Socapel PAM

A Programmable Axes Manager

PAM System V2.5

Reference Manual

Ordering Number: 006.8007.B

Rev. October 15, 1995

This upgraded and improved version replaces all the previous. We reserve the right to amend this document without prior notice and decline all responsibilities for eventual errors.

Atlas Copco Controls SA
En Montillier 4
CH-1303 PENTHAZ
Switzerland

Doc. No. 006.8007.B/PB-TK

TABLE OF CONTENTS

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

TABLE OF FIGURES

1 INTRODUCTION

1.1 INTRODUCTION TO THE PAM REFERENCE MANUAL

1.1.1 SCOPE OF PAM REFERENCE MANUAL

This manual is intended to serve as a reference for individuals who will be designing and writing motion control application programs for the PAM (Programmable Axes Manager). This manual presents details on the syntax, structure and use of the declarations and statements which comprise the PAM application language. It also provides descriptions of the PAM physical and application objects and details on applying them. Examples are used to illustrate concepts and syntax structures.

The scope of this manual is limited to the PAM applications programming language and the software interfaces to other PAM peripherals. For a complete listing of all technical publications covering PAM and it's associated peripherals (i.e. ST1, Smart I-O, VME Bus Master and Simatic S5), refer to the Technical Publications Overview in the PAM User's Manual (document 006.8017.A).

1.1.2 MANUAL ORGANISATION AND CONTENTS

This manual is organised into fifteen chapters and seven appendices. Each chapter covers one or more types of related objects or statements. Most object and statement descriptions include examples which illustrate proper syntax or use for the object/statement. Where appropriate, supplemental information or precautions regarding the use an item are included in the text. The following is a summary of the manual contents by chapter:

Chapter 1	description of manual organisation and contents some basic definitions and concepts symbols, abbreviations and conventions used in defining various syntactic structures
Chapter 2	Descriptions, declaration syntax, summary of available functions and declaration examples for physical objects
Chapter 3	Descriptions, declaration syntax, summary of available functions and declaration examples for application objects
Chapter 4	Descriptions, syntax, usage examples for all statement types except mathematical statements
Chapter 5	Concept of pipes Rules for creating activating and de-activating pipes Description, intended use, declaration syntax, summary of available functions, declaration and usage examples for all pipe block types
Chapter 6	Mathematical functions, operators and expressions
Chapter 7	Descriptions, syntax and examples of physical and application object executive functions
Chapter 8	Descriptions, syntax and examples of pipe block executive functions
Chapter 9	Descriptions, syntax and examples of physical and application object inquire functions
Chapter 10	Descriptions, syntax and examples of pipe block inquire functions
Chapter 11	Functions used to control LEDs, seven segment displays and the PamDisplay Uses of the PamDisplay by applications

In addition to this summary of manual contents, the Table of Contents and Index will be helpful for locating information on specific topics.

1.1.3 WHAT'S NEW IN VERSION 2.5

The following is a summary of the major changes in PAM software version 2.5:

1.1.3.1 PIPE BLOCKS

- The Phaser has been added
- For the Corrector, **CORRECTION_MODE** , **VALUE_PERIOD | VALUE_RANGE**, and **CORRECTION_REFERENCE** parameters have been added.
- For the Comparator, the **REVERSE_ROUTINE** parameter has been added.
- For the Amplifier, **GAIN_SLOPE** and **OFFSET_SLOPE** parameters have been added.
- For the TMP Generator, a new **ANTI_DELAY** parameter has been added. The **POSITION** parameter has been renamed **INITIAL_POSITION** and it's access level is (RW). The **PERIOD** parameter access level is now (RO). The **POSITION_PERIOD** parameter access level is now (RW).
- For the PMP Generator, a new **ANTI_DELAY** parameter has been added. The **POSITION** parameter has been renamed **INITIAL_POSITION** and it's access level is (RW). The **PERIOD** parameter access level is now (RO).
- For the Sampler, the **PERIOD** parameter access is now (RO).

1.1.3.2 PHYSICAL AND APPLICATION OBJECTS

- For the **AXIS** object, the **POSITION** parameter has been renamed **INITIAL_POSITION**, and it's access level is (NA). A new inquire function, **? pipe_motionless**, has been added.
- For the **AXES_SET** object, the **SHIFT** parameter has been renamed **SUBSET**. A new inquire function, **? pipe_motionless**, has been added.
- For the **ZERO_POSITIONER**, the access level for the **COARSE_SPEED**, **COARSE_MOVE**, **FINE_MOVE** and **RESOLVER_OFFSET** parameters has been changed to (RW).
- For the **ENCODER**, the **POSITION_PERIOD** parameter access is now (RW). The functionality of the Encoder is expanded to permit access to additional ST1 outputs. In

support of this expanded functionality, the **ADDRESS** parameter has been redefined. A new inquire function, **? value**, has been added.

- A new **SINK** object has been added.

1.1.4 THIS REVISION OF THE PAM REFERENCE MANUAL

Revision B (this edition) of the PAM Reference Manual incorporates all changes originally documented in Release Notes 006.8025 (Release 2.2) and 006.8026 (Release 2.3) as well as the changes related to Release 2.5 (see paragraph 1.1.3).

1.2 BASIC SYSTEM CONCEPTS AND DEFINITIONS

1.2.1 SYSTEM

Systems are comprised of components, and each component of the system is comprised of one or more ring nodes (nodes). Peripherals of various types are located at each node. The hierarchical relationship among system, components, nodes and peripherals is shown in Figure 1-1 and Figure 1-2.

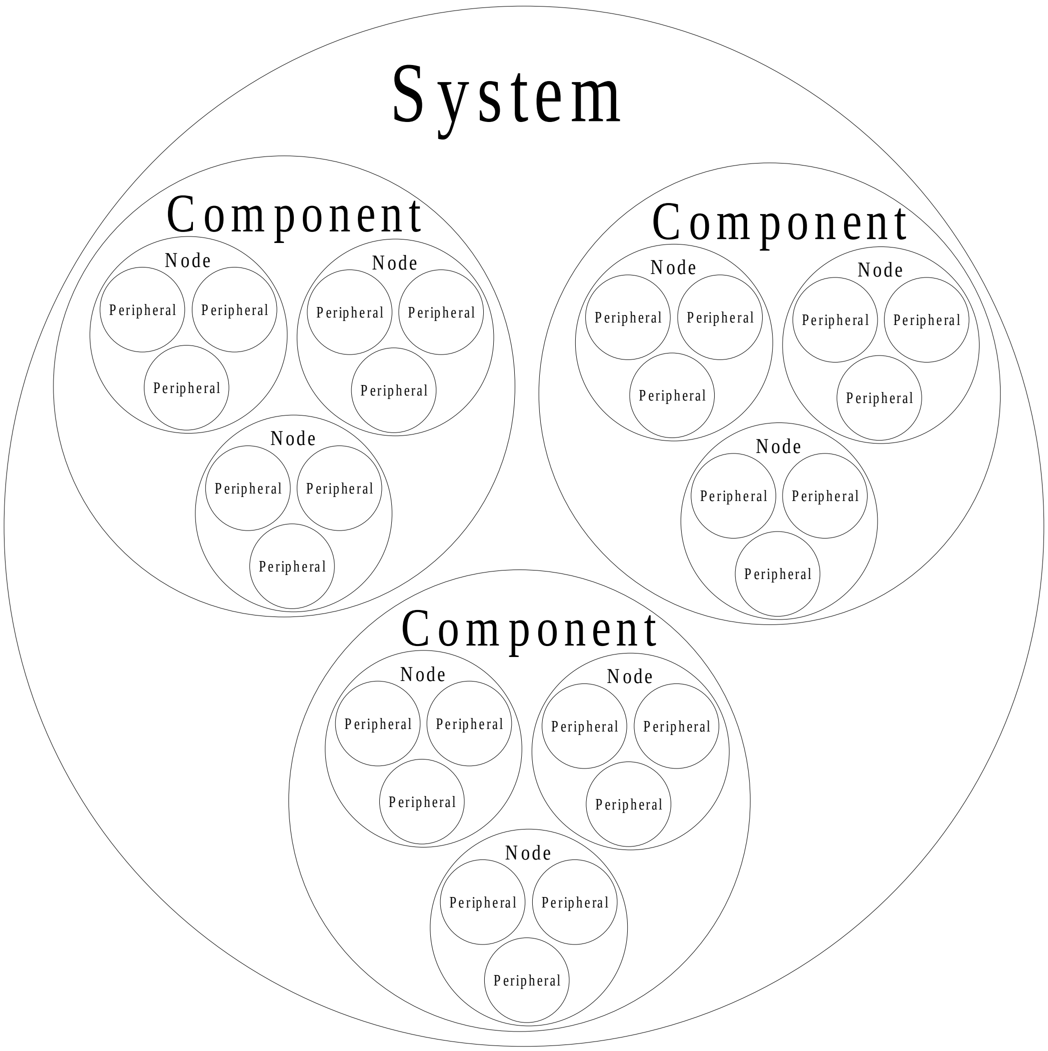


Figure 1-1 Graphical representation of a system

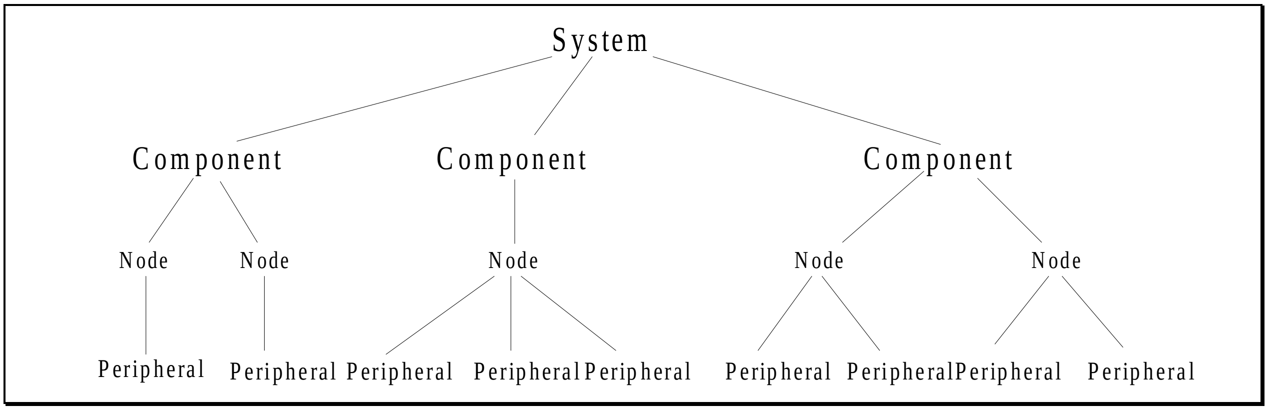


Figure 1-2 Alternative graphical representation of a system

1.2.1.1 DEFINITIONS

The behaviour of a component is the description of what a component does and how it does it. This behaviour is defined using the AGL (Alternative Graphical Language) declarations and statements in a PAM application program.

The function of a component is the set of operations executed by the component in the system.

1.2.1.2 EXAMPLE

Imagine a system comprised of two drilling machines (two components). The drilling machines have drills of different diameter. These two components have the same behaviour (drilling holes), but different functions (drill holes of different diameters).

1.2.2 KINDS OF SYSTEMS

From a PAM point of view, there are three basic kinds of systems:

1. Single

   Each component of the system has its own behaviour, which is different than the other system components' behaviour. All nodes of all components must be present and operational to have a working system. Figure 1-3 show a "single" system.

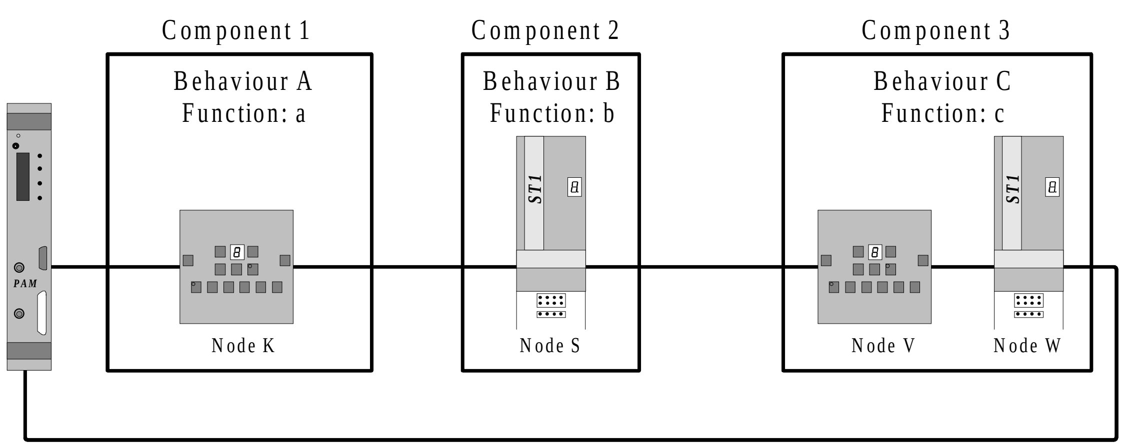


Figure 1-3 Single system with three components

2. Multiple with static configuration

   Several components of the system have the same behaviour, but different functions. The static configuration requires that all nodes of all components be present and operational to have a working system. Figure 1-4 show a "multiple with static configuration" system.

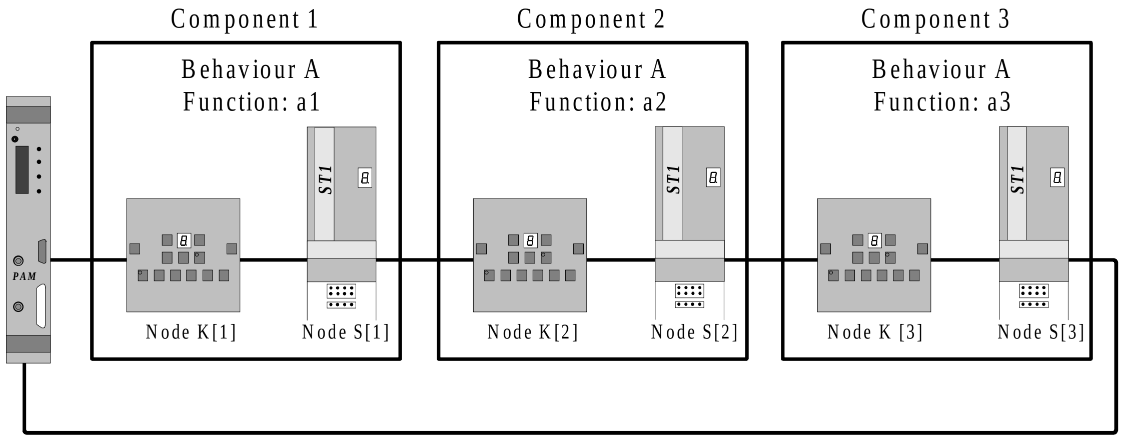


Figure 1-4 Multiple System with static configuration

3. Multiple with dynamic configuration

   Several components of the system have the same behaviour. With dynamic configuration the system can work even if some of these components are not present or not operational. Figure 1-5 show a "multiple system with dynamic configuration" .

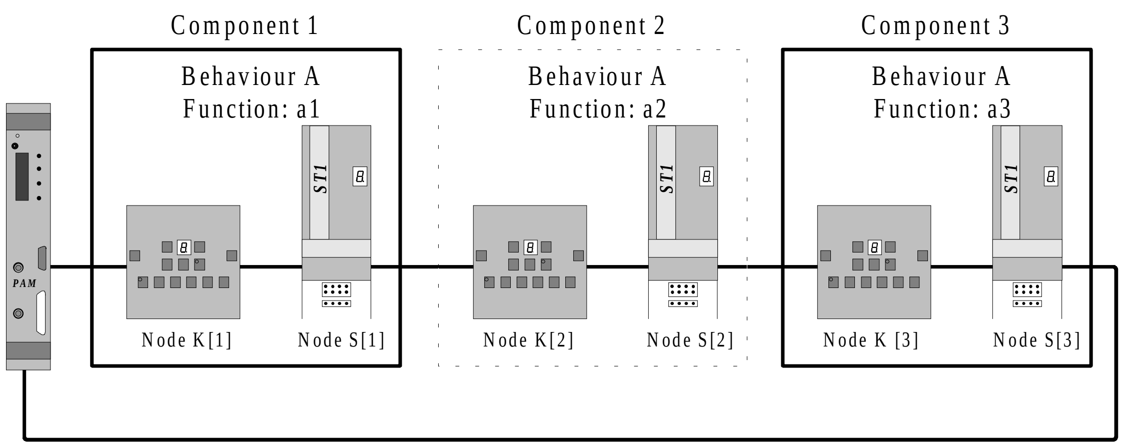


Figure 1-5 Multiple system with dynamic configuration

Multiple systems with dynamic configuration have two main advantages:

- auto-configuration
- node fault tolerance.

1.2.2.1 AUTO-CONFIGURATION

Using auto-configuration the same application (software) can drive systems with 1, 2, 3 or n identical components without modification or recompilation of the application.

1.2.2.2 NODE FAULT TOLERANCE

If a faulty component's node is bypassed on the ring, the whole faulty component (which can be composed of several nodes) will be ignored by the application without modification or recompilation of the application. To use auto-configuration with node fault tolerance, the application must know which nodes are assigned to a component. For this purpose, **NODES GROUPS** are used.

1.3 APPLICATION TERMS AND DEFINITIONS

1.3.1 PARALLELISM

All applications have a common characteristic, parallelism. Parallelism means that several components of the controlled machine or system must work simultaneously. To describe this behaviour in the application, we use tasks.

1.3.2 TASK

A task describes totally or partially the behaviour of a component of the system. A task can be divided into several sequences of operations. Only one sequence within a task can be active (under execution) at any time; however, multiple tasks may be executed simultaneously.

1.3.3 SEQUENCE

A sequence is started when an event occurs, a condition is true, an exception occurs or at power on. In a sequence, beyond statements involving physical objects (IO, axes, nodes, variables, etc.), it is possible to start another sequence within the same task, execute another task, abort the task, install or remove an exception, wait for a condition or a "time out", etc.

1.3.4 ACTIONS

An action is a succession of statements (excluding statements which wait for a condition.). An action is executed when an event occurs or at power on. Actions reside outside the task/sequence structure and are therefore executed whenever the triggering event or the state of a boolean variable is true.

1.4 DECLARATION AND STATEMENT SYNTAX

1.4.1 SYMBOLS AND ABBREVIATIONS

This section describes the meaning of the symbols and abbreviations used throughout this manual to define the syntactic structure(s) of the declarations and statements which comprise the PAM programming language.

- Words in upper case bold letters are Keywords. Keywords have special significance to the PAM programming language and may not be used identifiers, symbols, etc. in the application.
- A vertical bar "|" separating two syntactic structures indicates a syntactic alternative.
- Square brackets "[ ]" are used to enclose an optional syntactic structure.
- Brace "{ }" are used to enclose a repetitive syntactic structure:

  - {} means 1 repetition.
  - [{}] means 0 or 1 repetition.
  - {}* means 0,1,2,...,n repetitions.
  - {}+ mean 1,2,3,...,n repetitions.
- Angle brackets "< >" are used to enclose variable parts of syntactic structures.

1.4.2 APPLICATION COMMENTS

A comment is a sequence of text used to section and explain application code.

- The symbols **/*** and ***/** denote the start and end of a comment. They can not be nested.
- `/* This is a comment */`
- `/* This is a comment  /* but this is illegal */ */`

The characters "//" may be used to denote the start of a comment which terminates at the end of the line.

```
IFV_OutFlag <- set ;   // This is a comment
```

1.5 ARRANGEMENT OF APPLICATION PROGRAMS

The overall arrangement of an application program is as follows:

application information declaration

{node group declaration**}***

{node declaration**}+**

{axes set declaration**}***

{input-output peripheral declaration**}+**

dualport declaration

{variable declaration**}+**

{Boolean equation declaration **|**

actions declaration **|**

task declaration**}+**

1.5.1 APPLICATION INFORMATION DECLARATION

This section describes the structure and use of the Application Information declaration which is at the beginning of every application program.

SYNTAX

APPLICATION
NAME **=** <identifier> **V**<version> **;**
BASIC_PAM_CYCLE = <basic PAM cycle> ;
[**DEFAULT_TASK_WORKSPACE** = <default task workspace> ;]
END

<identifier>: the name of the application (only the 11 first characters are used)

<version>: the version of the application. The format is ddd.dd , where d is a digit.

<basic PAM cycle>: the basic PAM period time in 1/3 of milliseconds. All the application timing is based on this time.

<default task workspace> : the default workspace size (expressed in bytes) for all tasks of the application.

DECLARATION EXAMPLE

```
APPLICATION
   NAME             = my_applic V 1.00 ;
   BASIC_PAM_CYCLE   = 3 ;               // 1 ms cycle
END
```

1.6 APPLICATION SIZE

The application is saved in EEPROM after compression using a compression algorithm.

The maximum size for the application is limited to 256 Kbytes (262144 bytes). The size of the EEPROM is 128 Kbytes.

2 Physical Objects, Declarations and Uses

2.1 Introduction

Physical objects are parts of the system configuration which perform specific input/output functions utilising dedicated hardware/firmware. For most physical objects, this dedicated hardware/firmware resides in peripherals connected on the PAM Ring. All physical objects used in an application must be declared using the appropriate physical declaration statement. Similarly, peripherals connected on the PAM Ring must be declared in a ring node declaration (see paragraph 2.2). Table 2-1 lists the types of physical objects and their associated peripheral(s).

PHYSICAL OBJECT TYPE	ASSOCIATED PERIPHERAL
Axis	ST1
binary input	Smart IO or STI
digital input	Smart IO
counter input	Smart IO
key input	Smart IO
binary output	Smart IO or STI
digital output	Smart IO
analog output	Smart IO
LED output	Smart IO
seven segment display output	Smart IO
PAM analog output	PAM
DC motor	Smart IO
encoder	STI

Table 2-1 Physical Object Types

2.1.1 Physical Object Declaration Syntax

Each physical object is defined by a type (i.e. binary input), an identifier (symbolic name), and a set of parameters along with their initial values. The parameter set varies with the type of physical object. The general syntax for a physical object declaration is as follows:

<type> <identifier> **;**

{<parameter> = <initial value> **;**}*

END

2.1.2 Physical Object Parameters Access

Physical object parameters can be accessed by the application during its execution using a specific syntax. Each parameter has its own access level which is indicated in the parameter description. The possible levels with their abbreviations are as follows:

- No Access (NA)
- Read Only access (RO)

- Write Only access (WO)
- Read and Write access (RW)

PARAMETER INQUIRY SYNTAX

The syntax for parameter value inquiry is as follows:

<destination object> **<-** <object>**:**<parameter> ;

EXAMPLE:

Read current TRAVEL_SPEED parameter value of AXI_X and write the value into IWV_MyVariable:

```
IWV_MyVariable <- AXI_X:TRAVEL_SPEED ;
```

PARAMETER MODIFICATION SYNTAX

The syntax for parameter value modification is as follows:

<object>**:**<parameter> **<-** <expression> ;

EXAMPLE:

Modify the TRAVEL_SPEED parameter value of AXI_X:

```
AXI_X:TRAVEL_SPEED <- 123.75 * (IWV_OldSpeed - 100.0) ;
```

2.2 RING NODE DECLARATIONS

Ring Node declarations make the logical connection between the physical address on the PAM Ring assigned to a peripheral and it's identifier (symbolic name). All peripherals on the PAM Ring must be identified in a ring node declaration.

