**
IF			I1.0	Start Button is pressed
	AND		I1.1	Cylinder A is retracted
	AND		I1.3	Cylinder B is retracted
	AND		I1.5	Cylinder C is retracted
	AND	N	I1.7	pause button not active
THEN	SET		O1.0	begin extending cylinder A
STEP 10				**Cylinder A fully extended ?**
IF			I1.7	pause button
THEN	JMP TO		10	if so stay here
IF			I1.2	now it's fully extended
THEN	SET		T0	start the 3 second timer
STEP 12				**wait 3 seconds**
IF			I1.7	pause button
THEN	JMP TO		12	if so stay here
IF		N	T0	timer is complete
THEN	RESET		O1.0	begin retracting cylinder A
STEP 15				**Cylinder A fully retracted ?**
IF			I1.7	pause button
THEN	JMP TO		15	if so stay here
IF			I1.1	Cylinder A is retracted
THEN	SET		C2	setup counter 2 - 4 counts
	SET		O1.1	begin extending cylinder B
STEP 20				**Cylinder B fully extended ?**
IF			I1.7	pause button
THEN	JMP TO		20	if so stay here
IF			I1.4	now it's fully extended
THEN	INC		CW2	count this cycle
	RESET		O1.1	begin retracting cylinder B

Appendix B - Sample Programs

STEP 22				**is this the 4th extension ?**
IF			I1.7	pause button
THEN	JMP TO		22	stay here
IF			I1.3	Cylinder B retracted and 4
	AND		C2	strokes not done
THEN	SET		O1.1	begin extending cylinder B
	JMP TO		20	continue cycles
IF			I1.3	Cylinder B retracted and 4
	AND	N	C2	strokes are done
THEN	SET		O1.1	begin extending cylinder B
STEP 30				**Cylinder B fully extended ?**
IF			I1.7	pause button
THEN	JMP TO		30	if so stay here
IF			I1.4	Cylinder B fully extended 5 x
THEN	SET		O1.2	begin extending cylinder C
STEP 35				**Cylinder C fully extended ?**
IF			I1.7	pause button
THEN	JMP TO		35	
IF			I1.6	Cylinder C fully extended
THEN	SET		O1.0	Begin extending cylinder A
STEP 40				**All cylinders extended ?**
IF			I1.7	pause button
THEN	JMP TO		40	if so stay here
IF			I1.2	cylinder A fully extended too
THEN	RESET		O1.0	retract Cyl. A
	RESET		O1.1	retract Cyl. B
	RESET		O1.2	retract Cyl. C
	JMP TO		5	go back to Step 5

In summary, it is possible to handle limited amounts of parallel conditions within an otherwise strictly sequential process using the Step instruction.

Appendix B - Sample Programs

Sample 3: Completely Random Events

Some control situations cannot be organized in any logical sequence as the operations may occur in any random order. A typical example of such a task might be the **Setup** control program for a machine. The operation is defined by the machine operator pressing various push buttons, each of which is used for a single function.

The following depicts the Setup program for a plastic injection molding machine.

STEP 10				**initialization**
IF			NOP	always true
THEN	LOAD		V0	
	TO		OW1	Turn off all outputs
STEP 20				**scanning step**
IF			I1.0	Mold Close Push Button
THEN	SET		O1.0	Close Mold
IF			I1.1	Inject Plastic Push Button
	AND		I2.0	Mold Closed Sensor
THEN	SET		O1.3	Injection Solenoid
OTHRW	RESET		O1.3	
IF			I1.2	Mold Open Push button
	AND	N	O1.3	Injection not active
THEN	RESET		O1.0	Open Mold
IF			I1.3	Rotate Screw mechanism
THEN	SET		O1.1	Screw Rotate Solenoid
OTHRW	RESET		O1.1	Halt Screw mechanism
IF			I1.4	Mold fully Open Sensor
	AND		I1.5	Mold Ejector Push Button
THEN	SET		O1.4	Mold Ejector Solenoid
OTHRW	RESET		O1.4	Halt Ejection process
IF			NOP	always do
THEN	JMP TO		20	keep processing

Appendix B - Sample Programs

Sample 4: Multiple Sequences & Random Events

The final sample program combines many of the elements that have been presented in this manual along with several new topics.

It should be mentioned that if the controller model being programmed supports Multitasking that Appendix C should be consulted.

Multi-Station Rotary Table Machine

The following STL program is used to control a 4 station rotary table machine in which each station must perform its own sequence concurrently with all other stations. The various stations each have different numbers of steps associated with their individual functions.

The operator requires the ability to activate or deactivate any station. After all stations have completed their respective sequences, the rotary table will index and begin a new cycle. In addition, a Flag Word will be used as a Shift Register to determine which stations should operate based upon the presence of a part in place.

The overall process will be controlled by means of assigning a Register to each station. While individual Flags might be used, the use of Registers greatly enhances machine diagnostics should a stoppage occur.

This method of program structure allows as many parallel processes to be active as the number of Registers and further permits more than 65000 steps per process.

The operation of the machine is as follows:

Station 1: Station 1 is used to load empty ribbon cartridges. If no cartridge is present at this station, but cartridges are present at stations 2,3 or 4, then those stations will operate. When the machine indexes, the status of each station (part to process: yes/no) will be updated.

Station 2: Station 2 consists of several sequential events which insert two empty spools into the ribbon cartridge.

Station 3: Station 3 performs several steps in which a length ribbon is attached to the left hand spool, the ribbon is then fully wound on to the left hand spool and finally secured to the right hand spool.

Station 4: Station 4 fits the upper half of the ribbon cartridge and attaches it to the lower half by ultrasonic welding. Finally, the finished cartridge is ejected into a packing box..

Appendix B - Sample Programs

```
STEP 10                                          initialization
THEN    LOAD          V0
        TO            OW0        turn off all outputs
        TO            OW1
        TO            OW2
        TO            OW3
        TO            OW4
        TO            FW0        initialize "shift register"
        TO            R0         Table Index Control Register
        TO            R1         Station 1 Control Register
        TO            R2         Station 2 Control Register
        TO            R3         Station 3 Control Register
        TO            R4         Station 4 Control Register
        LOAD          V25
        TO            TP2        Timer 2 as 1/4 second
        LOAD          V250
        TO            TP3        Timer 3 as 2.5 seconds
        LOAD          V300
        TO            TP4        Timer 4 as 3 seconds

STEP 20                                          all stations home ?
IF              N     I0.0       E_Stop Active
THEN    JMP TO        99         Special Routine

IF                    I0.2       table is indexed
        AND           I2.1       Station 2 Left insert Cyl. Ret.
        AND           I2.3       Station 2 Right insert Cyl. Ret.
        AND     N     I2.5       Station 2 Left spool  in place
        AND     N     I2.6       Station 2 Right spool in place
        AND           I3.1       Station 3 pinch ribbon gripper
                                 open
        AND           I3.4       Station 3 Ribbon Advance Cyl.
                                 is retracted
        AND           I3.6       Station 3 Right side ribbon
                                 attach cylinder retracted
        AND           I4.1       Station 4  insertion cyl. retr'd.
                N     I4.3       no prior top half in place
        AND           I4.4       Station 4 Eject. Cyl. Home
        AND           I0.1       Run Switch to Run position
        AND           I0.0       E.Stop not Active
        AND     (     I1.1       Cartridge at Station 1
                                 or parts at some stations
        OR      (     FW0        entire 16 bit word
        AND           V15        mask all but bits 0,1,2,3
                >     V0    ))   then an active part exists
THEN                  NOP        Ok to proceed, else wait
```