2.2.1 DECLARATION SYNTAX

NODE <identifier> ;
{ [**NUMBER** =<number>] ¦
NODES_GROUP =<nodes group identifier> };
ADDRESS = <address> ;
TYPE = <type> ;
END

<identifier>: name assigned to the node.

<number>: (NA), indicate whether the node is multiple or single. If **NUMBER** *declaration is omitted or if <number> = 1, the node is single, otherwise the node is multiple.*

<nodes group identifier>: (NA), name of the nodes group when the node is a member of a nodes group.

Refer to paragraph 3.2 for details on Nodes Groups.

<address>: (NA), the PAM Ring address, in hexadecimal, of the peripheral assigned to the node. If the object is multiple, <address> is the first of <number> of consecutive addresses.

The Node Address switches in the peripheral assigned to the node must match **ADDRESS** in the node declaration.

<type>: (NA), the peripheral type (ST1 or SMART_IO) assigned to the node.

2.2.2 NODE FUNCTIONS

The following functions are available for all nodes regardless of their type:

FUNCTION NAME	SET/ INQUIRE	PARAMETERS	RETURN VALUE
error	INQUIRE	-	BOOLEAN
error_code	INQUIRE	-	INTEGER

– **error**

Returns error status of the node.

– **error_code**

returns error code of the node. An error code 0 mean "no error". The error codes meaning depends of the node type.

EXAMPLES

```
IF NOD_Main ? error THEN
   ...
   IF NOD_Main ? error_code = 12 THEN
      ...
   END_IF
   ...
END_IF
```

2.2.3 NODE DECLARATION EXAMPLES

SINGLE SYSTEM

This example illustrates the declarations needed for describing the "single" system shown in Figure 2-1.

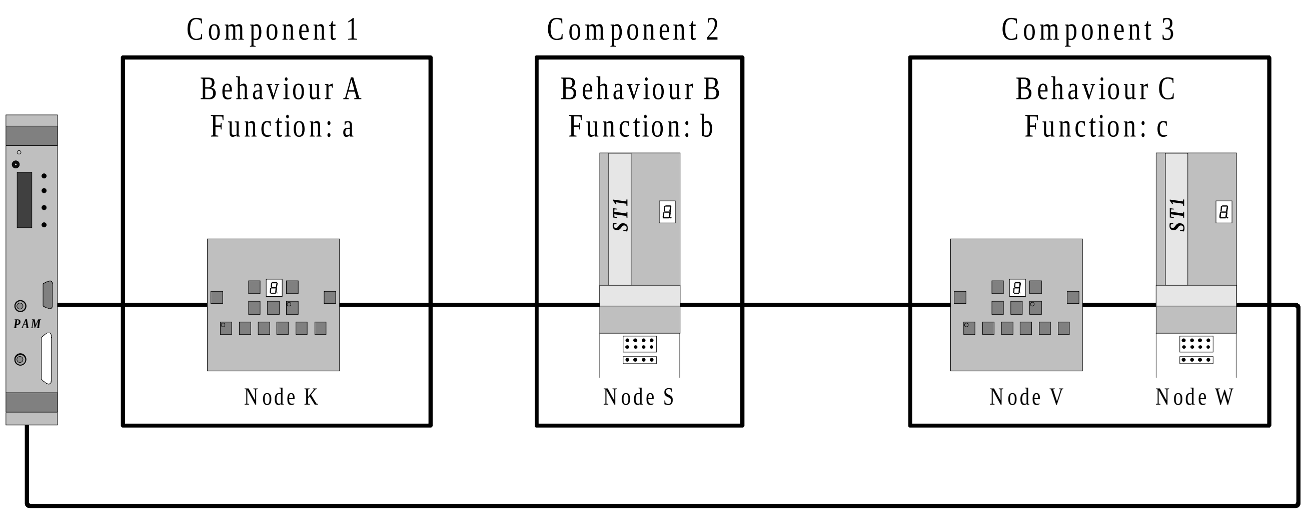


Figure 2-1 Single System

```
NODE NOD_K;
   ADDRESS     = 1 ;   // Address of the K node
   TYPE  = SMART_IO ;
END
...
NODE NOD_S;
   ADDRESS     = 17 ;   // Address of the S node
   TYPE  = ST1 ;
END
NODE NOD_V;
   ADDRESS     = 3 ;   // Address of the V node
   TYPE  = SMART_IO ;
END
...
NODE NOD_W;
   ADDRESS     = 33 ;   // Address of the W node
```

```
   TYPE  = ST1 ;
END
```

MULTIPLE NODES

This example shows the declarations needed for describing the multiple system illustrated in Figure 2-2.

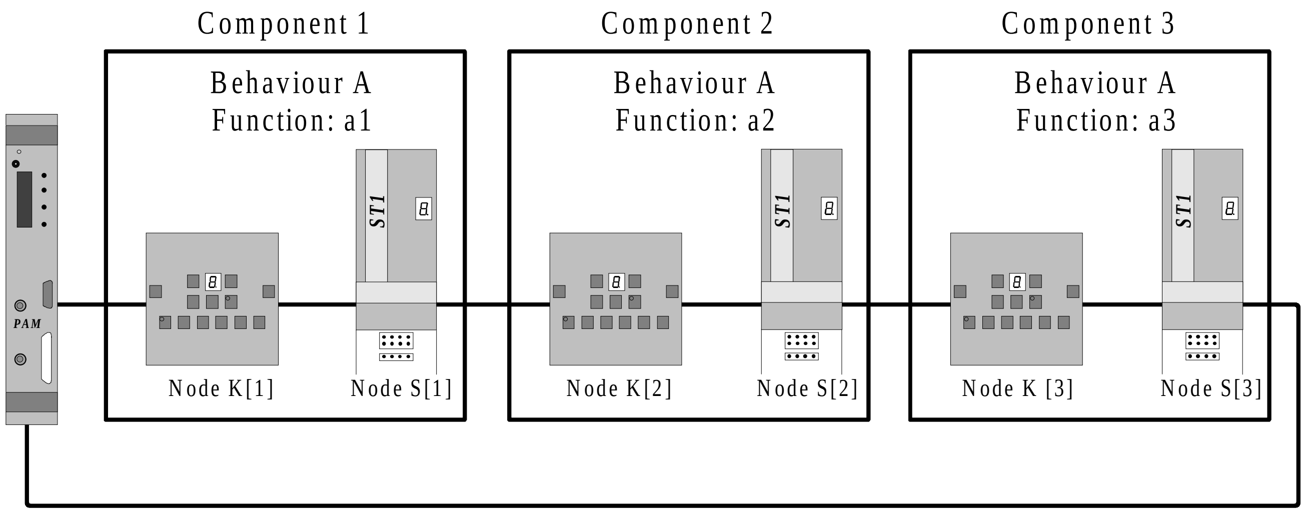
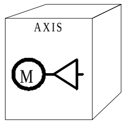


Figure 2-2 Multiple System

```
NODE NOD_K;
   NUMBER      = 3 ;
   ADDRESS     = 1 ;   // Address of the first K node
   TYPE  = SMART_IO ;
END
...
NODE NOD_S;
   NUMBER      = 3 ;
   ADDRESS     = 17 ;   // Address of the first S node
   TYPE  = ST1 ;
END
```

2.3 AXIS OBJECT



The Axis object handles the motion control (as opposed to the IO part) of an ST1 digital motion controller peripheral. There are four different ways to configure and control axis objects which include:

- controlling individually each axis using parameters and functions
- controlling directly multiple axes using parameters and functions
- controlling directly an axis set using parameters and functions
- controlling an axis (or axes) using the flow of values from a pipe

The optimum configuration is generally dictated by the particulars of the application.

As illustrated in the Axis object functional diagram (Figure 2-3), the trapezoidal motion profile generator responds to parameters and functions from the application, while the pipe interface processes pipe flow data. The axis supports simultaneous output from both sources, producing an axis motion profile which is the algebraic sum of the sources.

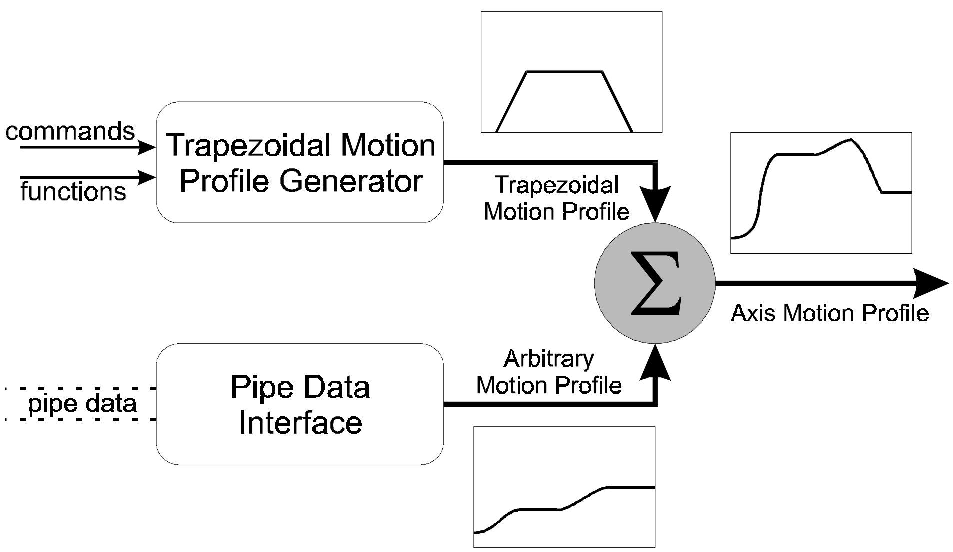


Figure 2-3 Axis Object Functional Diagram

DECLARATION SYNTAX

AXIS <identifier> ;
NODE = < node identifier>;
PULSES_PER_UNIT = <pulses per unit value> ;
TRAVEL_SPEED = <travel speed value> ;
ACCELERATION = <acceleration value> ;
[**DECELERATION** = <deceleration value> ;]
INITIAL_POSITION = <initial position value> ;
{ **POSITION_PERIOD** = <position period value> |
POSITION_RANGE = <min. position value> <max. position value>} ;
END

Only ST1 node types have axis objects.

<identifier> : name of the axis.

< node identifier> : (NA), name of the STI node where the axis part is located.

<travel speed value> : (RW), travel speed value for trapezoidal motion profile expressed in user length units per second. The travel speed value sets the constant speed part of the trapezoidal motion profile. This value must be greater than zero.

<acceleration value> : (RW), acceleration value for trapezoidal motion profile expressed in user length units per second squared. This value must be greater than zero.

<deceleration value> : (RW), deceleration value for trapezoidal motion profile expressed in user length units per second squared. This value must be greater than zero. If ***DECELERATION*** *is omitted, the <acceleration value> is used.*

<initial position value > : (RW), initial logical position value expressed in user length units. The axis is initialised to this value upon power-up or following a hardware reset of PAM. This operation generates no axis motion.

<position period value> : (NA), position period for cyclic motion systems, expressed in user length unit. This value must be greater than zero.

<min. position value>, <max. position value> : (NA), the position range for linear motion systems, expressed in user length units. The max. value must be greater than the min. value.

stop

The **POSITION_RANGE** parameter is not yet implemented. The position range of an axis can be specified by using the PHIL1 and PHIL2 parameters of the ST1. But to declare a non-periodic system, it must be specified in the axis declaration using dummy values.

<pulses per unit value>: (NA), the number of resolver units (RU) per user length unit.

The number of resolver units per turn for a single-speed resolver is 2^{32} RU.

For more information's see the "Motor Position" in the ST1 reference manual "Software for Synchronisation of Axes, 024.8068")

For example, with an ST1 node driving a motor with a single-speed resolver (2^{32} resolver units per revolution), if the user length unit is one degree (360 degrees per revolution) then the "pulse per unit value" must be 2^{32} / 360 = 11,930,464.71111111

Pulses per unit may also be specified as a negative number. A negative value produces rotation in the direction opposite from the motion produced with a positive value. This is the easiest way to reverse the motor rotation direction.

⚠ The precision of the motion control (especially for cyclic motion systems) directly depends on the precision of this parameter. The maximum number of digits is 16.

AXIS SAMPLE DECLARATION

```
AXIS AXI_Leader ;
   NODE              = NOD_Leader ;
   PULSES_PER_UNIT   = 11930464.7111 ;   /* (2^32/360)*/
   TRAVEL_SPEED      =  3600.0 ;         /*  10t/s     */
   ACCELERATION      = 36000.0 ;         /* 100t/s^2   */
   DECELERATION      = 36000.0 ;
   INITIAL_POSITION  =     0.0 ;
   POSITION_PERIOD   =   360.0 ;         /*   1t       */
END
```

FUNCTIONS

The following is a summary of the axis functions. Detailed descriptions of axis inquire functions are found in Chapter 9. Details on axis executive functions are found in Chapter 7.

FUNCTION NAME	SET/ INQUIRE	PARAMETERS	RETURN VALUE
absolute_move	set	- absolute position value	-
position	set	- position value	-
power_off	set	-	-
power_on	set	-	-
relative_move	set	- relative position value	-
run	set	- speed value	-
stop	set	-	-
update status	set	-	-
error	inquire	-	boolean
error_code	inquire	error code	boolean
error_code	inquire	-	integer
generator_position	inquire	-	real
pipe_motionless	inquire	-	boolean
position	inquire	-	real
ready	inquire	-	boolean
speed	inquire	-	real
status	inquire	status code	boolean
status	inquire	-	integer

2.3.1 PARAMETER MODIFICATION

2.3.1.1 TRAVEL_SPEED

Parameter access permits changes to the **TRAVEL_SPEED** parameter value specified in the **AXIS** declaration. The following rules apply when the travel speed value is modified:

- If the new travel speed value is greater than the maximum travel speed value (according to the ST1 axis CKV parameter), the maximum travel speed value is used.
- If a trapezoidal motion is in progress, the new travel speed value will be applied to subsequent trapezoidal motions.

i | Refer to document 024.8068 for details on the CKV parameter

SYNTAX

<object identifier>:**TRAVEL_SPEED <- (**<travel_speed value>**)**

EXAMPLE

```
AXI_AnAxis:TRAVEL_SPEED <- (3.22) ;
AXI_Axes[i]:TRAVEL_SPEED <- (IRV_Sp[i]) ;
AXI_Axes[all]:TRAVEL_SPEED <- (IRV_Sp * 1.25 - 1) ;
```

2.3.1.2 ACCELERATION

Parameter access permits changes to the **ACCELERATION** parameter value specified in the **AXIS** declaration. The acceleration value is always used to generate the first part of the trapezoidal motion profile.

The following rules apply when the acceleration value is modified:

- If the new acceleration value is less than or equal to zero, the current value is not replaced by the new value.
- If the new acceleration value is greater than the maximum acceleration value (according to the ST1 axis CKA parameter), the maximum acceleration value is used.
- If a trapezoidal motion is in progress, the new acceleration value will be used only for subsequent trapezoidal motions.

i | Refer to document 024.8068 for details on the CKA parameter

SYNTAX

<object identifier>:**ACCELERATION <- (**<acceleration value>**)**

EXAMPLE

```
AXI_AnAxis:ACCELERATION <- (3.22) ;
AXI_Axes[i]:ACCELERATION <- (IRV_Acc[i]) ;
AXI_Axes[all]:ACCELERATION <- (IRV_Acc * 1.25 - 1) ;
```

2.3.1.3 DECELERATION

Parameter access permits changes to the **DECELERATION** parameter value specified in the **AXIS** declaration. The deceleration value is always used to generate the last part of the trapezoidal motion profile.

The following rules apply when the deceleration value is modified:

- If the new deceleration value is less than or equal to zero, the current value is not replaced by the new value.
- If the new deceleration value is greater than the maximum deceleration value (according to the ST1 axis CKA parameter), the maximum deceleration value is used.
- If a trapezoidal motion is in progress, the new deceleration value will be used only for subsequent trapezoidal motions.

i | Refer to document 024.8068 for details on the CKA parameter

SYNTAX

<object identifier>**:DECELERATION <- (**<deceleration value>**)**

EXAMPLE

```
AXI_AnAxis:DECELERATION <- (3.22) ;
AXI_Axes[i]:DECELERATION <- (IRV_Decel[i]) ;
AXI_Axes[all]:DECELERATION <- (IRV_Decel * 1.25 - 1) ;
```

2.4 INPUT OBJECTS

Input objects are comprised of a physical part which resides within a peripheral (ST1 or SMART_IO) located within a ring node, and a logical part which resides within PAM. All inputs have two common characteristics, an owner node and an address. If the owner node is multiple, the peripheral is multiple; otherwise, if the owner node is single, the input is single.

2.4.1 DECLARATION SYNTAX

The general declaration syntax for an input is shown below. The specific declaration for each input type is found subsequent paragraphs of this section.

<logical input type> <identifier> **;**
    NODE = <owner node identifier> **;**
    <logical characteristics> ;
    [ON <physical peripheral type>**]**
    ADDRESS = <address> **;**
    <physical characteristics>
END

<logical input type>: type of input object.

<identifier>: name of the input object.

<owner node identifier>: (NA), identifier of the node where the input object is located.

<logical characteristics>: (NA), logical characteristics of the input.

<physical peripheral type>: (NA), descriptor for physical type of input.

<address>: (NA), input's address within the owner node. The nature of <address> depends on the owner node type (Refer to appendix D for ST1 and to appendix E for SMART_IO).

<physical characteristics>: (NA), physical characteristics of the output.

The most frequent physical characteristics are:

ACTIVE = { **HIGH** | **LOW** } ;
PERIOD = <period value in ms> ;
DEBOUNCE = <samples number> ;
<**HIGH** | **LOW**> : (NA), defines the input polarity.

*With **ACTIVE** = **HIGH**, the boolean true (1) corresponds to a high level state. With **ACTIVE** = **LOW**, the boolean true (1) corresponds to a low level state.*

<period value> : (NA), defines the scanning period in [ms] applied to an input.

<samples number> : (NA), defines the number of consecutive identical samples of an input required before the input assumes a new state.

2.4.2 BINARY INPUT

SYNTAX

BINARY_INPUT <identifier> **;**
NODE = <owner node identifier> **;**
[ON BINARY_INPUT]
ADDRESS = <address> **;**
ACTIVE = { **HIGH** | **LOW** } **;**
PERIOD = <period value in ms> **;**
DEBOUNCE = <samples number> **;**
END

FUNCTIONS

FUNCTION NAME	SET/ INQUIRE	PARAMETERS	RETURN VALUE
<- (read value)	INQUIRE	-	BOOLEAN

DECLARATION EXAMPLE

```
NODE NOD_XYZAxis ;
   NUMBER      = 3 ;
   ADDRESS     = 1 ;
   TYPE        = ST1 ;
END
   BINARY_INPUT SBI_InputA ;
   NODE        = NOD_XYZAxis ;
   ADDRESS     = 210 ;     // OIO board 10th input.
   ACTIVE      = LOW ;
   PERIOD      = 20 ;      // 20 ms.
   DEBOUNCE    = 1 ;
END
```

2.4.3 DIGITAL INPUT

DECLARATION SYNTAX

DIGITAL_INPUT <identifier> **;**
  NODE **=** <owner node identifier> **;**
  BITS = < number of bits > **;**
  [ON DIGITAL_INPUT]
  ADDRESS = <address> **;**
  ACTIVE = { **HIGH** | **LOW** } **;**
  PERIOD = <period value in ms> **;**
  DEBOUNCE **=** <samples number> **;**
END

< number of bits > : (NA), size in bits of the digital input.

FUNCTIONS

FUNCTION NAME	SET/ INQUIRE	PARAMETERS	RETURN VALUE
<- (read value)	INQUIRE	-	INTEGER

Digital Inputs are not available for ST1 nodes.

EXAMPLE

```
NODE NOD_ToolsIo;
   NUMBER      = 3 ;
   ADDRESS     = 41 ;      // #29
   TYPE        = SMART_IO ;
END
   DIGITAL_INPUT SDI_DigitalInput1 ;
   NODE        = NOD_ToolsIo;
   BITS        = 5 ;
   ADDRESS     = 208 ;     // module 2, use  8th to 12th inputs.
   ACTIVE      = HIGH ;
   PERIOD      = 10 ;      // 10 ms.
   DEBOUNCE    = 2 ;
END
```

Default implementation of digital inputs on a SMART_IO is as binary inputs.

2.4.4 Counter Input

DECLARATION SYNTAX

COUNTER_INPUT <identifier> ;
  NODE = <owner node identifier> ;
  [**ON COUNTER_INPUT**]
  ADDRESS = <address> ;
  ACTIVE = { **HIGH** | **LOW** } ;
  PERIOD = <period value in ms> ;
  DEBOUNCE = <samples number> ;
END

FUNCTIONS

FUNCTION NAME	SET/ INQUIRE	PARAMETERS	RETURN VALUE
<- (read value)	INQUIRE	-	INTEGER

Counter Inputs are designed to be used only with a DC_motor output (see paragraph 2.6.1) on a Smart-IO.

Example

```
NODE NOD_ToolsIo;
   NUMBER      = 3 ;
   ADDRESS     = 41 ;      // #29
   TYPE        = SMART_IO ;
END

COUNTER_INPUT CNT_DCmotorCounter ;
   NODE        = NOD_ToolsIo;
   ADDRESS     = 101 ;     // module 1, 1st input.
   ACTIVE      = HIGH ;
   PERIOD      = 1 ; // 1 ms.
   DEBOUNCE    = 1 ;
END
```

Default implementation of counter inputs on a SMART_IO is as binary inputs.