Appendix B - Sample Programs

```
STEP 30                                          STATION 1 first
IF                  (   R1                       station 1 control register
                    =   V0          )            just starting
          AND       (   R2                       station 2 control register
                    =   V255        )            indicates it's done
          AND       (   R3                       station 3 control register
                    =   V255        )            indicates it's done
          AND       (   R4                       station 4 control register
                    =   V255        )            indicates it's done
THEN      LOAD          V10                      all other stations are done, so
          TO            R1                       correct time to see if a part
                                                 was loaded into Station 1

IF                  (   R1                       Station 1 control register
                    =   V10         )            ready to read sensor
          AND           I1.1                     part in place sensor
THEN      SET           F0.0                     place a '1' in 'shift register'

IF                  (   R1                       when we are here, ALL
                    =   V10         )            stations are done
THEN      LOAD          V255
          TO            R1

                                                 STATION 2 section
IF                  (   R2                       Station 2 control register
                    =   V0          )
          AND       (
                    N   I2.0                     Station 2 not activated
          OR        N   F0.1        )            or no parts in Station 2
THEN      LOAD          V255                     so mark Station 2 as done
          TO            R2

IF                  (   R2                       Station 2 control register
                    =   V0          )            just starting
THEN      SET           O2.0                     extend Left Side Spool Cyl.
          SET           O2.1                     extend Right Side Spool Cyl.
          LOAD          V20
          TO            R2                       advance control sequence

IF                  (   R2                       Station 2 control register
                    =   V20         )
          AND           I2.2                     Left side fully extended
          AND           I2.4                     Right side fully extended
          AND           I2.5                     Left Spool in fixture
          AND           I2.6                     Right Spool in fixture
THEN      SET           O2.2                     Switch on holding vacuum
          SET           T2                       Start Timer
          LOAD          V30
          TO            R2                       Update Station 2 control Reg.
```

Appendix B - Sample Programs

```
IF              (   R2          Station 2 control register
                =   V30     )
          AND   N   T2          1/4 sec. dwell time complete
THEN      RESET     O2.0        retract Left Side Spool Cyl.
          RESET     O2.1        retract Right Side Spool Cyl.
          LOAD      V40
          TO        R2          Update Station 2 control Reg.

IF              (   R2          Station 2 control register
                =   V40     )
          AND       I2.1        Left Side Spool Cyl. is home
          AND       I2.2        Right Side Spool Cyl. is
                                home
THEN      RESET     O2.2        Switch vacuum off
          LOAD      V255
          TO        R2          Mark station 2 as complete

                                STATION 3 section
IF              (   R3          Station 3 control register
                =   V0      )
          AND   (
                N   I3.0        Station 3 not activated
          OR    N   F0.2    )   or no parts in Station 3
THEN      LOAD      V255        so mark Station 3 as done
          TO        R3

IF              (   R3          Station 3 control register
                =   V0      )
THEN      SET       O3.1        ribbon gripper close solenoid
          LOAD      V10
          TO        R3          Update Station 3 control Reg.

IF              (   R3          Station 3 control register
                =   V10     )
          AND       I3.2        gripper fully closed
THEN      SET       O3.2        insert ribbon in left spool
          LOAD      V30
          TO        R3          Update Station 3 control Reg.

IF              (   R3          Station 3 control register
                =   V30     )
          AND       I3.3        Ribbon is inserted into spool
THEN      RESET     O3.2        retract insertion cylinder
          RESET     O3.1        release ribbon gripper
          LOAD      V40
          TO        R3          Update Station 3 control Reg.
```

Appendix B - Sample Programs

```
IF              (   R3      Station 3 control register
                =   V40     )
        AND         I3.4    insertion cylinder is home
        SET         O3.3    start ribbon winding motor
        SET         T3      start winding timer
        LOAD        V50
        TO          R3      Update Station 3 control Reg.

IF              (   R3      Station 3 control register
                =   V50     )
        AND     N   T3      winding time is complete
THEN    RESET       O3.3    halt winding motor
        SET         O3.4    Rt. Side Ribbon Insertion Cyl.
        LOAD        V60
        TO          R3      Update Station 3 control Reg.

IF              (   R3      Station 3 control register
                =   V60     )
        AND         I3.5    Right Spool insertion sensor
THEN    RESET       O3.4    Retract Rt. Side Insertion Cyl.
        LOAD        V70
        TO          R3      Update Station 3 control Reg.

IF              (   R3      Station 3 control register
                =   V70     )
        AND         I3.6    Rt. Side Insertion Cyl.= home
THEN    LOAD        V255
        TO          R3      mark Station 3 as complete

                            STATION 4 section
IF              (   R4      Station 4 control register
                =   V0      )
        AND     (
                N   I4.0    Station 4 not activated
        OR      N   F0.3    )   or no parts in Station 4
THEN    LOAD        V255    so mark Station 4 as done
        TO          R4

IF              (   R4      Station 4 control register
                =   V0      )
THEN    SET         O4.1    Lower upper cartridge
        LOAD        V10
        TO          R4      Update Station 4 control Reg.
```