2.4.5 KEY INPUT

DECLARATION SYNTAX

The declaration syntax for a key (keyboard) input is as follows:

KEY_INPUT <identifier> **;**
  NODE = <owner node identifier> **;**
  MODE = { **ON_OFF** | **TOGGLE** | **AUTOREPEAT** } **;**
  [ON KEY_INPUT]
  ADDRESS = <address> **;**
  ACTIVE = { **HIGH** | **LOW** } **;**
  PERIOD = <period value in ms> **;**
END

The declaration syntax for a key (keyboard) input implemented using a binary input physical part is as follows:

KEY_INPUT <identifier> **;**
  NODE = <owner node identifier> **;**
  MODE = { **ON_OFF** | **TOGGLE** | **AUTOREPEAT** } **;**
  ON BINARY_INPUT
  ADDRESS = <address> **;**
  ACTIVE = { **HIGH** | **LOW** } **;**
  PERIOD = <period value in ms> **;**
  DEBOUNCE = <samples number> **;**
END

FUNCTIONS

FUNCTION NAME	SET/ INQUIRE	PARAMETERS	RETURN VALUE
invert	SET	-	-
reset	SET	-	-
set	SET	-	-
<- (read value)	INQUIRE	-	BOOLEAN

Key Inputs are not available for ST1 nodes.

DECLARATION EXAMPLES

Keyboard input declarations with a SMART_IO node :

```
NODE NOD_ToolsIo;
   NUMBER      = 3 ;
   ADDRESS     = 41 ;      // #29
   TYPE        = SMART_IO ;
END
   KEY_INPUT KEY_Start ;
   NODE        = NOD_ToolsIo;
   MODE        = ON_OFF ;
ADDRESS  = 12 ;      // position 12 in key array.
   ACTIVE      = HIGH ;
```

```
   PERIOD      = 50 ;      // [ms]
END


   KEY_INPUT KEY_Steps ;
      NODE           = NOD_ToolsIo;
      MODE           = AUTOREPEAT ;
      DELAY          = 1500 ;    // delay of 1500 ms before repeating.
      REPEAT   = 500 ;     // time interval between repeating.
      ADDRESS  = 18 ;      // position 18 in key array.
      ACTIVE   = HIGH ;
      PERIOD   = 50 ;      // [ms]
   END
```

Declaration Example for keyboard input on a physical binary input:

```
KEY_INPUT KEY_Deported1 ;
   NODE        = NOD_ToolsIo;
   MODE        = TOGGLE ;
ON BINARY_INPUT
   ADDRESS     = 103 ;     // module 1, 3rd input.
   ACTIVE      = HIGH ;
   PERIOD      = 20 ;      // [ms]
   DEBOUNCE    = 2 ;
END
```

2.5 OUTPUT OBJECTS

Output objects are comprised of a physical part which resides within PAM or within a peripheral (ST1 or SMART_IO), and a logical part which resides within PAM. All outputs except the PAM Analog Output have two common characteristics, an owner node and an address. If the owner node is multiple, the peripheral is multiple; otherwise, if the owner node is single, the output is single.

2.5.1 DECLARATION SYNTAX

The general declaration syntax for an output is shown below. The specific declaration for each output type is found subsequent paragraphs of this section.

<logical output type> <identifier> **;**
    NODE = **<owner node identifier> **;
    <logical characteristics>
    [ON <physical peripheral type>**]**
    ADDRESS = <address> **;**
    <physical characteristics>
END

<logical output type>: type of output object.

<identifier>: name of the output object.

<owner node identifier>: (NA), identifier of the node where the output object is located.

<logical characteristics>: (NA), logical characteristics of the output. The set of logical characteristics varies with the output type.

<physical peripheral type>: (NA), descriptor for physical type of output.

<address>: (NA), the output's address within the owner node. The nature of <address> depends on the owner node type (Refer to appendix D for ST1 and to appendix E for SMART_IO).

<physical characteristics>: (NA), the physicals characteristics of the output. The set of physical characteristics varies as a function of the output type.

2.5.2 BINARY OUTPUT

DECLARATION SYNTAX

BINARY_OUTPUT <identifier> **;**
    NODE = **<owner node identifier> **;
    [ON BINARY_OUTPUT]
    ADDRESS = <address> **;**
    ACTIVE = { **HIGH** | **LOW** } **;**
END

*<**HIGH** | **LOW**> : (NA), defines the output polarity.*

With active high, the boolean true (1) corresponds to a high level state. With active low, the boolean true (1) corresponds to a low level state.

FUNCTIONS

FUNCTION NAME	SET/ INQUIRE	PARAMETERS	RETURN VALUE
<- (assign value)	SET	- boolean expression	-
invert	SET	-	-
reset	SET	-	-
set	SET	-	-
<- (read value)	INQUIRE	-	BOOLEAN

DECLARATION EXAMPLE

```
NODE NOD_XYZAxis ;
   NUMBER      = 3 ;
   ADDRESS     = 1 ;
   TYPE        = ST1 ;
END
   BINARY_OUTPUT SBO_OutputA ;
   NODE        = NOD_XYZAxis ;
   ADDRESS     = 201 ;          // OIO board 1st input.
   ACTIVE      = HIGH ;
END
```

2.5.3 DIGITAL OUTPUT

DECLARATION SYNTAX

DIGITAL_OUTPUT <identifier> **;**
  NODE = <owner node identifier> **;**
  BITS = <bits number> **;**
  [ON DIGITAL_OUTPUT]
  ADDRESS = <address> **;**
  ACTIVE = { **HIGH** | **LOW** } **;**
END

<bits number> : (NA), size in bits of the digital output.

<HIGH | LOW> : (NA), defines the output polarity.

With active high, the boolean true (1) corresponds to a high level state. With active low, the boolean true (1) corresponds to a low level state.

FUNCTIONS

FUNCTION NAME	SET/ INQUIRE	PARAMETERS	RETURN VALUE
<- (assign value)	SET	- numerical expression	-
<- (read value)	INQUIRE	-	BOOLEAN

Digital Outputs are not available for ST1 nodes.

DECLARATION EXAMPLE

```
NODE NOD_ToolsIo;
   NUMBER      = 3 ;
   ADDRESS     = 41 ;      // #29
   TYPE        = SMART_IO ;
END
   DIGITAL_OUTPUT SDO_DigitalOutput1 ;
   NODE        = NOD_ToolsIo;
   BITS        = 6 ;
   ADDRESS     = 201 ;     // module 2, use 1st to 6th outputs.
   ACTIVE      = HIGH ;
END
```

Default implementation of a digital outputs on a SMART_IO is as binary outputs.

2.5.4 ANALOG OUTPUT

DECLARATION SYNTAX

ANALOG_OUTPUT <identifier> **;**
NODE = <owner node identifier> **;**
RANGE = <lower bound> <upper bound>**;**
[ON ANALOG_OUTPUT]
ADDRESS = <address> **;**
END

<lower bound> : (NA), lower bound in user units corresponding to the maximum negative voltage of the analog output.

<upper bound> : (NA), upper bound in user units corresponding to the maximum positive voltage of the analog output.

FUNCTIONS

FUNCTION NAME	SET/ INQUIRE	PARAMETERS	RETURN VALUE
<- (assign value)	SET	- numerical expression	-
<- (read value)	INQUIRE	-	INTEGER

⚠ Analog Outputs are not available for ST1 nodes.

DECLARATION EXAMPLE

```
NODE NOD_ToolsIo;
   NUMBER      = 3 ;
   ADDRESS     = 41 ;      // #29
   TYPE        = SMART_IO ;
END
   ANALOG_OUTPUT SAO_AnalogOutput1 ;
   NODE        = NOD_ToolsIo;
   RANGE       = -10000  10000 ; // define user units.
   ADDRESS     = 1 ;             // default address for the analog
                              output of a smart_io.
END
```

The output voltage delivered by the analog output of the SMART_IO ranges from - 10 [v] to + 10 [v]. In this example, by defining **RANGE** = -10000 10000, the user defines the scaling factor for the analog output to be 1 millivolt/unit.

2.5.5 LED OUTPUT

DECLARATION SYNTAX

LED_OUTPUT <identifier> **;**
    NODE = <owner node identifier> **;**
    [ON LED_OUTPUT]
    ADDRESS = <address> **;**
END

FUNCTIONS

FUNCTION NAME	SET/ INQUIRE	PARAMETERS	RETURN VALUE
<- (assign value)	SET	- boolean expression	-
invert	SET	-	-
reset	SET	-	-
set	SET	-	-
blink	SET	-	-

Led Outputs are not available for ST1 nodes.

DECLARATION EXAMPLE

```
NODE NOD_ToolsIo;
   NUMBER      = 3 ;
   ADDRESS     = 41 ;       // #29
   TYPE        = SMART_IO ;
END
   LED_OUTPUT LED_Output1 ;
   NODE        = NOD_ToolsIo;
   ADDRESS     = 6;         // position 6 in led array.
END
```

2.5.6 DISPLAY OUTPUT (7 SEGMENTS)

DECLARATION SYNTAX

D7SEG_OUTPUT <identifier> **;**
    NODE = <owner node identifier> **;**
    DIGITS = <digits number>**;**
    FONT = <font type>**;**
    [ON D7SEG_OUTPUT]
    ADDRESS = <address> **;**
END

FUNCTIONS

FUNCTION NAME	SET/ INQUIRE	PARAMETERS	RETURN VALUE
blink	SET	-	-
display	SET	- characters string	-
display	SET	- numerical expression - digits number	-
no_blink	SET	-	-

⚠ Display Outputs are not available for ST1 nodes.

DECLARATION EXAMPLE

```
NODE NOD_ToolsIo;
   NUMBER      = 3 ;
   ADDRESS     = 41 ;      // #29
   TYPE        = SMART_IO ;
END
   D7SEG_OUTPUT D7S_LedOutput1 ;
   NODE        = NOD_ToolsIo;
   DIGITS      = 1         // number of digits.
   ADDRESS     = 1 ;       // default address for the 7 segments
                    // display output of a smart_io.
END
```

2.5.7 PAM ANALOG OUTPUT

Two analog outputs located on the PAM front panel are provided for monitoring purposes. They utilise 8 bit DACs with an output amplitude range of 0 to 2.5 Volts.

DECLARATION SYNTAX

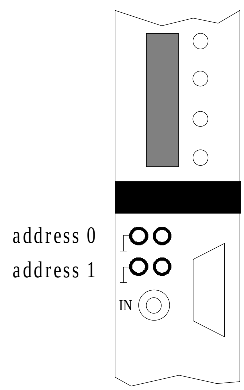
PAM_ANALOG_OUTPUT <identifier> **;**
  UNITS_PER_VOLT = <input attenuation value> ;
  OFFSET = <input offset value> ;
  ADDRESS = <analog output address> ;
END

<identifier> : (NA), name of the PAM analog output.

<input attenuation> : (NA), sum of (<input value>+<input offset>) which is to produce 1 Volt on the output. See output voltage equation below.

<input offset> : (NA), value added to the input value.

<analog output address> : (NA), address of the PAM analog output (0 or 1).



Output voltage = (input value + input offset) / input attenuation

FUNCTIONS

FUNCTION NAME	SET/ INQUIRE	PARAMETERS	RETURN VALUE
<- (assign value)	SET	- numerical expression	-
<- (read value)	INQUIRE	-	INTEGER

DECLARATION EXAMPLES:

Declarations and link with a pipe block

```
PAM_ANALOG_OUTPUT PAO_PamDac0 ;
   UNITS_PER_VOLT    = 1000.0 ;
   OFFSET            = 0.0 ;
   ADDRESS           = 0 ;
END


PAM_ANALOG_OUTPUT PAO_PamDac1 ;
   UNITS_PER_VOLT    = 10000.0 ;
```

```
   OFFSET           = 12500.0 ;
   ADDRESS          = 1 ;
END

CONVERTER CNV_PipeOutputProbe ;
   DESTINATION      = PAO_PamDac0 ;
   MODE             = VALUE ;
END
```

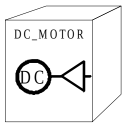
```
Monitoring of the output set point of a pipe:

   ....
   CNV_PipeOutputProbe << CAM_MyCam << TMP_Generator ;
   ...
```

Direct use of PAM analog outputs:

```
   PAO_PamDac0 <- 1245.38 ;           // 1.24 Volt on output
   PAO_PamDac1 <- - 12500 ;           // 0 Volt on output (Vmin)
   PAO_PamDac1 <- 0 ;                 // 1.25 Volt on output
   PAO_PamDac1 <- 12500 ;       // 2.5 Volt on output (Vmax)
```

2.6 SPECIAL OUTPUTS

2.6.1 DC MOTOR



The DC motor output, which is integral to the Smart IO peripheral, is a dedicated interface for small DC motor.

DECLARATION SYNTAX

DC_MOTOR <identifier> **;**
  INC = <binary output identifier> **;**
  DEC = <binary output identifier>**;**
  ZERO = <binary input identifier>**;**
  ENCODER = <counter input identifier> **;**
  TIMEOUT = <timeout value> **;**
  BREAKING = <braking value> **;**
END

<timeout value> : (NA), value in [ms] for the counting timeout when DC_motor rotation is ordered.

<braking value> : (NA), braking distance of the DC_motor given in number of pulses.

FUNCTIONS

FUNCTION NAME	SET/ INQUIRE	PARAMETERS	RETURN VALUE
absolute_move	SET	- final position value	-
position	SET	- position	-
relative_move	SET	- delta position value	-
run	SET	- direction { **INC** ¦ **DEC** }	-
stop	SET	-	-
zero_position	SET	- direction { **INC** ¦ **DEC** } - polarity { **HIGH** ¦ **LOW** }	-
stroke_limits	SET	- limits { lower, upper }	-
disable_stroke_limits	SET	-	-
position	INQ	-	INTEGER
ready	INQ	-	BOOLEAN

DC motors are only available on SMART_IO nodes.

DECLARATION EXAMPLE

```
NODE NOD_ToolsIo;
   NUMBER      = 3 ;
   ADDRESS     = 41 ;
   TYPE        = SMART_IO ;
END

BINARY_OUTPUT SBO_DCmotorInc ;
   NODE        = NOD_ToolsIo ;
   ADDRESS     = 201 ;
   ACTIVE      = HIGH ;
END


   BINARY_OUTPUT SBO_DCmotorDec ;
   NODE        = NOD_ToolsIo ;
   ADDRESS     = 202 ;
   ACTIVE      = HIGH ;
END


   BINARY_INPUT SBI_DCmotorZero ;
   NODE        = NOD_ToolsIo;
   ADDRESS     = 102 ;
   ACTIVE      = HIGH ;
   PERIOD      = 1 ; // 1 ms
   DEBOUNCE    = 1 ;
END


COUNTER_INPUT CNT_DCmotor ;
   NODE        = ToolsIo;
   ADDRESS     = 101 ;
   ACTIVE      = HIGH ;
   PERIOD      = 1 ; // 1 ms
   DEBOUNCE    = 1 ;
END

DC_MOTOR DCM_ToolsAdjust;
   INC         = SBO_DCmotorInc ;
   DEC         = SBO_DCmotorDec ;
   ZERO        = SBI_DCmotorZero ;
   ENCODER     = CNT_DCmotor ;
   TIMEOUT     = 500 ;            // counting timeout in [ms].
   BREAKING    = 2 ;              // breaking distance in pulses.
END
```

2.7 ENCODER OBJECT

2.7.1 INTRODUCTION

An encoder is a physical PAM object which enables an application to read the following types of information from an ST1:

- speed or position of a resolver or encoder connected to an ST1 peripheral
- value of an analog input (i.e. potentiometer) connected to an ST1 peripheral
- value of an ST1 parameter
- value of an ST1 variable

2.7.2 DECLARATION SYNTAX

ENCODER <identifier> ;
    NODE = <node identifier> ;
    ADDRESS = <address> ;
    PULSES_PER_UNIT = <pulses per unit value> ;
    {**POSITION_PERIOD** = <position period value> |
    POSITION RANGE = <min. position value > <max. position value >};
END

<identifier> : name of the encoder.

<node identifier> : (NA), name of a node.

<address> : (RW), address (in decimal) within the node of the variable/parameter to be encoded *(see paragraph 2.7.4).*

<pulse per unit value>: (RW), the number of encoder (or other) units per user unit.

The number of encoder units per turn for a single-speed resolver is 2^{16} RU.

For more information see the "PHIC/PHIB" in the ST1 reference manual "Software for Synchronisation of Axes, 024.8068")

For example, with an ST1 node which is connected to a motor with a single-speed resolver acting as an encoder (2^{16} encoder units per revolution), if the user length must be one degree (360 degrees per revolution), then the "pulses per unit value" must be 2^{16} / 360 = 182.0444444444444

Pulses per unit may be specified as a negative number. A negative value reverses the direction of positive encoder rotation compared to a positive value. This is the easiest way to reverse the encoder positive rotation direction.

⚠ The precision of the motion control (especially for cyclic motion systems) directly depends on the precision of this parameter. The maximum number of digits is 16.

<position period value> : (RW), the position period for cyclic motion systems, expressed in user's units. This value must be greater than zero.

Do not modify **POSITION_PERIOD** while a sampler pipe block is active with an encoder as it's source object. Doing so may result in the sampler producing erroneous outputs.

<min. position value>, <max. position value> : (NA), the position range for linear motion systems, expressed in user length units. The max. value must be greater than the min. value.

FUNCTIONS

FUNCTION NAME	SET/ INQUIRE	PARAMETERS	RETURN VALUE
value | position	inquire		binary
speed	inquire		binary

DECLARATION EXAMPLE

```
ENCODER ENC_Blanket;
      NODE                    = NOD_Blanket;
      ADDRESS        = 0;
      PULSES_PER_UNIT         = 182.0444444444444; // 2^16 / 360
      POSITION_PERIOD         = 10.0;
END
```

2.7.3 ENCODER USE AND BEHAVIOUR

GENERALITIES

The purpose of the encoder object is to acquire the desired parameter or variable value and perform the manipulations necessary to convert its internal units to the specified application units.

INTERNAL MECHANISM

When an encoder is used, it is important to understand some details about the internal mechanism that provides the desired information. For the encoder, two different operating environments must be distinguished:

- A sampler (pipe block) on the encoder output is currently active:

In this situation, the ST1 variable is periodically read and its current value recorded by PAM. Thus when the application asks:

```
IRV_PosAxis1 <- ENC_Blanket ? position ;
```

PAM immediately returns it's most recently recorded resolver position.

- No sampler is currently active on the encoder output:

In this situation PAM does not periodically read and record the ST1 variable. Thus when the application asks:

```
IWV_TempAxis1 <- ENC_Blanket ? value ;
```

PAM must perform a read cycle on the ST1 which takes six to seven × **BASIC_PAM_CYCLE** to complete. During this interval the sequence is suspended (see paragraph 3.10.7.1). When the sequence becomes active again, IWV_TempAxis1 contains the current value of the ST1 variable/parameter.

REMARKS

Even though the sequence may be suspended during execution of an inquire function, it is not a service statement because it does not specify when some action must take place but how this action must be performed. For this reason it is classified as an active statement; however, it can require some time for execution).

Never use an inquire function in an action.

The internal representation of ST1 variables is bounded. With machines that rotate always in the same direction, the position variable, for example, is always increasing. Eventually, the internal representation of position "rolls over" from the largest possible number to the smallest possible number. This position roll-over point (position periodicity) does not always correspond to a period of the machine or application. PAM automatically take care of this periodicity mismatch and it is "invisible" to the application. However, it is necessary that your application generates at least two encoder inquire functions **(? value** or other statement which inquires the STI variable) within one period (between successive roll-overs) of the variable.

For example, imagine a resolver with 16 bit resolution (2^{16} counts per rotation) connected to the resolver input of an ST1 and the periodicity (size) of the register within the ST1 keeping resolver position (PHIRE) is 2^{32}. The application must guarantee that PHIRE is read at least two times within 2^{16} rotations of the resolver. If not, the position value may be meaningless.

2.7.4 SPECIFYING ENCODER ADDRESS PARAMETER

Table 2-2 contains the information required to determine the **ADDRESS** parameter for an encoder declaration. Column 2 lists the decimal address (or range of addresses) assigned to ST1 items which may be connected to an encoder. Column 1 lists the equivalent hexadecimal address (or range of addresses). Columns 3 defines the item (or item type). Columns 4 and 5 specify the resolution and periodicity (where applicable) to be applied by the ST1.

Addresses 0 thru 4 are reserved for primary STI outputs, and addresses 5 thru 1A (hex) are reserved for future use.

Addresses x1B thru xFF (where x = 0, 1, 2 or 3) are for linking to ST1 internal variables, where 1B thru FF identifies the variable (see document 024.8068 for internal variable code numbers) and "x" defines how the ST1 handles the variable internally (i.e. the resolution and periodicity applied).

The situation for parameters is similar. Addresses y1B thru yFF (where y = 4, 5, 6 or 7) are for linking to ST1 internal parameters, where 1B thru FF identifies the parameter (see document

024.8068 for internal parameter code numbers) and “y” defines the parameter’s numerical characteristics.

ADDRESS EQUIVALENT HEX VALUE	<ADDRESS> DECIMAL VALUES	ITEM LINKED TO ENCODER	SIZE/RE-SOLUTION	PERIODICITY
0	0	main resolver	32 bits	2^{32}
1	1	second resolver	32 bits	2^{32}
2	2	axis set point	32 bits	2^{32}
3	3	incremental encoder	32 bits	2^{32}
4	4	ST1 aux. analog output	16 bits	2^{16}
5-1A	5-26	reserved	-	-
01B-0FF	27-255	Variable	16 bits	none
11B-1FF	282-511	Variable	16 bits	2^{16}
21B-2FF	538-767	Variable	32 bits	none
31B-3FF	795-1023	Variable	32 bits	2^{32}
41B-4FF	1051-1279	Parameter	16 bits	none
51B-5FF	1307-1535	Parameter	16 bits	2^{16}
61B-6FF	1563-1791	Parameter	32 bits	none
71B-7FF	1819-2047	Parameter	32 bits	2^{32}

Table 2-2 Encoder Address Parameter Selection

2.7.5 EXAMPLE OF ENCODED ADDRESS DETERMINATION

```
ENCODER ENC_TempAxis1;
   NODE                 = NOD_Axis1;
   ADDRESS              = 1273;            // DAP_TEMP  #F9 + #400
   PULSES_PER_UNIT      = 16.0;            // Degree
   POSITION_RANGE       =  0.0   60.0;
END
```

3 APPLICATION OBJECTS DECLARATIONS AND USES

3.1 INTRODUCTION

Application objects are the non-physical parts of an application configuration. Declarations are used to specify all application objects. Pipe objects (see chapter 5) form their own category and are not included in the application objects category. Application objects include:

nodes groups
axis sets
zero positioners
variables
equations
actions
routines
tasks
sequences

3.1.1 APPLICATION OBJECT DECLARATION SYNTAX

Each application object is defined by a type, a name and some specific parameters along with their initial values. Different syntax structures are used depending on the object. Refer to subsequent sections of this chapter for the declaration syntax for each type of application object.

3.1.2 APPLICATION OBJECT PARAMETERS ACCESS

Application object parameter values (except certain Zero Positioner parameters) can not be accessed by the application during its execution. The access level which is indicated in the parameter description is:

- No Access (NA)

For those parameters which are accessible for inquiry or modification, the syntax is the same as for physical objects (see paragraph 2.1.2).

3.2 NODES GROUPS

3.2.1 PURPOSE

In order to use the auto-configuration or node fault tolerance features, components with identical behaviour, (see paragraph 1.2) must be specified via a **NODES GROUP** declaration. The **NODES GROUP** declaration informs PAM about the number of components (see paragraph 1.2) with an identical behaviour and assigns an identifier (symbolic name) to the components group. Member nodes are identified in individual **NODE** declarations.

3.2.2 DECLARATION SYNTAX

NODES_GROUP <identifier> **;**
NUMBER = <components number> **;**
END

<identifier>: name of the nodes group.

<components number>: (NA), number of identical components in the nodes group

3.2.3 DECLARATION EXAMPLE

This example shows the node declarations needed for of the system illustrated in Figure 3-1. The **NODES_GROUP** declaration informs PAM of the existence of three identical components which are assigned the identifier NGR_COMPONENT. Two **NODE** declarations logically connect identical nodes K[ ] and nodes S[ ] to NGR_COMPONENT. PAM assigns **ADDRESS** = 1, 2 and 3 respectively to nodes K[ ] and **ADDRESS** = 63, 64 and 65 to nodes S[ ].

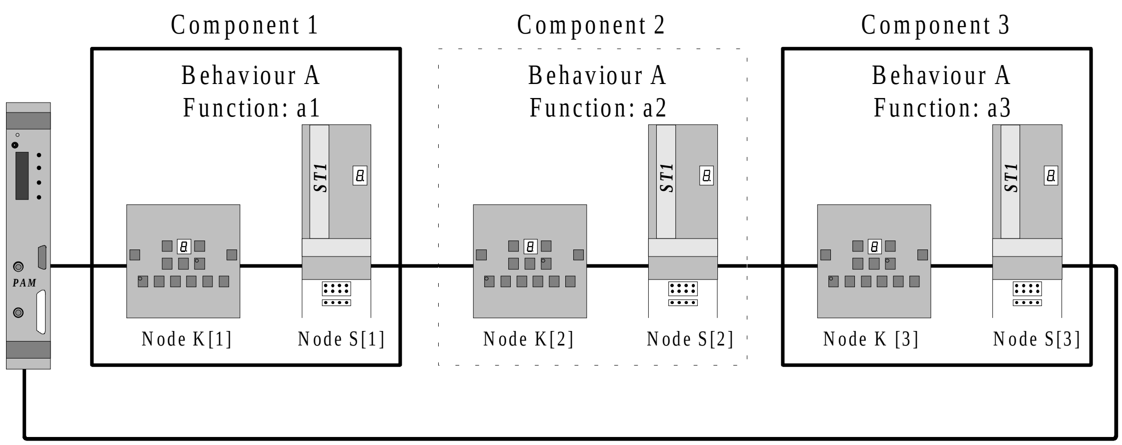
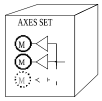


Figure 3-1 System with three Identical Components

```
NODES_GROUP NGR_Component ;
   NUMBER   = 3 ;        // Maximum components number
END
```

```
...
NODE NOD_K;
   NODES_GROUP    = NGR_Component ;
   ADDRESS        = 1 ;   // Address of the first K node
   TYPE           = SMART_IO ;
END
...
NODE NOD_S;
   NODES_GROUP    = NGR_Component ;
   ADDRESS        = 63 ;   // Address of the first S node
   TYPE           = ST1 ;
END
```

3.3 AXES SET



3.3.1 PURPOSE

Several axes (axis objects) can be grouped into one logical entity called an axes set which may then be referenced using a single identifier to simplify and reduce the size of an application. Furthermore, functions can then be applied globally to an entire axes set in a more efficient manner, thereby improving the application's execution performance.

3.3.2 DECLARATION SYNTAX

AXES_SET <identifier> **;**

[**OPTIMIZE** = { **YES** | **NO**} ;]
{ **AXIS =** <axis identifier> [<axes range>] [**SUBSET** = < subset number>] **;** }+

END

<identifier>: name of the axes set.

*<axis identifier>: (NA), name of the axis to be included in the axes set. This is the identifier assigned to the axis in the **AXIS** declaration.*

<axes range>: (NA), If the axis is multiple, (member of a nodes group) indicates which axes to include in the axes set.

***[all]** -> indicates that all axes are included*

<subset number>: (NA), used to group the axes within a set into subsets for data updating purposes. Subset number must be a positive integer number (≥ 0). If this option is not used, the default subset number is 0.

i | **SHIFT**, the previous syntax for **SUBSET** is recognised; however, do not use the **SHIFT** syntax in new applications.

EXAMPLE

The following axes were already declared:

```
AXI_X, AXI_Y
```

An axes set representing an X-Y table looks like this:

```
AXES_SET AXS_Table ;          //  X-Y table
   AXIS     = AXI_X ;
   AXIS     = AXI_Y ;
END
```

Powering-on the motor supplies can be programmed like this:

```
AXS_Table <- power_on
```

3.3.3 OPTIMIZE PARAMETER

3.3.3.1 OPTIMIZE = YES.

PAM attempts to form subgroups of "identical" axes within an axis set using the criteria listed below. During execution, commands and set-points are broadcast simultaneously (in the same PAM frame) to all axes in a given subgroup. This significantly reduces execution time for the pipe network because the time needed to service a subgroup of identical axes is the same as to service a single axis. This also reduces the PAM-Ring bandwidth (number of frames) used to transmit set-points and commands. Furthermore, all axes in a given subgroup receive the information at the same time.

When PAM forms multiple subgroups of axes within an axis set, the subgroups are serviced sequentially (in successive PAM frames).

i | A maximum of 15 different subgroups of axes can coexist at the same time in one PAM. Even if the necessary conditions exist to build a sixteenth subgroup, it will not be built.

3.3.3.2 CRITERIA FOR DETERMINING IDENTICAL AXES.

During boot-up, PAM analyses every axis with respect to the following ST1 parameters:

- CKV,
- CKA (amplitude and sign),
- CKR,
- CKH,
- EQUIP

as well as the **PULSES_PER_UNIT** parameter of each **AXIS** declaration and notes their initial values. During run time, as long as an AXES_SET is connected to a converter, PAM treats as identical all axes of the AXES_SET whose above-listed parameters and RATIO are the same.

i | The RATIOS applied to axes of an axes set connected to a converter may be individually adjusted via the **change_ratio** Converter function (see paragraph 8.4).

The application must insure that the status (power_on, run, move in process, etc.) of each axis in the axis set is the same as all the others and, when necessary, remove non-conforming axes from the axis set

3.3.3.3 OPTIMIZE = NO (DEFAULT).

This selection informs PAM not to attempt to form subgroups of identical axes within an axis set for the purpose broadcasting commands and data to identical axes in a single PAM frame. **OPTIMIZE** = **NO** is usually selected when the user knows in advance that the axes do not meet the criteria for optimising. Even without optimisation there are benefits from grouping closely related axes into an axes set; namely, PAM attempts to group sequentially (in successive PAM frames) setpoints and commands resulting from statements whose object is an axes set. An axes set can also simplify the expression of a pipe network ending with several axes which are receiving the same pipe flow data.

3.3.4 SUBSET PARAMETER

3.3.4.1 WHEN SUBSETS SHOULD BE CREATED

When the number of non-identical axes in one axes-set is greater than or near the number of frames in a **BASIC_PAM_CYCLE**, loading of the PAM-Ring imposed by the axes-set becomes severe. Creating subsets, permits PAM to distribute set-point computations and transmissions to the axes-set over several PAM cycles, thereby reducing loading of the PAM ring.

> i All non-indexed axes in an axes set receive their set-points independently which requires one PAM-Ring frame per **BASIC_PAM_CYCLE** for each non-indexed axis in the axes set

3.3.4.2 SERVICING SUBSETS

When a **SUBSET** parameters are used in an **AXES_SET** declaration , PAM forms axes subsets according to the **SUBSET** parameter value in each **AXIS** parameter. Servicing of the axes then proceeds as follows:

- The first subset (axes with **SUBSET** = 1) only is processed in the first **BASIC_PAM_CYCLE**. The second subset only is processed in the next **BASIC_PAM_CYCLE** and so on until all subsets have been processed.
- All axes of the same subset receive their set-points in the same **BASIC_PAM_CYCLE**, while axes of other subsets do not receive set-points.
- All axes in the axes set with no **SUBSET** parameter receive set-points every **BASIC_PAM_CYCLE**.

3.3.4.3 EXAMPLE

Lets imagine a system with 21 axes which must be synchronised, so they are linked to the same pipe network. The first 20 axes belong to an axes set connected to the pipe network and the axis 21 is directly connected to the same pipe network.

The most critical axis is axis 21. It must be updated (receive new setpoint) every millisecond. The other axes can be updated every 4 milliseconds. The **BASIC_PAM_CYCLE** for this system is one millisecond. The solution is to form 4 subsets, each with 5 axes.

The declaration is as follows:

```
AXES_SET AXS_Example ;
   AXIS     = AXI_Regular1  SUBSET=1 ;    // group 1
   AXIS     = AXI_Regular2  SUBSET=1 ;    // group 1
   AXIS     = AXI_Regular3  SUBSET=1 ;    // group 1
   AXIS     = AXI_Regular4  SUBSET=1 ;    // group 1
   AXIS     = AXI_Regular5  SUBSET=1 ;    // group 1
   AXIS     = AXI_Regular6  SUBSET=2 ;    // group 2
   AXIS     = AXI_Regular7  SUBSET=2 ;    // group 2
   AXIS     = AXI_Regular8  SUBSET=2 ;    // group 2
   AXIS     = AXI_Regular9  SUBSET=2 ;    // group 2
   AXIS     = AXI_Regular10 SUBSET=2 ;    // group 2
   AXIS     = AXI_Regular11 SUBSET=3 ;    // group 3
   AXIS     = AXI_Regular12 SUBSET=3 ;    // group 3
   AXIS     = AXI_Regular13 SUBSET=3 ;    // group 3
   AXIS     = AXI_Regular14 SUBSET=3 ;    // group 3
   AXIS     = AXI_Regular15 SUBSET=3 ;    // group 3
   AXIS     = AXI_Regular16 SUBSET=4 ;    // group 4
   AXIS     = AXI_Regular17 SUBSET=4 ;    // group 4
```

```
    AXIS     = AXI_Regular18 SUBSET=4 ;    // group 4
    AXIS     = AXI_Regular19 SUBSET=4 ;    // group 4
    AXIS     = AXI_Regular20 SUBSET=4 ;    // group 4
END
```

The result is as follows:

PAM cycle 1	set-points for axis 21 and axes of subset 1
PAM cycle 2	set-points for axis 21 and axes of subset 2
PAM cycle 3	set-points for axis 21 and axes of subset 3
PAM cycle 4	set-points for axis 21 and axes of subset 4
PAM cycle 5	set-points for axis 21 and axes of subset 5

It occupies only 5 PAM-Ring frames each pipe period. Without subset distribution, the pipe network would have requested 21 frames each pipe period, which is greater than the 20 free frames per millisecond, so not possible in the same PAM cycle.

3.4 SINK

PURPOSE

The sink is a new object whose purpose is to receive values instead of an axis or a PAM Analog Output. The values received are not transformed. A Sink can be used to simulate an axis which is not present at the moment or to verify some pipe position or speed data without performing any physical motion.

DECLARATION SYNTAX

SINK <identifier> ;
END

<identifier> : name of the sink

FUNCTIONS

No executive or inquire functions are defined for the Sink.

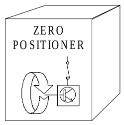
SAMPLE DECLARATION

```
SINK SNK_Dummy;
END
```

3.4.1 USE WITH A CONVERTER

The sink can be used as the destination object of a Converter pipe block. The modes TORQUE, POSITION, SPEED & VALUE are defined.

3.5 ZERO-POSITIONER



3.5.1 PURPOSE

The purpose of the zero-positioner is to position an axis at a user-definable absolute reference position with the aid of a physical reference. This is accomplished by moving the axis while looking at external sensors in order to determine with sufficient precision the “real” position of the axis (or the object controlled by the axis). Although there are a number of ways to perform this task, the Zero Positioner implements zero-positioning utilising the three phase sequence and parameters described below.

3.5.2 DECLARATION SYNTAX

```
ZERO_POSITIONER <identifier> ;
        [{NUMBER                = <number> ¦
        NODES_GROUP             = <nodes group identifier> } ; ]
        OBJECT                  = <axis object identifier> ;
        [SENSOR                 = <binary object identifier> ; ]
        [COARSE_SPEED           = <speed value> ;
         COARSE_EDGE            = <edge value> ;
         COARSE_MOVE            = <move value> ; ]
        [FINE_SPEED             = <speed value> ;
         FINE_EDGE              = <edge value> ;
         FINE_MOVE              = <move value> ; ]
        [RESOLVER_OFFSET        = <resolver offset value> ; ]
END
```

<identifier> : Name of the zero positioner.

*<number>: (NA), indicates if the object is multiple or single. If the **NUMBER** declaration is omitted or if <number> = 1, the object is single, otherwise the object is multiple.*

<nodes group identifier>: (NA), name of a group of nodes

<axis object identifier > : (NA), name of the axis object to be zero-positioned.

<binary object identifier>: (NA), name of the binary input to be sensed in Phases 1 and 2. If no sensor is declared, Phase 1 and 2 are omitted.

*<speed value> : (RW), speed value used for the COARSE or FINE phase, if present. If **CURRENT** is used for the speed value, the current **AXIS_TRAVEL_SPEED** value is used.*

*<edge value> : (NA), active edge of the sensor used for the COARSE or FINE phase, if present. The edge value can be **RISING** or **FALLING**.*

*<move value> : (RW), the distance to move relative to axis position after sensor detection for the COARSE or FINE phase, if present. The move value can be **NONE** if no move is requested.*

<resolver offset value> : (RW), offset of the resolver given in user length units.

⚠ The defined sequence (i.e. phases implemented via optional parameter specification) cannot be modified dynamically. If different sequences are needed, multiple zero positioners must be declared.

FUNCTIONS

– **start**

This function starts the Zero-Positioner. Phases are executed in the following order: coarse phase, fine phase, resolver phase.

– **stop**

This function stops the zero-positioner and all other functions running on the corresponding axis. This function is identical to "<axis object> <- stop ;".

DECLARATION EXAMPLES

Zero positioner with fine and resolver phases:

```
ZERO_POSITIONER ZEP_Main ;
   OBJECT =                  AXI_Main;
   SENSOR =                  SBI_MainZeroSensor ;
   FINE_SPEED =        720.0 ;
   FINE_EDGE =         RISING ;
   FINE_MOVE =         180.0 ;
   RESOLVER_OFFSET     =     240.0 ;
END
```

Zero positioner with coarse and fine phases:

```
ZERO_POSITIONER ZEP_Main ;
   OBJECT =                  AXI_Main ;
   SENSOR =                  SBI_MainZeroSensor ;
   COARSE_SPEED =            CURRENT ;
   COARSE_EDGE =       FALLING ;
   COARSE_MOVE =       NONE ;
   FINE_SPEED =        720.0 ;
   FINE_EDGE =         RISING ;
   FINE_MOVE =         180.0 ;
END
```

3.5.3 ZEROING SEQUENCE

3.5.3.1 GENERAL INFORMATION

The axis zeroing sequence is performed in the order phase 1, phase 2, phase 3. If the parameters for a phase are omitted, the corresponding phase is not executed.

Any combination is allowed, even though some combinations are practically meaningless. At the end of any chosen zeroing sequence, the axis absolute position is cleared to zero. If a non-zero absolute position is needed, it can be initialised to any value using the "position" executive function.

3.5.3.2 PHASE 1 - COARSE PHASE WAITING FOR BOOLEAN SENSOR.

ZERO_POSITIONER ZERO_POSITIONER The motor is ramped up to **COARSE_SPEED** in the direction specified by the sign of this parameter. Whether the given speed is reached or not, PAM is waiting for the next **COARSE_EDGE** transition of the binary input object defined by **SENSOR**.

ZERO_POSITIONER When the specified sensor transition is detected, a relative move of distance specified by **COARSE_MOVE** is performed. The position reference for the relative move is the axis position when the sensor's transition is detected. Therefore, target position becomes the axis position at sensor transition plus **COARSE_MOVE**. The relative move part of the phase may be omitted by setting **COARSE_MOVE** = **NONE**.

Note that if the distance of the relative move is less than the displacement necessary to stop the motor, the motor will go "too far", then come back to the target position. Note also the difference between **COARSE_MOVE** = 0.0 in which the motor ramps down then goes back to the position of the sensor's transition and **COARSE_MOVE** = **NONE** in which the motor simply stops.

By setting **COARSE_SPEED** = **CURRENT**, the Zero Positioner uses axis **TRAVEL_SPEED,** current value, for it's coarse speed value. Current axis **ACCELERATION** and **DECELERATION** parameter values set the acceleration and deceleration rates used during the coarse move.

3.5.3.3 PHASE 2 - FINE PHASE WAITING FOR BOOLEAN SENSOR.

The fine phase works in exactly the same manner as the coarse phase (see paragraph 3.5.3.2) using an equivalent "FINE" phase parameter set. However, the intended purpose for phase 2 is different. The purpose of the first phase is to detect the sensor position coarsely and quickly. When the sensor's position is reached, the final move of the first phase places the object near the sensor. The second phase is generally performed at a slower speed and over a shorter distance, resulting in less uncertainty of final axis position upon completion of phase 2.

> i | To avoid all sensors in large machines, switching at the same time (generating delays and then inaccuracies) the overall number of motors running in the Zero-Positioner fine phase is limited automatically by PAM. Additional Zero-Positioners in this phase are delayed until one of the currently running ones terminates its fine phase. The upper limit is 6 zero-positioners in the fine phase at the same time.

3.5.3.4 PHASE 3 - RESOLVER POSITIONING

ZERO_POSITIONER The resolver positioning phase is used to place the motor shaft exactly at a specific angular position. This phase performs a move to an absolute angular position specified by **RESOLVER_OFFSET**. The current axis **TRAVEL_SPEED**, **ACCELERATION** and **DECELERATION** are used.

> ! | The zero_positioner cannot be executed while a pipe is active on the same axis.

3.5.4 APPLICATION EXAMPLE

The following example defines the user length unit (ULU) for *my_axis* is 1 degree at motor's shaft:

```
AXIS AXI_Main;
   NODE  = NOD_Main;
   PULSES_PER_UNIT   = 11930464.71111111; // 2^32/360->ULU=1degree
   TRAVEL_SPEED      = 3600.0    // 10 rev. per sec
```

```
   ACCELERATION      = 36000.0   // 100 rev. per sqr. sec.
   DECELERATION      = 72000.0   // 200 rev. per sqr. sec.
   INITIAL_POSITION  = 0.0
   POSITION_PERIOD   = 360.0     // 1 revolution
END
```

The zero positioner is defined a follows:

```
ZERO_POSITIONER ZEP_Main;
   OBJECT      = AXI_Main;
   SENSOR      = SBI_Mainreference;
   // Looks for the reference revolution ( given by
   SBI_Mainreference)
   COARSE_SPEED      = 3600.0;   // 10 rpsec, positive way
   COARSE_EDGE = RISING;
   COARSE_MOVE = 0.0;            // goes back to detection     // Into
this turn goes the position 100 degree with respect to  the resolver
   RESOLVER_OFFSET   = 100.0;
END
```

The following part of a sequence executes the zero-positioning cycle (see Figure 3-2).

```
...
Change the default travel speed:
   AXI_Main <- travel_speed(720.0);   //2 rpsec
Now perform the specified job with the statement :
   ZEP_Main <- start;
And wait the job's completion to set led_1
   CONDITION ZEP_Main ? ready;
   SBI_led1 <- set;
```

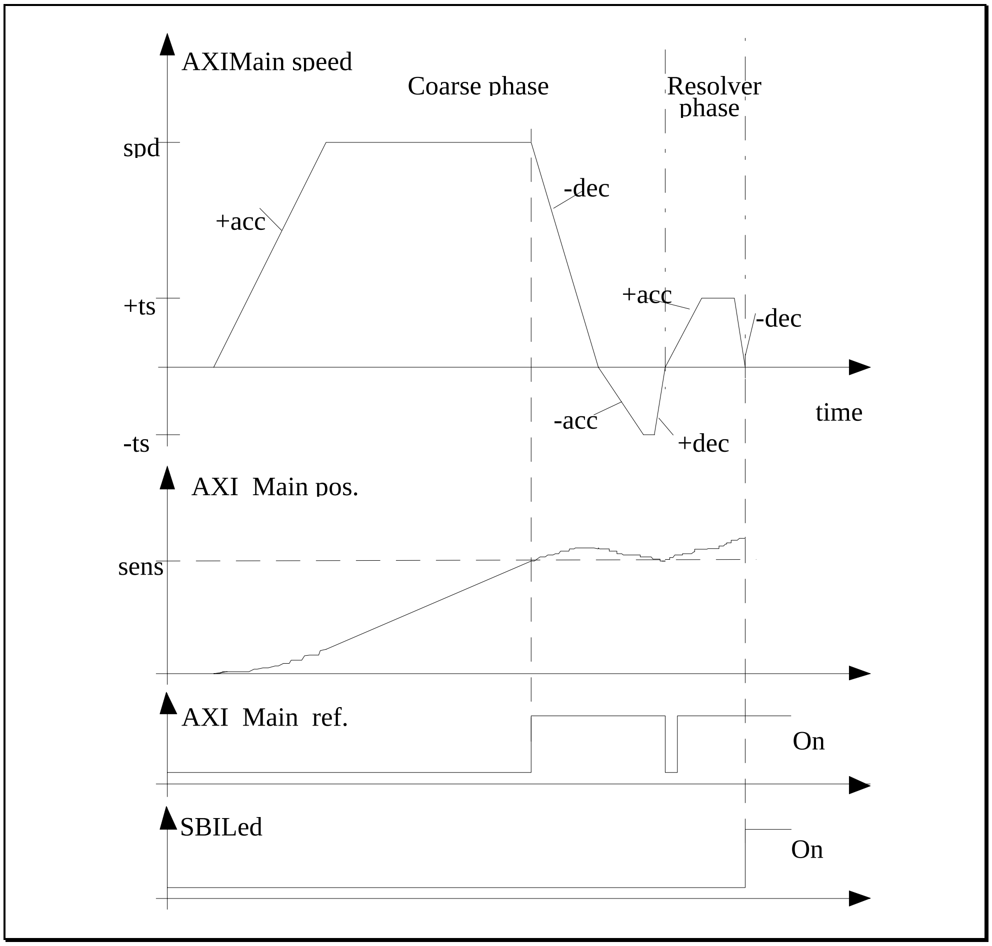


Figure 3-2 Axis motion corresponding to Zero-Positioning Example

acc is the current acceleration value;

dec is the current deceleration value;

ts is the current travel speed value;

spd is the specified coarse speed;

sens is the position of the rising edge of the sensor.