Appendix B - Sample Programs

```
IF                  (   R4      Station 4 control register
                    =   V10     )
          AND           I4.2    Cartridge cylinder extended
          AND           I4.3    Cartridge fully in fixture
THEN      SET           O4.2    Start Ultrasonic bonding
          SET           T4      Start welding timer
          LOAD          V20
          TO            R4      Update Station 4 control Reg.

IF                  (   R4      Station 4 control register
                    =   V20     )
          AND       N   T3      welding Timer complete
THEN      RESET         O4.2    halt welding
          RESET         O4.1    unclamp up cartridge Cyl.
          LOAD          V30
          TO            R4      Update Station 4 control Reg.

IF                  (   R4      Station 4 control register
                    =   V30     )
          AND           I4.1    Upper shell Cyl. is home
THEN      SET           O4.3    Extend Ejection Cylinder
          LOAD          V40
          TO            R4      Update Station 4 control Reg.

IF                  (   R4      Station 4 control register
                    =   V40     )
          AND           I4.5    Ejection Cylinder extended
THEN      RESET         O4.3    Retract Ejection Cylinder
          LOAD          V50     Update Station 4 control Reg.

IF                  (   R4      Station 4 control register
                    =   V50     )
          AND           I4.4    Ejection Cylinder is home
THEN      RESET         F0.3    Empty Position in Shift Reg.
          LOAD          V255
          TO            R4      mark station 4 as complete

IF                  (   R1      Stations 1-4 done
                    =   V255    )
THEN      LOAD          V10     Index Control Register
          TO            R0


                                TABLE INDEX section
IF                  (   R0      Index Control Register
                    =   V10     )
          AND       ((  FW0     complete 16 bit unit
          AND           V15     )   mask all except bits 0,1,2,3
                    >   V0      )   at least 1 station occupied
THEN      LOAD          V20     Update Index Control Reg.
          TO            R0
```

Appendix B - Sample Programs

```
IF                  (   R0      Index Control Register
                    =   V10     ) no index required
THEN    JMP TO          10      Continue Process

IF                  (   R0      Index Control Register
                    =   V20     ) an index is needed
THEN    SET             O0.0    Begin table index
        LOAD            V30     Update Index Control Reg.

IF                  (   R0      Index Control Register
                    =   V30     )
        AND         N   I0.2    indexing underway
THEN    LOAD            V40
        TO              R0      Update Index Control Reg.
        LOAD            FW0     Load Shift Register to MBA
        SHL                     Shift bits left to match actual
        TO              FW0     parts

                                **Index Sequence complete**
IF                  (   R0      Index Control Register
                    =   V40     )
        AND             I0.2    New index found
THEN    RESET           O0.0    halt indexing
        LOAD            V0      clear control registers
        TO              R0
        TO              R1
        TO              R2
        TO              R3
        TO              R4
        JMP TO          20      Resume processing

IF              N       I0.0    E_Stop Active
THEN    JMP TO          99      Special Routine

                                **Continue scanning**
IF                      NOP     unconditionally continue to
THEN    JMP TO          30      process Step 30

STEP 99                         **ESTOP ROUTINE**
IF                      I0.0    wait E_Stop until is released
THEN    JMP TO          10      & handle like power-up
```

Appendix B - Sample Programs

Appendix C - Multitasking ...

Appendix C - Multitasking ...

Contents

BRIEF 139

DETAILS 139

General Concepts 139
Multitasking 139

Assigning Programs 140
FEC 140
FPC100B/AF 140
FPC405 140
IPC 140
Using Multitasking 141

Examples 141

Multiprocessing 142

Appendix C - Multitasking ...

BRIEF

This section provides information regarding the concept, definition, purpose and structure of **Multitasking and Multiprocessing** in Festo programmable controllers.

Please refer to Appendix A of this document which details which controller models support Multitasking. Multiprocessing is only applicable to the FPC405.

DETAILS

General Concepts

Multitasking

All Festo programmable controllers allow storing multiple programs in memory. Additionally, all models provide the ability to concurrently execute more than one program by means of multitasking.

Modern CPU's are constructed using high speed microprocessors. These devices generally are only able to perform a single function at any point in time. However, since microprocessor operation is so rapid (millions of operations per second), it is possible to divide the available processing time into multiple parts and assign a specific task to each of these parts.

When a single physical processor's power is divided in such a manner, the resulting parts are often referred to as Virtual Processors.