3.6 VARIABLES

3.6.1 INTRODUCTION

Variables are application objects which define the specific application-related information that PAM must manage. Three categories of variables are defined:

DUALPORT VARIABLES

Dual-port variables are directly accessible to an external main controller (industrial computer, programmable controller etc.) for system level control and monitoring.

i | Dualport variables are discussed in chapters 13, 14 and 15} in this manual.

INTERNAL VARIABLES

Internal variables are similar to the variables of classical programming languages. They are used to hold data such as state of the machine process variables, etc. or to improve the structure of the program. Variables can be named, modified and used anywhere in the program (in sequences, Boolean equations, etc.). Changes in internal variable's value/state can be events (i.e. trigger actions, conditions, exceptions etc.). They reside in PAM memory.

COMMON VARIABLES

Common variables are similar to internal variables, but do <u>not produce event</u>. Common variables can be used, for instance, as loop counter or for local computations.

3.6.2 GENERAL SYNTAX

The general declaration syntax for all common and internal variables is as follows:

<location> <type> <identifier> **;**
    [{**NUMBER** **=** <number> ¦
    NODES_GROUP = <nodes group identifier> } ;
    <characteristics>
END]

*<location>: location of the variable ({ **DUALPORT_IN** ¦ **DUALPORT_OUT** ¦ **COMMON** ¦ **INTERNAL** }).*

*<type>: type of the variable ({ **FLAG_VAR** ¦ **WORD_VAR** ¦ **REAL_VAR** }).*

<identifier>: name of the variable.

*<number>: (NA), indicates if the object is multiple or single. If the **NUMBER** declaration is omitted or if <number> = 1, the variable is single, otherwise the variable is multiple.*

*<nodes group identifier>: (NA), name of the nodes group. The **NUMBER** parameter from the specified nodes group declaration is used as **NUMBER** in the variable declaration.*

<characteristics>: (NA), characteristics of the variable, depends of location and type.

3.6.3 INTERNAL VARIABLES

GENERAL SYNTAX

INTERNAL { **FLAG_VAR** | **WORD_VAR** | **REAL_VAR** } <identifier> ;
[{**NUMBER** = <number> |
NODES_GROUP = <nodes group identifier> }] ;
END]

EXAMPLES

```
INTERNAL FLAG_VAR IFV_SystemInOperation ;  // State of the machine


INTERNAL REAL_VAR IRV_Area ;
   NUMBER = 8 ;
END


INTERNAL WORD_VAR IWV_HeadPosition ;
   NODES_GROUP = NGR_Heads ;
   END ;
```

3.6.3.1 INTERNAL FLAG VARIABLE

SYNTAX

INTERNAL FLAG_VAR <identifier> **;**
[{ **NUMBER** = <number> |
NODES_GROUP = <nodes group identifier> } ;
END]

SIZE

Boolean (1bit)

FUNCTIONS

FUNCTION NAME	SET/ INQUIRE	PARAMETERS	RETURN VALUE
<- (assign value)	SET	- boolean expression	-
invert	SET	-	-
reset	SET	-	-
set	SET	-	-
<- (read value)	INQUIRE	-	BOOLEAN

3.6.3.2 INTERNAL WORD VARIABLE

SYNTAX

INTERNAL WORD_VAR <identifier> **;**
[{ **NUMBER** = <number> |
NODES_GROUP = <nodes group identifier> } ;
END]

SIZE

Long Word (signed 32 bits)

FUNCTIONS

FUNCTION NAME	SET/ INQUIRE	PARAMETERS	RETURN VALUE
<- (assign value)	SET	- numerical expression	-
<- (read value)	INQUIRE	-	INTEGER

3.6.3.3 INTERNAL REAL VARIABLE

SYNTAX

INTERNAL REAL_VAR <identifier> **;**
[{ **NUMBER** = <number> |
NODES_GROUP = <nodes group identifier> } ;
END]

SIZE

Real (64 bits, floating point, as IEEE 754-1985)

FUNCTIONS

FUNCTION NAME	SET/ INQUIRE	PARAMETERS	RETURN VALUE
<- (assign value)	SET	- numerical expression	-
<- (read value)	INQUIRE	-	REAL

3.6.4 COMMON VARIABLES

SYNTAX

COMMON { **FLAG_VAR** | **WORD_VAR** | **REAL_VAR** } <identifier> ;
[{ **NUMBER** = <number> |
NODES_GROUP = <nodes group identifier> } ;
END]

DECLARATION EXAMPLE

```
COMMON FLAG_VAR CFV_MyFlag ;    /* NUMBER = 1 */


COMMON REAL_VAR CRV_AxisOffset ;
   NUMBER = 8 ;
END


COMMON WORD_VAR CWV_LoopCounter ;
   NODES_GROUP = NGR_Heads ;
END ;
```

3.6.4.1 COMMON FLAG VARIABLE

SYNTAX

COMMON FLAG_VAR <identifier> **;**
[{ **NUMBER** = <number> |
NODES_GROUP = <group identifier> } ;
END]

SIZE

Boolean (1bit)

FUNCTIONS

FUNCTION NAME	SET/ INQUIRE	PARAMETERS	RETURN VALUE
<- (assign value)	SET	- boolean expression	-
invert	SET	-	-
reset	SET	-	-
set	SET	-	-
<- (read value)	INQUIRE	-	BOOLEAN

3.6.4.2 COMMON WORD VARIABLE

SYNTAX

COMMON WORD_VAR <identifier> **;**
[{ **NUMBER** = <number> |
NODES_GROUP = <group identifier> } ;
END]

SIZE

Long Word (signed 32 bits)

FUNCTIONS

FUNCTION NAME	SET/ INQUIRE	PARAMETERS	RETURN VALUE
<- (assign value)	SET	- numerical expression	-
<- (read value)	INQUIRE	-	INTEGER

3.6.4.3 COMMON REAL VARIABLE

SYNTAX

COMMON REAL_VAR <identifier> **;**
[{ **NUMBER** = <number> |
NODES_GROUP = <group identifier> } ;
END]

SIZE

Real (64 bits, floating point, as IEEE 754-1985)

FUNCTIONS

FUNCTION NAME	SET/ INQUIRE	PARAMETERS	RETURN VALUE
<- (assign value)	SET	- numerical expression	-
<- (read value)	INQUIRE	-	REAL

3.7 BOOLEAN EQUATIONS

3.7.1 INTRODUCTION

Boolean equations perform logical combinations of binary (or Boolean) variables and may include the result of a comparison of numerical variables.

The general properties of Boolean equations are:

- As for other variables, the result of a Boolean equation can be invoked in other parts of a program simply by using its name.
- A Boolean equation is a read only variable.
- A binary output can be linked to a Boolean equation. This means that any change of Boolean equation value is reported automatically to the linked binary output.
- A boolean equation can have only two possible results, true or false.
- Boolean equations can be considered as a class of variables.

3.7.2 DECLARATION SYNTAX

BOOLEAN <identifier> [<multiple tag>]**;**
    [**LINKED_OUTPUT =** <output object identifier> **;**]
    EQUATION **=** <boolean expression>**;**
END_BOOLEAN

<identifier>: name of the boolean equation.

<multiple tag> : if the equation is multiple, the ***[i]*** *tag must follow the identifier.*

<output object identifier>: (NA), the identifier of an output object. The output object must be one of the following:
- *a binary output.*
- *a led output.*
- *a dualport flag output.*

<boolean expression>: (NA), an expression composed of boolean objects, boolean operators, boolean inquire functions and comparison expressions.

FUNCTIONS

FUNCTION NAME	SET/ INQUIRE	PARAMETERS	RETURN VALUE
<- (read value)	INQUIRE	-	BOOLEAN

3.7.3 OBJECTS THAT CAN BE USED IN BOOLEAN EQUATIONS

All of the following Boolean objects can be used in Boolean expressions:

- Binary input.
- Keyboard input.
- Dualport flag input variable.
- Internal flag variable.
- Common flag variable.
- Boolean equation.
- Binary output.
- Dualport flag output variable.

3.7.4 BOOLEAN OPERATORS

The Boolean operators are as follows:

+	the OR operator.
*	the AND operator.
!	the NOT operator.
[**all+]**	the multiple OR operator.
[**all*]**	the multiple AND operator.

The **[all+]** operator means "do the Boolean OR of all items of the multiple object or expression". The result is a single object.

The **[all*]** operator means "do the Boolean AND of all items of the multiple object or expression". The result is a single object.

3.7.5 INQUIRE FUNCTIONS IN BOOLEAN EQUATIONS

Some inquire functions can be used in Boolean expressions if they return a Boolean value. They are as follows:

error	ask if object is in error.
ready	ask if object is ready.
error_code (<value>)	ask if the error "value" occurs
status (<value>)	ask if the status "value" is active
triggered	ask if object has triggered

3.7.6 COMPARISON EXPRESSIONS IN BOOLEAN EQUATIONS

Comparison expressions (expressions composed of numerical expressions and comparison operators) may be included in Boolean equations.

3.7.6.1 COMPARISON OPERATORS

The following comparison operators may be used in comparison expressions:

>	the "greater than" operator.
<	the "less than" operator
>=	the "greater than or equal" operator
<=	the "less than or equal" operator
=	the "equal" operator"
<>	the "non equal" operator

3.7.7 WRITING BOOLEAN EXPRESSIONS

Parenthesis must be used to specify the order of precedence in evaluating Boolean expressions. Parenthesis may also be used to improve readability.

3.7.8 BOOLEAN EQUATION DECLARATION EXAMPLES

```
BOOLEAN BOL_equation1;
   EQUATION
   SBI_a * IFV_b * (SBI__c + CFV_d) ;
END_BOOLEAN

BOOLEAN BOL_equation2 [i];
   EQUATION
   IFV_e[i] * CFV_f[i] * (CFV_g[i] + IFV_h[i]) ;
END_BOOLEAN

BOOLEAN BOL_equation3;
   EQUATION
   (IFV_e[i] * CFV_f[i])[all+] * (SBI_c + CFV_d) ;
END_BOOLEAN

BOOLEAN BOL_equation4;
   EQUATION
   IFV_e[all+] * CFV_f[all*] * !(SBI_c + CFV_d) ;
END_BOOLEAN

BOOLEAN BOL_equation5;
   EQUATION
   (AXI_axes[i] ? ready)[all*] * !(DCM_motor ? error) ;
END_BOOLEAN

BOOLEAN BOL_equation_6[i];
   LINKED_OUTPUT = SBI_GreenLed ;
   EQUATION
   (AXI_axes[i] ? speed  >= 123.5) * !(AXI_axes[i] ? error) ;
END_BOOLEAN

BOOLEAN BOL_equation_7[i];
   LINKED_OUTPUT = SBI_out1;
   EQUATION
   IFV_e[i] + !SBI_c  ;
END_BOOLEAN
```

3.8 ACTIONS GROUP

3.8.1 INTRODUCTION

Actions represent one type of object belonging to the executive part of the PAM application language.). Actions exist outside the Task/Sequence structure. They are intended to perform operations which must be executed without regard to the state of tasks or sequences. Actions are intended to perform operations which execute quickly (assignment statements or elementary actions); therefore, Service statements like **CONDITION**, **WAIT-TIME**, **EXCEPTION** and **CASE** are not allowed in **ACTIONS** objects.

3.8.2 DECLARATION SYNTAX

```
ACTIONS <identifier> ;
        SPECS
                [{NUMBER                = <number> ¦
                NODES_GROUP             = <nodes group identifier> } ; ]
                CYCLES = <cycles number> ;
        END_SPECS
        { {ON_EVENT¦ON_STATE} <boolean expression> ACTION
                {<action statement>}+
         END_ACTION
        }+
        [ POWERON ;
                {<action statement>}+
         END_POWERON
        ]
END_ACTIONS
```

<identifier>: name of the actions group.

*<number>: (NA), indicates if the object is multiple or single. If **NUMBER** is omitted or if <number> = 1, the object is single, otherwise the object is multiple.*

<nodes group identifier>: (NA), the name of a group of nodes

*<cycles number> : (NA), the servicing interval for the actions group expressed in **BASIC PAM CYCLES**.*

<Boolean expression>: an expression composed of Boolean objects, Boolean operators, Boolean inquire functions and comparison expressions.

<action statement> : any assignment statement

3.8.3 FUNCTIONS

No functions are available for the Actions group.

3.8.4 CYCLES SPECIFICATIONS

The **CYCLES** specification defines the number of BASIC_PAM_CYCLEs between successive executions of the actions group. For actions specified for "**ON_STATE**" activation, the cycles parameter permits regulation of the frequency of execution of the action. For actions with an "**ON_EVENT**" specification, the cycles parameter establishes the maximum delay (number of PAM basic cycles) between an event's detection and execution of the corresponding action.

3.8.5 ACTIONS

Two types of action may be specified. When **ON_EVENT** is specified in an **ACTION** declaration, the specified action statement is executed once each time the associated boolean expression becomes true. When **ON_STATE** is specified, the specified action statement is executed cyclically as for as long as the associated boolean expression is true.

3.8.6 POWERON ACTIONS

POWERON actions are a special type of Action statement which are executed only <u>once</u> after a <u>power on or a reset of PAM</u>. They are used for system initialisation purposes and for establishing starting/restarting conditions. Normal Task/Action execution is started only when all **POWERON** Actions and **POWERON** Sequences within TASKs have finished executing.

3.9 ROUTINES GROUP

The Routines Group, which is similar to an Actions group at the level of their statement types, represents one type of the executive part of the PAM application language. They are intended to perform operations which execute quickly. (i.e. assignment statements or elementary actions).

A **ROUTINE** object can have up to 4 parameters. Each parameter is formally defined in the **ROUTINE** declaration by a parameter identifier. The parameter identifiers can be used anywhere within the routine itself. Actual parameters (included as arguments of the routine calling statement) are substituted for the parameter identifiers when the routine is executed. That means no parameters values can be return by the routine.

Routine execution is started in one of two ways; by connecting it to a **COMPARATOR** or **MULTI_COMPARATOR** pipe block (see paragraph 3.9.4), or by calling it from within a **SEQUENCE**, **ACTION** or another **ROUTINE** (see paragraph 3.9.3).

Service statements like **CONDITION**, **WAIT-TIME**, **EXCEPTION** and **CASE** are not allowed in **ROUTINES**.

3.9.1 DECLARATION SYNTAX

ROUTINES <routines group identifier> **;**
    { **ROUTINE** <routine identifier> [**WITH** <param ident>**,** <param ident>**,** ...] **;**
    { <routine statement> }+
    END_ROUTINE
    }+

    [**POWERON**
    { <routine statement> }+
    END_POWERON
    ]
END_ROUTINES

<routines group identifier> : name of the group of subsequent routines objects (string of characters).

<routine identifier> : name of the routine object (string of characters).

<param ident> : formal parameter identifiers (string of characters). The number of formal parameters must be between 0 and 4..

<routine statement> : any assignment statement.

3.9.2 FUNCTIONS

No functions are available for routines.

3.9.3 ROUTINE CALL

Routine execution can be started by calling it from any **SEQUENCE**, **ACTION** or **ROUTINE** (excepted itself) using a **CALL** statement. When a routine is called, it is necessary to specify in the routine call statement the same number of parameters as specified in the routine declaration. Actual parameters are linked to parameter identifiers by the order in which they are listed in the **CALL** statement.

Parameter expressions are evaluated just before starting routine execution, and the actual parameter current values substituted for the parameter identifiers specified in the routine declaration. When routine execution is completed, the application continues with the statement immediately following the **CALL** statement. No parameters values are returned.

A routine must be declared before it can be called in a CALL statement.

No parameters values are returned by a routine

3.9.3.1 CALL STATEMENT SYNTAX

CALL <routine identifier> [**WITH** <param expression>**,** <param expression>**,** ...] **;**

<routine identifier> : name of a routine object (string of characters).

<param expression> : actual parameter expression. It can be any type of expression. The number of parameters expressions can be between 0 and 4. The number of parameter expressions must match the number of parameters declared in the Routine declaration.

3.9.4 ROUTINE CONNECTED TO A COMPARATOR

When a routine is connected to a **COMPARATOR** or a **MULTI_COMPARATOR** pipe block, its execution is automatically started by PAM firmware immediately upon occurrence of the related event, therefore providing a very short reaction time. Routine execution takes place immediately after the pipes & ring processing in the current PAM cycle prior to continuing execution of the application sequences and actions.

If PAM is overloaded at this time, routine execution is deferred until next sequence or action execution interruption. This type of interruption occurs at each condition transition, wait time, end of a loop, end of a sequence, end of an action, exceptions and at some internal firmware conditions.

In most cases, routine execution will occur within the current PAM cycle; however, it may be necessary to add some transition conditions in long sequences without execution interruption to insure that reaction time does become too long.

3.9.5 ROUTINE EXAMPLE

```
ROUTINES RGR_SetOfRoutines ;

   ROUTINE RTN_SendValue WITH IRV_Amplitude, IWV_Mode ;
      IFV_ValueModified <- set ;
      SAO_AnalogOut <- IRV_Amplitude * IRV_MyRatio ;
      IWV_ModeOut <- IWV_Mode ;
      IFV_UpdateValue <- set ;
   END_ROUTINE
```

```
END_ROUTINES
...
SEQUENCE SEQ_Example ;
   ...
   CALL RTN_SendValue WITH IRV_AmplitudeObjectOne/10, IWV_ModeObjectOne
;
   ...
END_SEQUENCE
```

3.10 TASKS

3.10.1 INTRODUCTION

Tasks represent the main type of object used in the executive part of the PAM application language. A task describes the runtime behaviour of a component in an application program.

Tasks are intended to execute operations (active statements), sequenced by service statements and grouped in SEQUENCEs. Sequences are started when a pre-defined event occurs (ON_EVENT).

3.10.2 DECLARATION SYNTAX

TASK <identifier> ;

SPECS
[{**NUMBER** = <number> ¦
NODES_GROUP = <nodes group identifier> } ;]
[**DUPL** = <duplication number> ;]
CYCLES = <cycles number> ;
[**DEFAULT_SEQUENCE_WORKSPACE** = <default sequ. workspace>;]
END_SPECS

[**EVENTS**
{**ON_EVENT** <boolean expression> **XEQ_SEQUENCE**
<sequence identifier> <multiple tag> ; }+
END_EVENTS]

{ **SEQUENCE** <sequence identifier> <multiple tag> ;
{<sequence statement>}+
END_SEQUENCE }*

[**POWERON** <multiple tag> ;
{<sequence statement>}+
END_POWERON]

END_TASK

<identifier>: name of the task.

*<number>: (NA), indicates whether the object is multiple or single. If the **NUMBER** declaration is omitted or if <number> = 1, the object is single, otherwise the object is multiple.*

<nodes group identifier>: (NA), name of a group of nodes

<duplication number>: (NA), maximum number of copies of the same task allowed to be active at the same time.

<cycles number>: (NA), period of the task expressed in basic PAM cycles.

<default sequ. workspace>: (NA), workspace size (expressed in bytes) for all sequences of the task. If not specified, the general default value is used.

<Boolean expression>: an expression composed of Boolean objects, Boolean operators, Boolean inquire functions and comparison expression.

<sequence identifier>: name of the sequence.

<multiple tag>: if the equation is multiple, the ***[i]*** *tag must follow the identifier.*

<sequence statement>: any active or service statement.

3.10.3 FUNCTIONS

No functions are available for tasks.

3.10.4 TASKS STATES AND RULES

Tasks (under control of the PAM executive software) may be in one of several states. Definitions of the task states and rules for changing states are as follows:

- A task is active when one of its sequences is active.
- A task is suspended when one of its sequences is suspended.
- A task is alive when one of its sequences is active or suspended.
- A task is dead when none of its sequences are alive or suspended.
- Only one sequence of a task can be alive at the same time. this means that the sequences within a task are mutually exclusive.
- Several tasks can be alive at the same time.

3.10.5 TASK SPECIFICATIONS

3.10.5.1 DUPL

If a task is multiple (**m** times), virtually **m** copies (1,2,3,..,**m**) of the task may be actives simultaneously. The **DUPL** declaration is used, if necessary (ex. available power limitation) to control the maximum number of copies alive simultaneously. When the maximum number of active copies is reached, each new copy waits for the completion of a running copy.

This feature is activated by the service statement **CONDITION DUPL_START** which can be placed anywhere in a sequence.

3.10.5.2 CYCLES

The **CYCLES** declaration is used to indicate the typical reaction time (time between event detection and start of task execution), expressed in PAM basic cycles, for activation of a task when it is in the suspended or dead states.

3.10.6 EVENTS

The **EVENTS** section of the task declaration defines the enabling event for each sequence within the task.

The following declaration establishes a link between an event (described by <boolean expression>) and a sequence of the task identified by a <sequence identifier>.

ON_EVENT <boolean expression> **XEQ_SEQUENCE**

<sequence identifier> ;

When the event occurs, if no other sequence of the task is running, the corresponding sequence is started, otherwise the event is not handled and lost.

If the boolean expression is multiple, the corresponding sequence must also be multiple, with the same number of items.

3.10.7 SEQUENCES

A sequence is a set of active or service statements which are processed one after another.

DECLARATION SYNTAX

SEQUENCE <identifier> <multiple tag>;
    [**SPECS**
        WORKSPACE = <workspace> ;
    END_SPECS]
    ...
END_SEQUENCE

<identifier>: name of the sequence.

*<multiple tag>: if the sequence is multiple, the **[i]** tag must follow the identifier.*

<workspace> : (NA), workspace size (expressed in bytes) of the sequence. If not specified, the default sequence workspace is used.

3.10.7.1 SEQUENCE STATES AND RULES

A sequence can be in the following states:

- A sequence is active when one of its active statements is being processed.
- A sequence is suspended when it is waiting on a transition condition (service statements)
- A sequence is dead once its end has been reached, or it has been aborted, or if it has not been activated.
- A sequence is alive if is active or suspended.

Two different sequences located in two different tasks can have the same name.

If a sequence is multiple, all other sequences of the task (except the "power on" sequence) must also be multiple, with the same number of items.

3.10.7.2 POWERON SEQUENCE

A **POWERON** sequence is executed only once after a power on or a reset of PAM. All **POWERON** sequences of all tasks are intend for system initialisation purposes and for establishing starting/restarting conditions. Normal Task/Action execution is started only when all **POWERON** Actions and **POWERON** Sequences within TASKs have completed execution.

3.10.8 TASK EXAMPLES

Declaration of a single task:

```
TASK TSK_First ;

   SPECS
      PERIOD = 10 ;
   END_SPECS
      EVENTS
      ON_EVENT IFV_Event1 XEQ_SEQUENCE SEQ_First ;
      ON_EVENT IFV_Event2 XEQ_SEQUENCE SEQ_Second ;
```

```
   END_EVENTS
   SEQUENCE SEQ_First ;
   ...
   END_SEQUENCE

   SEQUENCE SEQ_Second ;
         ...
   END_SEQUENCE

   POWERON
         ...
   END_POWERON

END_TASK
```

Declaration of a multiple task:

```
TASK TSK_Second ;

   SPECS
      PERIOD = 10 ;
   END_SPECS


   EVENTS
      ON_EVENT IFV_Event_3[i] XEQ_SEQUENCE SEQ_First[i] ;
      ON_EVENT IFV_Event_4[i] XEQ_SEQUENCE SEQ_Second[i] ;
   END_EVENTS

   SEQUENCE SEQ_First[i] ;
         ...
   END_SEQUENCE

   SEQUENCE SEQ_Second[i] ;
         ...
   END_SEQUENCE

   POWERON
         ...
   END_POWERON

END_TASK
```

4 STATEMENTS

4.1 INTRODUCTION

Statements are the basic building blocks of the executive part of an application. Two types of statements are defined: active statements and service statements. Active statements define how a component functions. Service statements determine when a component functions.

Active statements can be further classified as Object Access or flow control statements. Service statements are categorised as Transition Condition or Exception statements.

4.2 GENERAL DEFINITIONS

Statements are composed of combinations of expressions and keywords. In the following statement:

```
WAIT_TIME IWV_VarDuration / 10 ;
```

"**WAIT_TIME**" is a keyword and "IWV_VarDuration / 10" is an expression.

EXPRESSION

An expression is generally composed of operands and operators. Expressions are evaluated by applying operators to operands in a defined order, generally from left to right.

FUNCTION

Functions are a set of PAM key words which prescribe specific operations to be performed on objects.

KEYWORD

A keyword is a pre-defined, reserved word within the PAM programming language. A keyword cannot be used for user definitions such as objects names. A listing of keywords is provided in appendix A.

OPERAND

Operands can be constants, variables or functions.

OPERATOR

Operators can be any mathematical function (i.e. addition, cosine, square root) or boolean function (i.e. AND, OR).

4.3 OBJECT ACCESS STATEMENT

An Object access statement is a type of active statement consisting of a left part which designates a destination or source object, and a right part which describes the action to be performed on this object.

4.3.1 GENERAL SYNTAX

The general syntax to access any field of an object is as follows:

<object> **operator** <function name>[(parameters)]**;**

i | Object parameters are accessed using Parameter Access statements (see paragraph 4.3.4).

4.3.2 FUNCTIONS

Functions are used to control, command and monitor objects. Two types of functions, Executive Functions and Inquire Functions, are defined. Executive Functions send commands and data to objects, while inquire functions interrogate objects resulting in the object returning status or data. Most objects have an associated set of functions which are listed in the object description. Detailed descriptions of the more complex functions and those functions common to a number of objects are provided in subsequent chapters of this manual. Table 4-1 provides chapter references to function details for various categories of objects. A third type of functions, mathematical functions are described in chapter 6.

OBJECT CATEGORY	FUNCTION REFERENCE INFORMATION	CHAPTER
physical objects (except axis object)	I/O functions error functions	11 12
axis object	executive functions inquire functions	7 9
pipe blocks	executive functions inquire functions	8 10

Table 4-1 Functions Reference Information Locations

SYNTAX

The general syntax for executive function is as follows:

<object> **<-** <function name>[(parameters)]**;**

The syntax for inquire function is as follows:

<object> **?** <function name>[(parameters)]**;**

EXAMPLES:

```
AXI_X <- relative_move (158) ;
AXI_X ? error_code ;
```

4.3.2.1 OBJECT VALUE ACCESS FUNCTIONS

Object value can be accessed by two specific functions: **getvalue** and **setvalue**. To simplify use of these statements, the names of these functions may be omitted. In that case, the statement syntax where the target object is a destination (information is going to the object) is as follows:

<target object> **<-** <expression> **;**

The following combinations of object class/ variable class are valid:

<boolean object> **<-** <boolean expression> **;**

<integer object> **<-** <{integer expression¦real expression}> **;**
<real object> **<-** <{integer expression¦real expression}> **;**

<boolean|integer|real object> <- <object>

EXAMPLE

```
BWV_MyInputA <- SBI_InputA ;
```

In integer object ← real expression operations, the fractional portion of the expression is truncated. In real object ← integer expression operations, zeros are inserted to the right of the decimal point.

The statement syntax where the target object is a source (information is coming from the object) is as follows:

<target object> **?** <expression> **;**

This syntactic structure is used as a component of an Inquire-Set Value combination statement (see paragraph 4.3.3) which transfers an inquired value (with or without modification) to a final destination object.

4.3.3 INQUIRE-SET VALUE COMBINATION STATEMENT

An inquire function can be combined with a set value operation in a single statement which permits transfer of the value returned by an inquire function to a final destination object (Boolean, integer or real).

SYNTAX

<boolean|integer|real object> **<-** <object> **?** <function name>[(parameters)]**;**

EXAMPLE:

```
IWV_CurrentError <- AXI_X ? error_code ;
```

4.3.4 PARAMETER ACCESS STATEMENTS

Some object parameters, principally physical and pipe object parameters, can be accessed by an application during its execution using a specific parameter access statement syntax. Every parameter has its own access level which is indicated in the parameter description. The possible levels with their abbreviations are as follows:

- No Access	(NA)
- Read Only access	(RO)
- Write Only access	(WO)
- Read and Write access	(RW)

Only those parameters with the necessary access level can be accessed using the parameter access statement. Where specific rules or special considerations apply to parameter access for a particular object, they are included with the object description.

4.3.4.1 PARAMETER INQUIRY SYNTAX

The syntax for parameter value inquiry is as follows:

<destination object> **<-** <object>**:**<parameter> ;

EXAMPLE:

Read current TRAVEL_SPEED parameter value of AXI_X and write the value into IWV_MyVariable:

```
IWV_MyVariable <- AXI_X:TRAVEL_SPEED ;
```

4.3.4.2 PARAMETER MODIFICATION SYNTAX

The syntax for parameter value modification is as follows:

<object>**:**<parameter> **<-** <expression> ;

EXAMPLE:

Modify the TRAVEL_SPEED parameter value of AXI_X:

```
AXI_X:TRAVEL_SPEED <- 123.75 * (IWV_OldSpeed - 100.0) ;
```

4.4 FLOW-CONTROL STATEMENTS

4.4.1 INTRODUCTION

Flow-control statements are a type of active statement. They provide the capability to modify program flow under defined circumstances. There are four types of flow-control statements which include:

- **IF THEN ELSE ENDIF** statement structure
- **LOOP END_LOOP** statement structure
- **XEQ_TASK** statement
- **XEQ_SEQUENCE** statement

4.4.2 IF ... THEN ... ELSE ... ENDIF

This statement structure is used to select one program path or another based upon some condition.

SYNTAX

IF <boolean expression> **THEN**

...

[**ELSE**]

...

END_IF

<boolean expression> : an expression composed of Boolean objects, Boolean operators, Boolean inquire functions and comparison expressions.

EXAMPLE

If StopInput is true processing 1 is executed, otherwise processing 2.

```
SEQUENCE SEQ_ExampleIf
   ...
   IF SBI_StopInput THEN
      IFV_StopMachine <- set ;        // processing 1
   ELSE
   ...IFV_StopMachine <- reset ;// processing 2
   END_IF
END_SEQUENCE
```

4.4.3 LOOP ... END LOOP

This statement structure is used to repeat several times a part of a sequence.

All functions between LOOP and END_LOOP keywords are executed repeatedly until the condition which stops loop execution becomes true.

SYNTAX

LOOP

...

END_LOOP <boolean expression>;

<boolean expression> : an expression composed of Boolean objects, Boolean operators, Boolean inquire functions and comparison expressions.

EXAMPLE

The statements between **LOOP** and **END_LOOP** are executed repeatedly until StopLoop become true.

```
SEQUENCE SEQ_ExampleLoop
   ...
   LOOP
      CFV_OutputSquare <- set ; // processing
      WAIT_TIME 1000 ;
      CFV_OutputSquare <- reset ;
   END_LOOP StopLoop;
   ...
END_SEQUENCE
```

4.4.4 XEQ_SEQUENCE

The **XEQ_SEQUENCE** statement is used to abort the sequence in which the statement is located and to start an alternate sequence specified in the statement. The alternate sequence must reside within the same task.

SYNTAX

XEQ_SEQUENCE <sequence identifier>;

<sequence identifier> : the name of an alternate sequence in the same task.

EXAMPLE

If StopInput is true, the sequence SeqOne is stopped and the sequence SeqTwo is started Otherwise, (StopInput is false) execution of SeqOne continues.

```
TASK TSK_XeqSeqExample ;
   SPECS
   ...
   END_SPECS
   SEQUENCE SEQ_One ;
      ...
      IF SBI_StopInput THEN
      XEQ_SEQUENCE SEQ_Two
      END_IF
      statement_1_1 ;           // processing 1
      ...
      statement_1_n ;
   END_SEQUENCE
   SEQUENCE SEQ_Two ;
      statement_2_1 ;           // processing 2
      ...
      statement_2_n ;
   END_SEQUENCE
END_TASK
```

4.4.5 XEQ_TASK

The **XEQ_TASK** statement is used to start another sequence located in a different task. Three dispositions of the sequence in which the **XEQ_TASK** is placed are possible:

- The sequence may continue its execution simultaneously with the started sequence. That's the **default** behaviour.
- The sequence may wait in suspended state until the started sequence has finished execution, then continue its execution (WAIT mode).
- The sequence may be aborted (ABORT mode).

SYNTAX

XEQ_TASK <task identifier> **SEQUENCE** <sequence identifier> [**WAIT**¦**ABORT**];

<sequence identifier> : name of the sequence to be started.

<task identifier> : name of a task which contains the sequence to be started.

User must avoid situation where **XEQ_TASK** statements attempt to start a second sequence in a given task while another sequence is alive. Since task rules allow only one sequence to be alive within a task at a time, the task executing the second **XEQ_TASK** will be stopped with an error (see paragraphs 4.4.5.1 and 4.4.5.2.

EXAMPLE

In this example, processing 1 and processing 2 are executed simultaneously.

```
TASK TSK_XeqTaskExample ;
   SPECS
   ...
   END_SPECS
   SEQUENCE SEQ_NormalProcessing ;
      ...
      XEQ_TASK TSK_Exception SEQUENCE SEQ_StartedByXeqSeq ;
      statement_1_1 ;           // processing 1
      ...
      statement_1_n ;
   END_SEQUENCE
END_TASK

TASK TSK_Exception ;
   SPECS
   ...
   END_SPECS
   SEQUENCE SEQ_StartedByXeqSeq ;
      statement_2_1 ;           // processing 2
      ...
      statement_2_n ;
   END_SEQUENCE
```

```
END_TASK
```

4.4.5.1 CORRECT USAGE OF XEQ_TASK...

The following example shows a correct use of **XEQ_TASK** (see Figure 4-1). Note that for each task, containing a **XEQ_TASK** statement, there is separate task where the sequence which is started with **XEQ_TASK** is located.

```
TASK TSK_1;
   SPECS
   ...
   END_SPECS
   SEQUENCE SEQ_1;
      XEQ_TASK TSK_XEQ_1 SEQUENCE SEQ_a;
      statement_1;
      ...
      statement_n;
   END_SEQUENCE
END_TASK

TASK TSK_2;
   SPECS
   ...
   END_SPECS
   SEQUENCE SEQ_1;
      XEQ_TASK TSK_XEQ_2 SEQUENCE SEQ_b WAIT;
      statement_1;
      ...
      statement_n;
   END_SEQUENCE
END_TASK

TASK TSK_XEQ_1;
   SPECS
   ...
   END_SPECS
   SEQUENCE SEQ_a;
      exception_statements;
   END_SEQUENCE
END_TASK

TASK TSK_XEQ_2;
   SPECS
   ...
   END_SPECS
   SEQUENCE SEQ_b;
      exception_statements;
   END_SEQUENCE
END_TASK
```

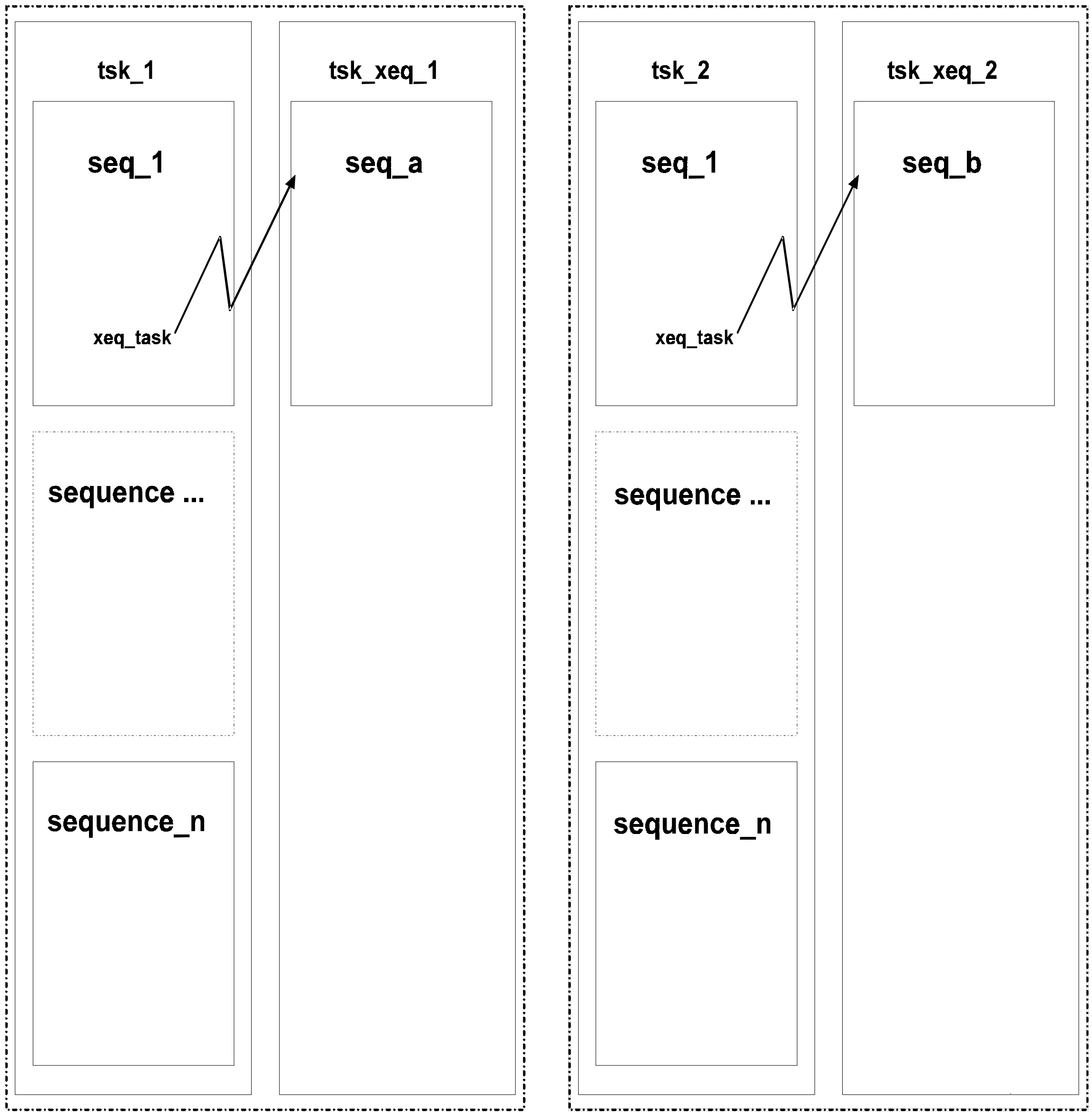


Figure 4-1 Proper use of XEQ_TASK statements

4.4.5.2 INCORRECT USAGE OF XEQ_TASK

The following example shows incorrect use **XEQ_TASK** (see Figure 4-2). The sequences started with **XEQ_TASK** statements are in the same task. If the **XEQ_TASK** of seq_1 in tsk_2 executes "XEQ_TASK seq_b" while seq_a (in the same task) is running, seq_b **can not be run** and tsk_2 will stop on error.

```
TASK TSK_1;
   SPECS
   ...
   END_SPECS
   SEQUENCE SEQ_1;
      XEQ_TASK TSK_xeq SEQUENCE SEQ_a;
      statement_1;
      ...
      statement_n;
   END_SEQUENCE
END_TASK


TASK TSK_2;
   SPECS
   ...
   END_SPECS
   SEQUENCE SEQ_1;
      XEQ_TASK TSK_xeq SEQUENCE SEQ_b;
      statement_1;
      ...
      statement_n;
   END_SEQUENCE
END_TASK


TASK TSK_xeq;
   SPECS
   ...
   END_SPECS
   SEQUENCE SEQ_a;
      statement_1;
      ...
      statement_n;
   END_SEQUENCE
   SEQUENCE SEQ_b;
      statement_1;
      ...
      statement_n;
   END_SEQUENCE
END_TASK
```

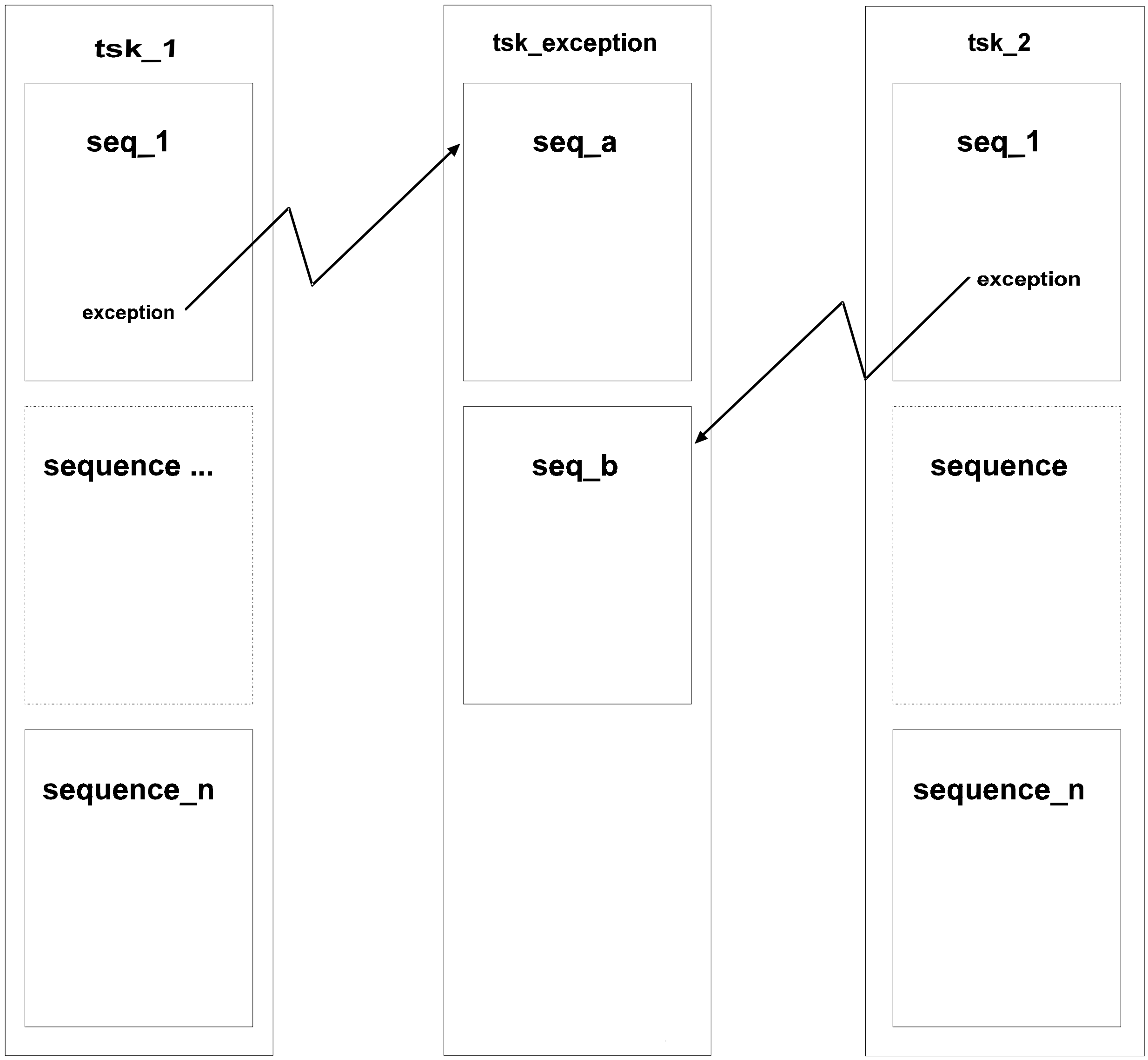


Figure 4-2 Incorrect use of XEQ_Task statement

4.5 TRANSITION CONDITION STATEMENTS

Transition condition statements are a type of service statements which are used to wait until an event occurs. There is four types of transition condition which offer the following control possibilities:

CONDITION	wait for a single Boolean event
CASE	wait for one of several Boolean events
WAIT_TIME	wait a specified time
CONDITION DUPL_START	wait and limit maximum number of parallel sequences.

4.5.1 CONDITION STATEMENT

The **CONDITION** statement is used in a sequence to wait for an event to occur. It is always associated with a transition condition defined by a Boolean expression, and can be also associated with an optional time-out which specifies the maximum time allowed waiting for the transition condition to become true.

The transition condition is evaluated upon **CONDITION** statement execution, If the transition condition is true, execution of the sequence continues without delay. If the transition condition is false, the sequence is suspended until the transition condition becomes true or **TIMEOUT** occurs, where-upon sequence execution resumes. The application can determine if time-out occurred by testing the pre-defined status indicator, TIMEOUT.

SYNTAX

CONDITION <boolean expression> [**TIMEOUT** <time expression>];

<boolean expression> : expression composed of Boolean objects, Boolean operators, Boolean inquire functions and comparison expressions.

<time expression> : expression composed of constants and variables which specify the time-out duration in milliseconds.

A sequence will wait forever and remain indefinitely in the suspended state if its transition condition declared without time-out never occurs.