By means of the controller's internal operating software and the available programming language instructions, it is possible to assign multiple programs to be processed in such a task-swapping manner.

In order to function properly, a tightly defined set of rules must exist that determine how and when each task (program) will be processed.

When multiple STL programs are executing on a single CPU the following rules apply:

1. A program, in turn, is allocated processor resources.

2. A complete program Step (or complete program if no Steps have been used) is processed Sentence by Sentence.

3. If the Conditional part of any Sentence is true, the Executive part of the Sentence is performed. See chapter 5 for more details.

Appendix C - Multitasking ...

4. When the **last** Sentence of a Step has been processed, regardless of whether or not the Conditional part is true, the CPU will save the current program's status and...

5. If other programs are currently assigned, the next program will be made active and will be processed (1-4 above).

This "circular" process continues until the original program is again allocated processor resources.

Assigning Programs

When a controller is supplied with operating power, and assuming the required dedicated system hardware inputs (e.g. RUN etc.) are supplied with the appropriate signal levels, processing will begin:

FEC: Program 0 will begin processing. The FEC allows up to 64 STL programs to be processed concurrently. Programs can be started by means of the **SET Pn** instruction and stopped by means of the **RESET Pn** instruction.

FPC100B/AF: Program 0 will begin processing. By using function module CFM2, up to 8 additional STL program modules can be processed concurrently to P0.

FPC405: Program 0 will begin processing. The FPC405 allows up to 64 STL programs to be processed concurrently in each CPU. Programs can be started by means of the **SET Pn** instruction and stopped by means of the **RESET Pn** instruction.

IPC: Program 0 will begin processing. The IPC allows up to 64 STL programs to be processed concurrently. Programs can be started by means of the **SET Pn** instruction and stopped by means of the **RESET Pn** instruction.

Appendix C - Multitasking ...

Using Multitasking

The ability to store multiple programs in the controller's memory in conjunction with multitasking capabilities, provides a wide array of organizational possibilities to solve complex control tasks.

Examples

A typical machine might have the following requirements:

1. Manual operation
2. Automatic cycle

During operation, in addition to some sequential tasks, there is also the need to continuously monitor functions such as Emergency Stop, Stop, Watch Dog Timer, and home position etc.

These tasks might be solved by dividing the overall control requirements into easily manageable parts:

Program 0: Performs any required power-up initialization and acts as a dispatcher program to start and stop other programs depending on the desired operation. This program also provides the continuous monitoring functions (e.g. Emergency Stop).

Program 1: This program provides the logic required for manual operation. In addition, by means of Flags (see chapter 12), this program is able to check the physical status of the machine as determined by program 3.

Program 2: This program provides the logic required for automatic operation. In addition, by means of Flags (see chapter 12), this program is able to check the physical status of the machine as determined by program 3.

Program 3: This program constantly monitors the physical status of various machine parts, and based upon their positions Sets or Resets Flags which can then be read by other programs. This often eliminates duplicate program logic.

In this example, the following programs would be active depending upon the mode of operation:

Manual Mode	Automatic Mode
Program 0	Program 0
Program 1	Program 2
Program 3	Program 3

Multiprocessing

Multiprocessing is possible in systems which employ multiple CPU's. When multiple CPU's are used, true **concurrent** processing of multiple programs is possible in addition to the

Appendix C - Multitasking ...

facilities provided by using Multitasking.

Appendix C - Multitasking ...

Appendix D - Binary Numbers

Appendix D - Binary Numbers

Contents

BRIEF 145

DETAILS 145

Decimal Numbers 145
Binary Numbers 145

Appendix D - Binary Numbers

BRIEF

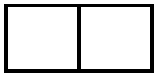
This section provides basic information on the relationship between decimal and binary number formats. This concept is valuable if the reader wishes to make full use of the multibit arithmetic and logic instructions provided in the STL language.

DETAILS

The Multibit Operands (MBO) used in Festo programmable controllers are either 8 or 16 bits in width. Since the width of each MBO is fixed, the range of values that can be stored is also fixed.

Decimal Numbers

In everyday life we use the decimal number system for nearly every function. The mathematical rules which apply to all numbering systems are often overlooked. For example, if the following form was provided, along with instructions to "Enter your age in years:"

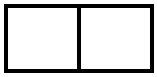


and we were informed to enter only (1) one digit per box, we would all quickly realize that the maximum age that could be entered would be 99. In total, we would be able to make 100 different entries, ranging from 0 to 99.