EXAMPLES

This sequence waits indefinitely until the condition WaitForStart becomes true.

```
SEQUENCE SEQ_ConditionExample1 ;
   SBO_MachineStart <- reset ;
   CONDITION BOL_WaitForStart ;
   SBO_MachineStart <- set ;
   SBO_LedMachineRun <- set ;
END_SEQUENCE
```

This sequence is suspended until the ready condition becomes true or the time-out of 5000 msec is reached, then execution of the sequence resumes with an **IF** statement which tests if the time-out occurred.

```
SEQUENCE SEQ_ConditionExample2 ;
   ZEP_MyAxisToZero <- start ;
   CONDITION ZEP_MyAxisToZero ? ready TIMEOUT 5000 ;
   IF TIMEOUT THEN
      ZEP_MyAxisToZero <- stop ;
      IFV_MachineInError <- set ;
   ELSE
      IFV_MachineReady <- set ;
   END_IF
END_SEQUENCE
```

4.5.2 CONDITION DUPL_START

A system composed of multiple nodes or groups of nodes, with static or dynamic configuration, may have multiple tasks containing identical sequences.

By default, it is feasible that all instances of a sequence (within a multiple task structure) may be executing simultaneously. Sometimes it is necessary to limit the number of identical sequences which can be executed simultaneously (due to power limitation, for example). The number of simultaneous instances allowed is specified by the **DUPL** parameter in the task declaration, and this limitation is imposed using the **CONDITION DUPL_START** statement anywhere in a sequence. Execution of a sequence beyond the **CONDITION DUPL_START** statement is permitted whenever the number of identical alive sequences is less than **DUPL**. If the number of currently alive identical sequences is equal to (or greater than) **DUPL**, the subject sequence is suspended at the **CONDITION DUPL_START** statement until the transition condition (i.e. number of currently alive identical sequences is < **DUPL**) becomes true.

SYNTAX

CONDITION DUPL_START;

EXAMPLE

When the sequence HowToUseDuplStart is started, and until the **CONDITION DUPL_START** statement is reached, all sequence instances (20 in this example) can run together. Beyond the **CONDITION DUPL_START** statement, only 10 instances can run together. However, the **EXCEPTION** statement (prior to the **CONDITION DUPL START)** remains active for all instances, allowing the subject sequence to generate an exception even if it is suspended waiting for DUPL_START become true.

```
TASK TSK_DuplExample ;
  SPECS
    NUMBER = 20 ;
    DUPL   = 10 ;
    CYCLES = 10 ;
  END_SPECS
...
  SEQUENCE SEQ_HowToUseDuplStart[i] ;
    EXCEPTION IFV_SequenceStop[i] ABORT_SEQUENCE ;
    CONDITION DUPL_START ;
    ...
  END_SEQUENCE
END_TASK
```

4.5.3 WAIT_TIME

The **WAIT_TIME** statement is used to force a sequence into the suspended state for a period of time specified by the parameter. After the specified wait interval expires, the sequence returns to the active state and its execution continues.

SYNTAX

WAIT_TIME <time expression>;

<time expression> : an expression composed of constants and variables which specify the waiting time expressed in millisecond.

EXAMPLE

At the **WAIT_TIME** 100 statement, the sequence waits in the suspended state for 100 milliseconds.

```
SEQUENCE SEQ_WaitTimeExample ;
   ...
   WAIT_TIME 100 ;
   ...
END_SEQUENCE
```

4.5.4 CASE

Selection and activation of one sequence among several sequences as a function of conditions if sometimes necessary. This is possible using the **CASE** statement which allows the user to specify a list of sequences with their corresponding transition conditions.

When only one condition in the **CASE** statement is true, the corresponding sequence is activated. If several conditions become true at the same time, the first true condition in the list will be recognised and activate the corresponding sequence. The other conditions will have no effect.

When a **CONDITION** within a **CASE** statement is activated, the initial calling sequence containing the **CASE** statement is aborted.

To avoid an infinite waiting time in the situation where no condition is true, an optional **TIMEOUT** statement may be included to activate a corresponding sequence if the specified wait time is reached.

SYNTAX

CASE

CONDITION <boolean expression 1> **XEQ_SEQUENCE** <sequence identifier 1>;

[**CONDITION** <boolean expression 2> **XEQ_SEQUENCE** <sequence identifier 2>;

...

CONDITION <boolean expression n> **XEQ_SEQUENCE** <sequence identifier n>;]

[**TIMEOUT** <time expression> **XEQ_SEQUENCE** <sequence identifier timer>];

END_CASE

<boolean expression> : an expression composed of Boolean objects, Boolean operators, Boolean inquire functions and comparison expressions.

<sequence identifier> : name of a sequence.

<time expression> : an expression composed of constants and variables which specify the time-out duration expressed in millisecond.

EXAMPLE

depending on which event occurs first, SEQ_1, SEQ_2 or SEQ_3 will be executed. If neither Event1, Event2 nor Event3 occurs within 500 milliseconds after execution of the **CASE STATEMENT,** SEQ_Timeout is executed. The Dummy Statement will never be executed.

```
SEQUENCE SEQ_CaseExample ;
   ...
   CASE
      CONDITION IFV_Event1 XEQ_SEQUENCE SEQ_1 ;
      CONDITION IFV_Event2 XEQ_SEQUENCE SEQ_2 ;
      CONDITION IFV_Event3 XEQ_SEQUENCE SEQ_3 ;
      TIMEOUT 500 XEQ_SEQUENCE SEQ_Timeout ;
   END_CASE
   DummyStatement ;  // this statment will never be reached !
END_SEQUENCE
```

```
SEQUENCE SEQ_1;
   ...
END_SEQUENCE
SEQUENCE SEQ_2;
   ...
END_SEQUENCE
SEQUENCE SEQ_3;
   ...
END_SEQUENCE
SEQUENCE SEQ_Timeout;
   ...
END_SEQUENCE
```

4.6 EXCEPTION STATEMENTS

Normal machine behaviour, which is totally deterministic, is generally described by a set of related sequences. The **EXCEPTION** statement provides a convenient means for describing system behaviour when abnormal events occur and can initiate special processing at any time in a sequence. An Exception event can be any Boolean expression. Each form of the **EXCEPTION** statement has an optional **TIMEOUT** parameter (see paragraph 4.6.5) which, if specified, functions as an additional exception event when the maximum time allowed for execution of a sequence is exceeded. Exception statements are a type of service statement. There are several processing alternatives which include the following:

EXCEPTION ... SEQUENCE	stop execution of the current sequence and execute another sequence in the same task.
EXCEPTION ... ENTRY	jump to an entry point anywhere in the same task.
EXCEPTION ... XEQ_TASK	start a sequence in another task and abort, suspend or continue the current sequence.
EXCEPTION ... ABORT_SEQUENCE	abort the current sequence.

The following rules apply to execution of **EXCEPTION** statements:

- An exception statement can be placed anywhere in a sequence. Once the exception statement is executed, processing of the exception is started as soon as the exception condition (Boolean expression) becomes true,
- An exception is active only while the sequence in which it resides is alive.
- An Exception may be cancelled by the **REMOVE_EXCEPTION** statement.
- Only one exception can be ON at the same time in a sequence. If a new exception is defined, it replaces the current exception and only the new exception has effect.
- If the exception event is already true when the exception statement is executed, the exception is immediately implemented.
- In the situation where a condition event and an exception event within the same sequence occur simultaneously, only the exception will be executed.
- When an exception is executed, the sequence is aborted first (if specified), then the exception is executed.
- The exception treatment acts only on alive tasks.
- If, following execution of an exception statement, the sequence is suspended, and the exception event occurs while the sequence is suspended, the event will be recognized and remembered (even if the exception event subsequently becomes not true). The exception will be executed when the sequence becomes active again.
- If an exception event becomes true while the sequence is active (during execution of a statement) PAM completes executing the current statement, then executes the exception.

4.6.1 EXCEPTION ... SEQUENCE

The Exception with sequence form is used when it is necessary that the 'normal sequence' runs from start to finish if no exception occurs; but if the exception does occur, the 'normal sequence' is aborted and the 'exception sequence' is started. Processing of the 'normal sequence' and the 'exception sequence' are completely separate. The optional **TIMEOUT** ,if specified, becomes a second exception event (see paragraph 4.6.5).

SYNTAX

EXCEPTION <boolean expression> [**TIMEOUT** [<time expression>]] **SEQUENCE** <sequence identifier>;

<boolean expression> : an expression composed of Boolean objects, Boolean operators, Boolean inquire functions and comparison expressions.

<time expression> : an expression composed of constants and variables which specify the time-out duration expressed in milliseconds

<sequence identifier> : name of a sequence.

EXAMPLE

If the exception does not occur, the sequence runs from statement_1_1 to statement_1_n, but if the exception occurs anytime during its execution, the sequence NormalProcessing is aborted and the sequence ExceptionProcessing is started. Similarly, if SEQ_NormalProcessing is alive for more than 1000 milliseconds, the exception occurs due to timeout.

```
TASK TSK_ExceptionSequenceExample ;
   SPECS
   ...
   END_SPECS
   SEQUENCE SEQ_NormalProcessing ;
    EXCEPTION SBI_Stop TIMEOUT 1000 SEQUENCE SEQ_ExceptionProcessing ;
      statement_1_1 ;
      ...
      statement_1_n ;
   END_SEQUENCE
   SEQUENCE SEQ_ExceptionProcessing;
      statement_2_1;
      ...
      statement_2_n;
   END_SEQUENCE
END_TASK
```

4.6.2 EXCEPTION ... ENTRY

The exception with entry is used when it is necessary that the sequence runs from beginning to end, even through the 'exception part', if the exception does not occur.

If the exception occurs, the sequence is aborted and restarted at the **EXCEPTION_ENTRY** statement (see paragraph 4.6.6) named in the **ENTRY** parameter to execute the 'exception part'. The optional **TIMEOUT** (see paragraph 4.6.5) functions as a second exception event.

SYNTAX:

EXCEPTION <boolean expression> [**TIMEOUT** [<time expression>]] **ENTRY** <entry identifier>;

<boolean expression> : an expression composed of Boolean objects, Boolean operators, Boolean inquire functions and comparison expressions.

<time expression> : an expression composed of constants and variables which specify the time-out in milliseconds.

<entry identifier> : name of a label. When the exception occurs the sequence is stopped and restart at the entry identifier (label).

If an exception occurs, the sequence is aborted and restarted at the **EXCEPTION_ENTRY** statement even if sequence execution is past the **EXCEPTION_ENTRY** statement.

EXAMPLE

If no exception occurs, the sequence runs from statement_1 to statement_n through statement_x and statement_x+1. If an exception occurs, the sequence is aborted and restarted at EXCEPTION_ENTRY exit statement, even if sequence execution is between statement_x+1 and statement_n.

If it is necessary that the sequence not be interrupted between statement_x+1 and statement_n, a REMOVE_EXCEPTION statement must be inserted before the exception entry point.

```
SEQUENCE SEQ_ExceptionEntryExample ;
   EXCEPTION SBI_Stop ENTRY exit;
      statement_1;
      ...
      statement_x;
   EXCEPTION_ENTRY exit;
      statement_x+1;
      ...
      statement_n;
END_SEQUENCE
```

If statements_x+1 through statement_n are not to be executed in normal operation, the exception with entry label structure cannot be used. It is necessary to implement two sequences and use the EXCEPTION...SEQUENCE structure.

4.6.3 EXCEPTION ...XEQ_TASK

The exception with task form is used when it is necessary that the 'normal sequence' runs from beginning to end when no exception occurs, and if an exception occurs the 'exception sequence' is started. The 'normal sequence' can continue to run simultaneously with the 'exception sequence' (by default), can wait until the 'exception sequence' is completed (WAIT mode) or can be aborted (ABORT mode). The optional **TIMEOUT** (see paragraph 4.6.5) functions as a second exception condition.

SYNTAX

EXCEPTION <boolean expression> [**TIMEOUT** [<time expression>]]

XEQ_TASK <task identifier> **SEQUENCE** <sequence identifier> [**WAIT**¦**ABORT**];

<boolean expression> : an expression composed of Boolean objects, Boolean operators, Boolean inquire functions and comparison expressions.

<time expression> : an expression composed of constants and variables which specify the time-out duration in milliseconds.

<sequence identifier> : name of a sequence.

<task identifier> : name of a task.

User must avoid situation where **Exception ... XEQ_TASK** statements attempt to start a second sequence in a given task while another sequence is alive. Since task rules allow only one sequence to be alive within a task at a time, the task executing the second **XEQ_TASK** will be stopped with an error (see paragraphs 4.6.3.1 and 4.6.3.2.)

EXAMPLE

If no exception occurs, the sequence NormalProcessing runs from statement_1_1 to statement_1_n. If exception SBI_Stop occurs, execution of NormalProcessing is not affected and the sequence SEQ_ExceptionProcessing in task TSK_Exception starts execution simultaneously.

```
TASK TSK_Normal ;
   SPECS
   ...
   END_SPECS
   SEQUENCE SEQ_NormalProcessing ;
      EXCEPTION SBI_Stop TIMEOUT 1000
         XEQ_TASK TSK_Exception SEQUENCE SEQ_ExceptionProcessing ;
      statement_1_1;
      ...
      statement_1_n;
   END_SEQUENCE
END_TASK


TASK TSK_Exception ;
   SEQUENCE SEQ_ExceptionProcessing;
      statement_2_1;
      ...
```

```
      statement_2_n;
   END_SEQUENCE
END_TASK
```

4.6.3.1 Correct Usage of EXCEPTION...XEQ_TASK...

For each task (see Figure 4-3) there is an exception task. The exception for sequence_1 in task_1 can occur at the same time as the exception for the sequence_1 in task_2.

```
TASK TSK_1 ;
   SPECS
   ...
   END_SPECS
   SEQUENCE SEQ_1 ;
      EXCEPTION SBI_Stop XEQ_TASK TSK_Exception1 SEQUENCE SEQ_a ;
      statement_1 ;
      ...
      statement_n ;
   END_SEQUENCE
END_TASK


TASK TSK_2 ;
   SPECS
   ...
   END_SPECS
   SEQUENCE SEQ_1 ;
      EXCEPTION SBI_Stop XEQ_TASK TSK_Exception2 SEQUENCE SEQ_b ;
      statement_1 ;
      ...
      statement_n ;
   END_SEQUENCE
END_TASK


TASK TSK_Exception1 ;
   SPECS
   ...
   END_SPECS
   SEQUENCE SEQ_a ;
      exception_statements ;
   END_SEQUENCE
END_TASK


TASK TSK_Exception2 ;
   SPECS
   ...
   END_SPECS
   SEQUENCE SEQ_b ;
      exception_statements ;
   END_SEQUENCE
END_TASK
```

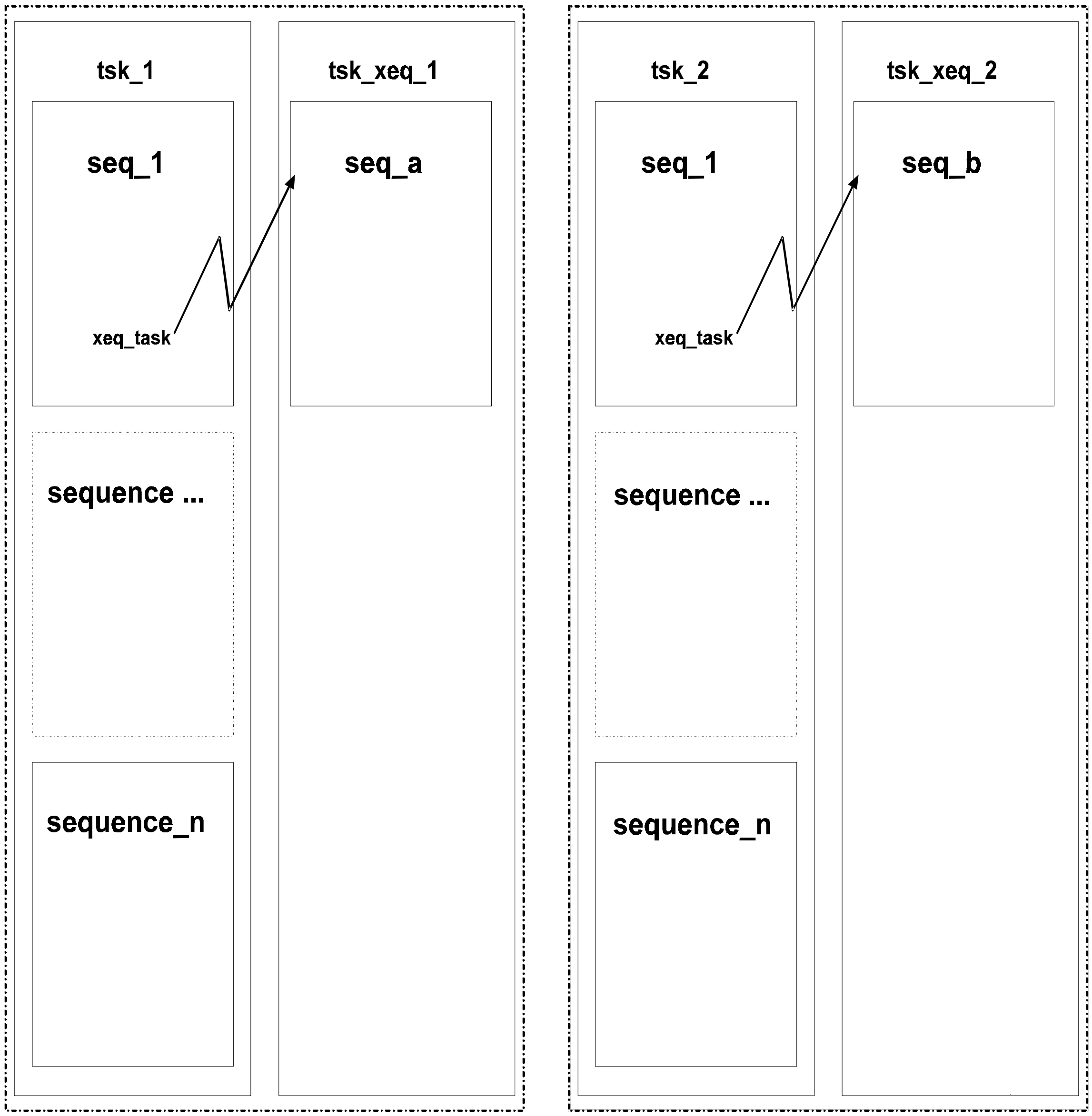


Figure 4-3 Correct usage of Exception ... Xeq_task

4.6.3.2 INCORRECT USAGE OF EXCEPTION...XEQ_TASK...

The exception sequences are in the same task (see Figure 4-4). If the exception for seq_1 in tsk_2 occurs while seq_a is running, sequence_b **can not run** and tsk_2 will be stopped on error.

```
TASK TSK_1 ;
   SPECS
   ...
   END_SPECS
   SEQUENCE SEQ_1 ;
      EXCEPTION SBI_Stop XEQ_TASK TSK_Exception SEQUENCE SEQ_a ;
```

```
      statement_1 ;
      ...
      statement_n ;
   END_SEQUENCE
END_TASK


TASK TSK_2 ;
   SPECS
   ...
   END_SPECS
   SEQUENCE SEQ_1 ;
      EXCEPTION SBI_Stop XEQ_TASK TSK_Exception SEQUENCE SEQ_b ;
      statement_1 ;
      ...
      statement_n ;
   END_SEQUENCE
END_TASK


TASK TSK_Exception ;
   SPECS
   ...
   END_SPECS
   SEQUENCE SEQ_a ;
      statement_1 ;
      ...
      statement_n ;
   END_SEQUENCE
   SEQUENCE SEQ_b ;
      statement_1 ;
      ...
      statement_n ;
   END_SEQUENCE
END_TASK
```

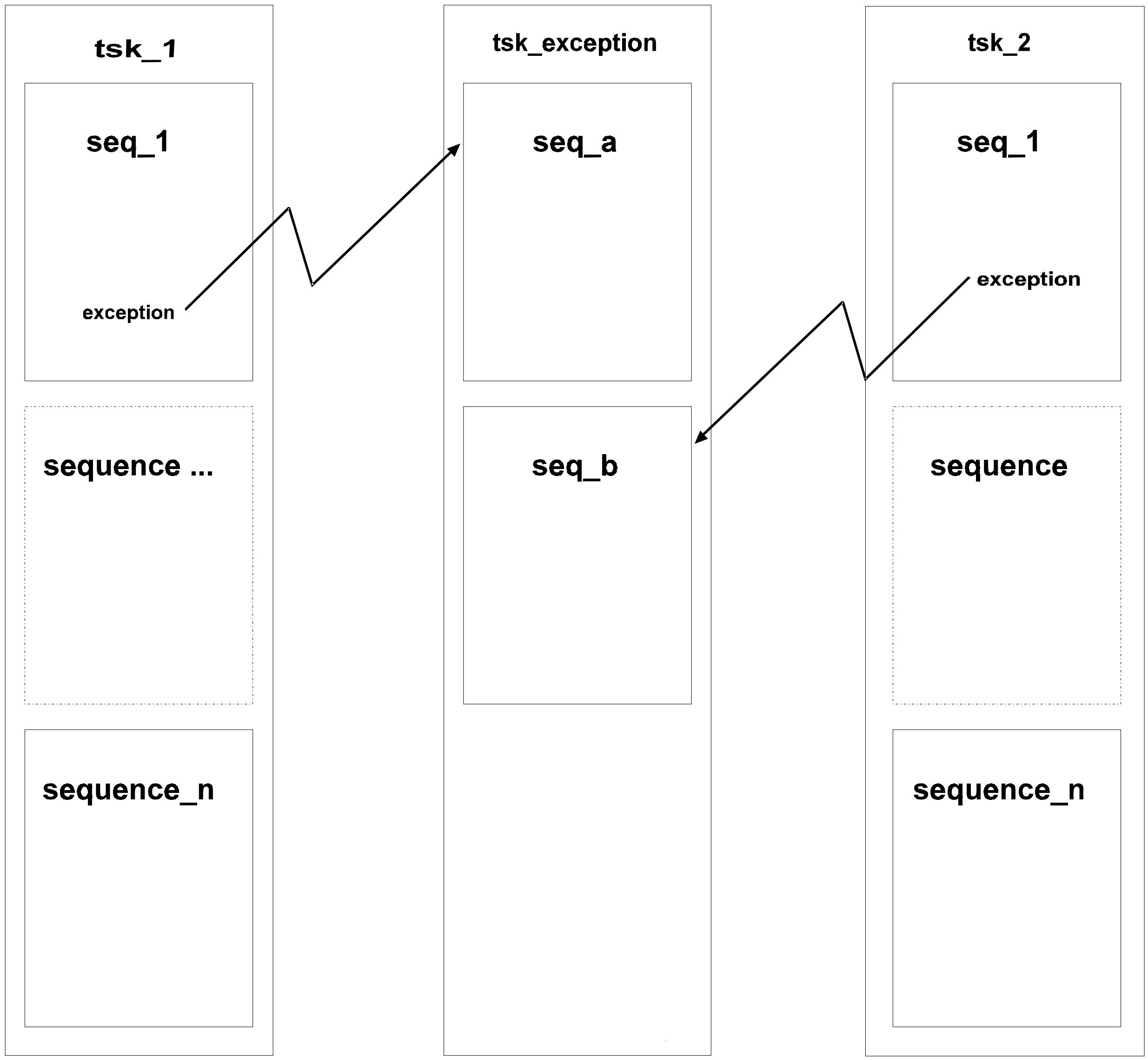


Figure 4-4 Incorrect usage of Exception ... Xeq_Task

4.6.4 EXCEPTION ... ABORT_SEQUENCE

The Exception with abort form is used when there is no 'exception sequence' to perform and it is necessary only to abort the 'normal sequence' when an exception occurs. The optional **TIMEOUT** (see paragraph 4.6.5) serves as a second exception condition.