This is possible as each box is able to accept any one of 10 possible entries (0-9).

Binary Numbers

In the world of digital computers, binary format is very common due to technical reasons. If the previous question were rewritten to read "Enter your age in **binary** years:" and the same two boxes were provided:



then the maximum age that could be entered would be 3 decimal or "11" binary. Therefore, a total of only 4 different entries (0-3 decimal) would be possible because only a '0' or '1' could be entered in each box.

Appendix D - Binary Numbers

The following combinations and their decimal equivalents exist:

0	0	= 0 decimal
0	1	= 1 decimal
1	0	= 2 decimal
1	1	= 3 decimal

Just as in the decimal format, it can be seen that each box, or column has a certain weighted value. In decimal numbers we refer to these columns as:

the "one's column"
the "ten's column"
the "hundred's column"

The equivalent descriptions when working with binary numbers would be:

the "one's column"
the "two's column"
the "four's column" etc.

To convert a value between binary and decimal formats it is necessary to know the weighted value of each column or position. Assuming that we are working with unsigned integers the following values can be stored:

8 bit MBO range 0 - 255 decimal

16 bit MBO range 0 - 65535

Appendix D - Binary Numbers

Numerous conversion charts and inexpensive calculators are available to assist in the conversion process. The following table provides basic information regarding the weighted value of each column of a 16 bit binary number.

15							Bit number								0	decimal value
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1
0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	2
0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	4
0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	8
0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	16
0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	32
0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	64
0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	128
0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	256
0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	512
0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1024
0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	2048
0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	4096
0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	8192
0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	16384
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	32768

Appendix D - Binary Numbers

Index

Index

A
Analog 111-112
AND 27, 37, 41-42

B
BID 37, 43
Binary 37, 43, 145

C
CCU 10
CFM 37, 44-45
CMP 37, 46-47
Control registers 129
Counters
  Initializing 92
  Preselect 91
  Standard 91
  Starting 93
  Status 91, 93
  Stopping 93
  Updating 93
  UP/DOWN 97
  Word 91-92
CPL 37, 48
CPU 10

D
DEB 37, 49
DEC 37, 50
Decimal 49, 145

E
Edge triggering 30
EXOR 37, 51-52

F
Field Bus 77, 115
Flags 105-107
FST 9

I
IF 26-27, 37
INC 37, 53
Inputs 78-79
Installation 13
I/O 77-78
INV 37, 54

J
JMP 33, 37, 55-57

L
Ladder Diagram
  Comparison to 28
Language
  Elements of 26
  Structure 25
LOAD 37, 58-60

M
MBA 17
MBO 17
Motors 114
Multibit Operands 17
  Listing of 19
Multiprocessing 142
Multitasking 115, 139, 140, 141

N
Network 77, 111, 113
NOP 32, 37, 61-62
NOT 21, 27

O
On-line Mode 14
Operands 17
  Absolute 17
  Global 20
  Listing by model 119
  Local 20
  Multibit 19
  Single Bit 18
  Symbolic 17
Operators 21
OR 27, 37, 63-64
OTHRW 34, 38
Outputs 80

P
Parallel processing 30,61
Positioning 114
Program 10
  Creating 13
  Execution 26-29
  Loading 14
  Samples 123-136
  Starting 140
  Structure 25-34, 123
  Types 123
  Version 13
  Writing 14
Project 10
PSE 38, 65

R
Random events 125, 128
Registers 101, 129
RESET 38, 66, 85
ROL 38, 67
ROR 38, 68

S
Sample programs 123-136
SBA 17
SBO 17
Scanning 30, 61
Sentences 26-31
Sequential task 123
SET 38, 69
SHIFT 38, 70
Shift Register 107
SHL 38, 71-72
SHR 38, 73
Single Bit Operands 18
  Listing of 18
STEP instruction 26, 27, 29
  Conditional Part 26
  Execution rules 30-31
  Executive Part 26
  Label 26, 29
SWAP 38, 74

T
THEN 26, 38
Timers 83, 87
  Clock rate 84
  Preselect 83, 84
  Resetting 85-86
  Starting 85
  Status 83, 85
  Stopping 85
  Word 83
TO 38