SYNTAX

EXCEPTION <boolean expression> [**TIMEOUT** [<time expression>]] **ABORT_SEQUENCE;**

<boolean expression> : an expression composed of Boolean objects, Boolean operators, Boolean inquire functions and comparison expressions.

<time expression> : an expression composed of constants and variables which specify the time-out in milliseconds.

EXAMPLE

The sequence runs from statement_1 to statement_n only if no exception occurs. If StopInput exception occurs, sequence execution is aborted.

```
SEQUENCE SEQ_AbortSequenceExample ;
   EXCEPTION SBI_Stop ABORT_SEQUENCE ;
      statement_1;
      ...
      statement_n;
END_SEQUENCE
```

4.6.5 TIMEOUT PARAMETER

Within any form of the **EXCEPTION** statement it is possible to specify a time-out by including an optional **TIMEOUT** parameter in the statement. Time measurement for the optional timeout exception event begins upon execution of the **EXCEPTION** statement. This is a true time measurement which continues even if the sequence is suspended.

SYNTAX

EXCEPTION <boolean expression> [**TIMEOUT** [<time expression>]]...

<boolean expression> : an expression composed of Boolean objects, Boolean operators, Boolean inquire functions and comparison expressions.

<time expression> : an expression composed of constants and variables which specify the time-out in millisecond.

The time-out exception occurs if the <time expression> becomes true. At the beginning of an ‘exception sequence’ it is possible to test if this exception resulted from a timeout using an (**IF TIMEOUT THEN .....END_IF**) statement structure.

When a new **EXCEPTION** is defined in a sequence to replace the current one, the unexpired portion of the timeout value from the superseded **EXCEPTION** is used when **TIMEOUT** is specified without a time value (i.e. if 80% of time is spent, the replacement **TIMEOUT** value is the unused 20%).

EXAMPLE

The following are possible execution sequences for the example program as a function of when and where exception events occur. The maximum time allowed to execute the whole sequence is 1000 ms.

```
   SEQUENCE SEQ_ExceptionExample ;
L1     EXCEPTION SBI_Stop TIMEOUT 1000 ENTRY exit ;
L2     statement_1 ;
L3     EXCEPTION SBI_Emergency TIMEOUT ABORT_SEQUENCE ;
L4     statement_2 ;
L5   EXCEPTION_ENTRY exit ;
L6     IF TIMEOUT THEN
L7        statement_3 ;
L8     ELSE
L9        statement_4 ;
L10    END_IF
   END_SEQUENCE
```

- If no exception occurs the sequence executes statement_1, statement_2 and statement_4.
- If SBI_Stop becomes true before line L3 is executed then statement_4 is the next executed.
- If SBI_Stop becomes true after line L3 is executed, it does not cause an exception.
- If time (1000 ms) is spent before line L3 is executed then statement_3 is the next statement executed.

- If time (1000 ms) is spent after line L3 is executed, the sequence is aborted.
- If SBI_Emergency becomes true before line L3 is executed nothing happens in the sequence.
- If SBI_Emergency becomes true after line L3 is executed, then the sequence is aborted.

4.6.6 EXCEPTION_ENTRY

The **EXCEPTION_ENTRY** statement with <entry identifier> marks the entry point for an **EXCEPTION ... ENTRY** statement (see paragraph 4.6.2) when an exception occurs.

SYNTAX

EXCEPTION_ENTRY <entry identifier>;

<entry identifier> :the name of the label.

EXAMPLE

If no exception occurs, the sequence runs from statement_1 to statement_n through statement_x and statement_x+1.

```
SEQUENCE SEQ_ExceptionEntryExample ;
   EXCEPTION SBI_Stop ENTRY Exit ;
      statement_1;
      ...
      statement_x;
   EXCEPTION_ENTRY Exit;
      statement_x+1;
      ...
      statement_n;
END_SEQUENCE
```

If an exception occurs, the sequence is stopped, even if the sequence is between statement_x+1 and statement_n, and execution is restarted at statement_x+1.

If it is necessary that the sequence not be interrupted between statement_x+1 and statement_n, a **REMOVE_EXCEPTION** statement must be inserted before the exception entry point.

If statements_x+1 through statement_n are not to be executed in normal operation, the exception with entry label structure cannot be used. It is necessary to implement two sequences and use the **EXCEPTION...SEQUENCE** structure.

4.6.7 REMOVE_EXCEPTION STATEMENT

If an exception is defined within a sequence, it is possible to inhibit it's execution using a **REMOVE_EXCEPTION** statement anywhere in the sequence.

SYNTAX

REMOVE_EXCEPTION;

EXAMPLE

In the following example the exception is effective only between statement_1 and statement_x.

```
SEQUENCE SEQ_RemoveExceptionExample ;
   EXCEPTION SBI_Stop ABORT_SEQUENCE ;
      statement_1 ;
      ...
      statement_x ;
   REMOVE_EXCEPTION
      statement_x+1 ;
      ...
      statement_n ;
END_SEQUENCE
```

5 PIPES

5.1 INTRODUCTION

For synchronising axes, PAM provides the capability to define channels, called pipes, between source objects and destination objects. A source object supplies values, and a destination object consumes values. A pipe handles the flow of discrete, programmable, numerical values. The main application of the pipe concept is to describe motion profiles and other positional relationships.

Pipes are built using logical entities called *pipe blocks.* There are three kinds of pipe blocks:

- Input pipe blocks.
- Output pipe blocks.
- Transformer pipe blocks.

The general structure of a pipe is quite simple, being comprised of an input pipe block, followed by zero, one, or more transformer pipe blocks, followed by an output pipe block (see Figure 5-1).

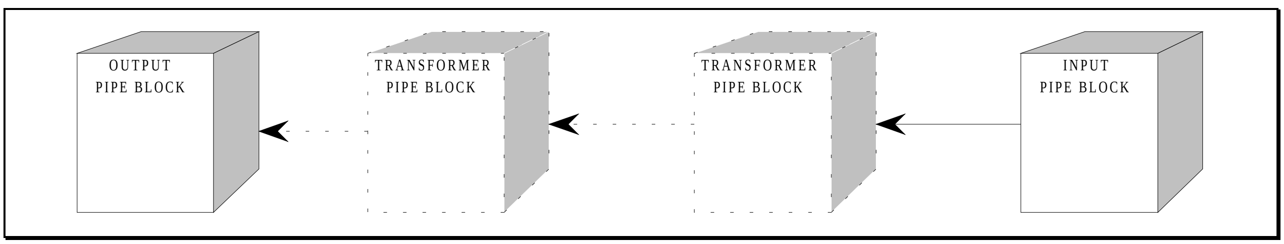


Figure 5-1   General Pipe Structure

Figure 5-2 illustrates a typical pipe structure which includes a sampler pipe block, followed by zero, one or more transformer pipe blocks, followed by a converter pipe block. The sampler gets values from a source object, the converter puts values into a destination object.

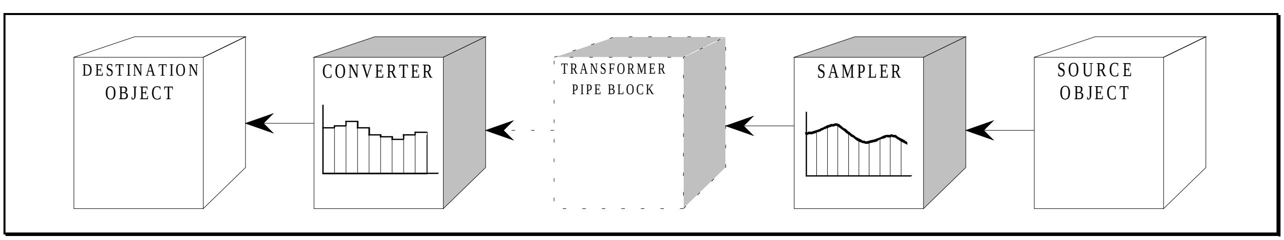


Figure 5-2   Typical Pipe Structure with Source and Destination Objects

In the alternative structure shown in Figure 5-3, the sampler pipe block is replaced by a TMP generator.

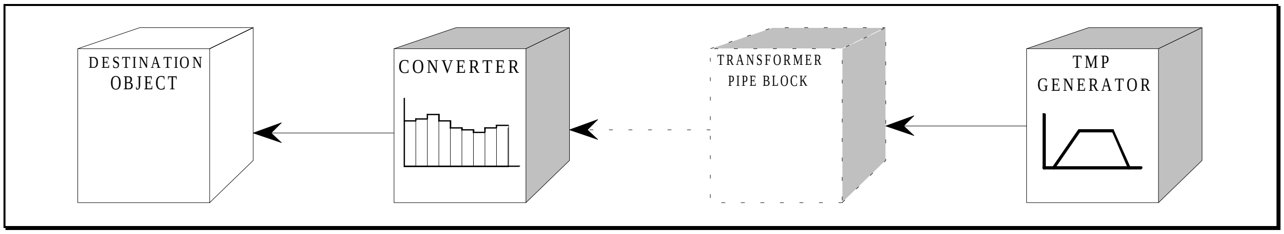


Figure 5-3 Alternative Pipe Structure with TMP Generator

This powerful, modular approach provides a solution for almost any multi-axis requirement. It opens the way to the addition of other functions as user may require. Pipes compute their values periodically. This period is selected using the **PERIOD** parameter in the input pipe block declaration. All pipe values are computed independently of events and sequences execution, as a "protected" task of PAM's processor; thereby assuring they are serviced at the required interval.

Pipes and/or pipe blocks can be installed and removed by statements within sequences; thereby permitting machine behaviour to be adjusted dynamically, depending on what happens on the machine.

5.2 CREATION, ACTIVATION AND DISACTIVATION

5.2.1 CREATION, ACTIVATION STATEMENTS

A pipe is created and activated when statements describing the pipe are processed during the application execution.

STATEMENT SYNTAX

The syntax of the creation-activation statement is directly derived from the graphical representation of a pipe. The general pipe creation statement syntax is as follows:

```
<Output block> { << <Transformer block> }* << <Input block>
```

Figure 5-4 shows a typical pipe and it's associated creation-activation statement. Activation of a pipe is completed when the output pipe block of the pipe is ready.

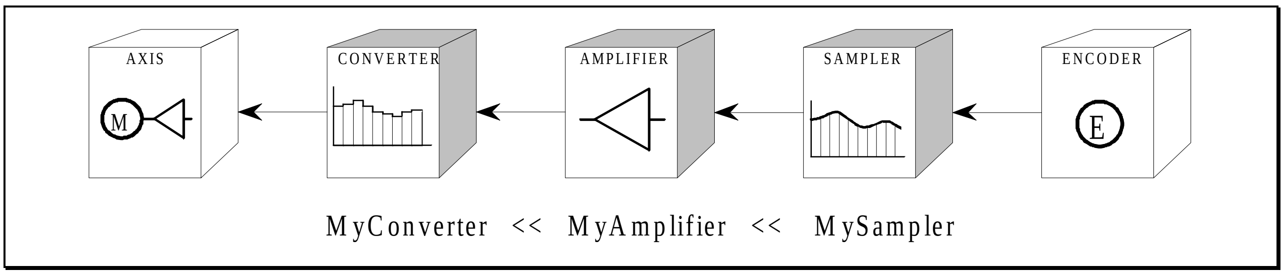


Figure 5-4 Pipe Creation-Activation Statement

EXAMPLE:

```
MyConverter << MyAmplifier << MySampler; // Creation
CONDITION MyConverter ? ready ;          // Wait for activation
...
```

i — The destination object of the output pipe block (MyConverter) is stated in the converter pipe block declaration.

5.2.2 PIPES ACTIVATION CAUTION

⚠ Upon pipe activation, if the first set-point provided by the converter (pipe block) and the current destination axis set-point are not equal, the axis will execute a jump to set-point and the motor controlled by the axis will react by moving toward the new set point at the maximum allowed torque specified in the axis parameters.

To avoid a jump of set-point, the following rules must be applied:

Converter in **POSITION** mode	Converter and axis must have same position set-point at connection time

Converter in **SPEED** mode	Converter and axis must have same speed set-point at connection time but not necessarily same position.
Converter in **TORQUE** mode	Converter and axis must have same torque set-point at connection time but not necessarily same position and speed.

5.2.3 PIPE DISACTIVATION

The de-activation of a pipe is performed when the **disactivate** function is applied to the output pipe block of the pipe.

EXAMPLE

This statement disactivates the pipe shown in Figure 5-4.

```
MyConverter <- disactivate ;                // Disactivate pipe
```

5.3 BUILD UP RULES

When using pipes, it is important to take into account pipe build up rules to avoid compilation errors or unexpected run-time behaviour.

5.3.1 DEFINITIONS

LEADING SUBPIPE

The pipes blocks which are in front of a pipe block (starting from the input pipe block) make up the **leading subpipe** of this pipe block (see Figure 5-5).

SHARED PIPE BLOCK

A pipe block which is present in several pipe's creation-activation statements is called a **shared pipe block** (see Figure 5-5).

SAME DESTINATION

Two pipes have the **same destination** if their output pipe blocks refer to the same destination object (see Figure 5-6).

5.3.2 MUTUAL EXCLUSION RULE

When several pipes have the same destination, only one can be active at a time. Every time a pipe creation-activation statement is encountered during execution, the new pipe is substituted for the current pipe with same destination (if one exists). In other words, destination objects may have only one input.

5.3.3 BLOCK SHARING RULE

If a pipe block is shared by several pipes, the leading subpipe of this pipe block must be identical, except if all the sharing pipes have the same destination. In other words, transformer pipe blocks have only one input and only one output.

A Graphical representation of pipes should be drawn before starting to write the application.

A graphical representation of pipes shows the static (compile time) connections between pipe blocks and pipes of an application, but does not show the dynamic (run time) connections between pipe blocks and pipes.

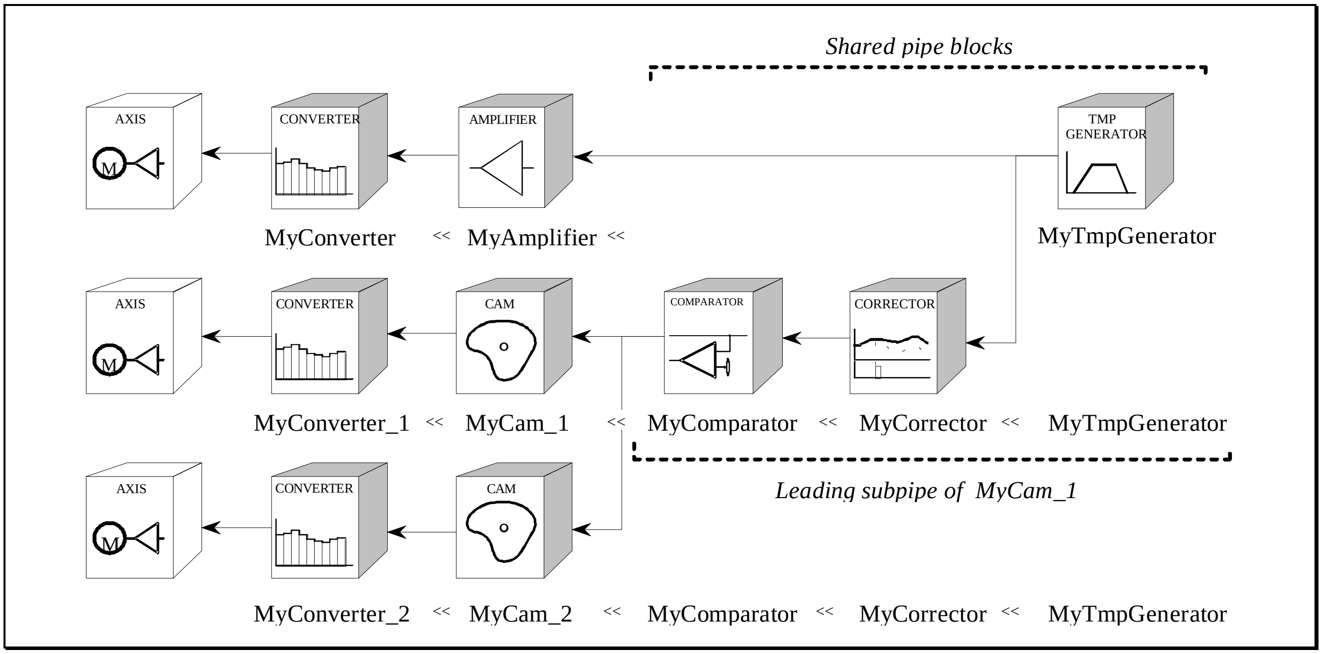


Figure 5-5 Illustration of Pipe Definitions

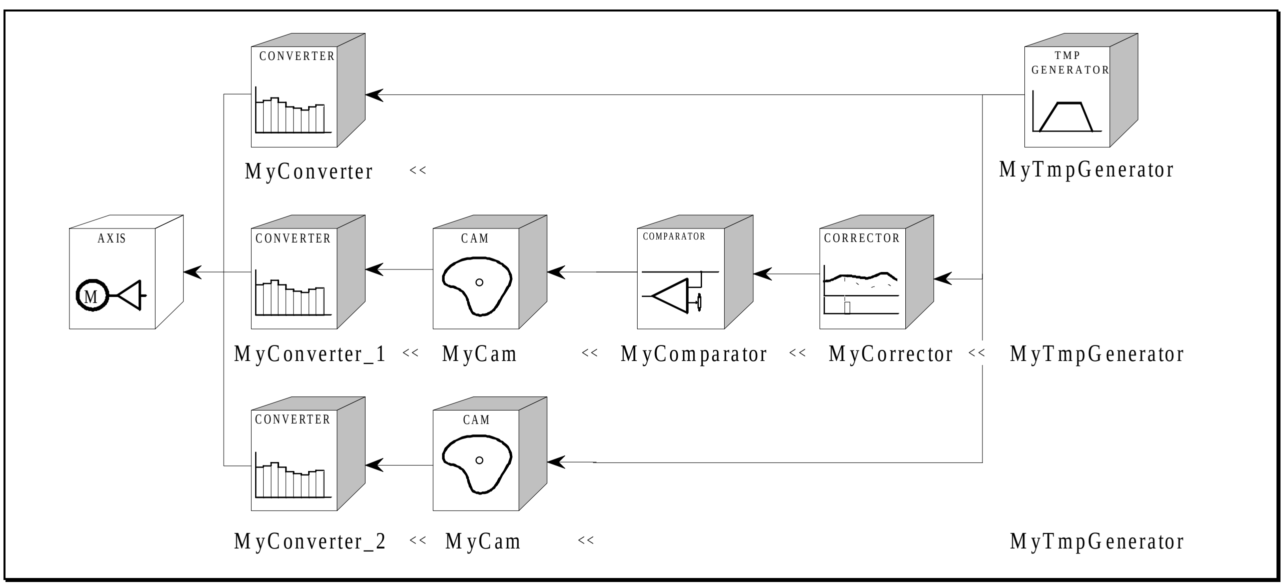
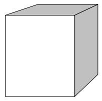


Figure 5-6 Pipes with same Destination and Shared Blocks

5.4 PIPE BLOCKS GENERAL INFORMATION



Pipe blocks are another type of object for use as building blocks in applications. The different types of pipe blocks are listed in Table 5-1.

OUTPUT	TRANSFORMER	INPUT
CONVERTER	AMPLIFIER	SAMPLER
SINK	DERIVATOR	TMP_GENERATOR
	MULTI-COMPARATOR	PMP_GENERATOR
	DISTRIBUTOR	
	PHASER	
	CAM	
	COMPARATOR	
	CORRECTOR	

Table 5-1 Types of Pipe Blocks

5.4.1 LIFETIME

The life of a pipe block begins as soon as its pipe is activated. It ends at the time it is no longer used in any activated pipes. The life begins means that all characteristics are reset to the declaration values and the history of the block begins. Conversely, the life ends means that all internal current values are lost and the block ceases to exist.

As illustrated in Figure 5-5, the same pipe block may be used in several pipe activation statements; however, it can exist (live) only once. For example, MyComparitor and MyCorrector appear in two pipe activation statements, but each of those pipe blocks exists only once.

5.4.2 PERIODICITY AND PHASE OF COMPUTATION

Periodicity (modulus or roll-over point) and phase of computation of each block is determined with the following rules:

- The periodicity in the block declaration is used for pipe source blocks and periodic transformer blocks (Phaser and Filter).
- The periodicity of the preceding block is used in all successive blocks.
- The phase is the same for every block of the same pipe.
- The phase is the same for every pipe sharing at least one block.
- The phase of a pipe is determined mainly by the time it is activated.
- The Phaser pipe block provides a convenient way to modify the phase of a pipe.

5.4.3 PIPE BLOCKS PARAMETERS ACCESS

Pipe blocks parameters are accessed by the application during its execution using parameter access statements (see paragraph 4.3.4).

EXAMPLES:

Read the current **OUTPUT_AMPLITUDE** parameter value of CAM_Example and write the value into CRV_Example:

```
CRV_Example <- CAM_Example:OUTPUT_AMPLITUDE ;
```

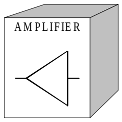
Modify the OUTPUT_AMPLITUDE parameter value of CAM_MyCam:

```
CAM_Example:OUTPUT_AMPLITUDE <- IRV_BasicAmplitude * 3 ;
```

5.4.4 PIPE BLOCK FUNCTIONS

Pipe block functions are a type of statements (see paragraph 4.3.2) used to command, control and monitor pipe block objects. Functions available for each type of pipe block are listed in the pipe block descriptions presented in this chapter. Detailed information on pipe block functions and their use are found in chapters 8 and 10 of this manual.

5.5 AMPLIFIER



PURPOSE

The purpose of the Amplifier block is to amplify or attenuate the flow of values. It can be used as a "gearing ratio" between a virtual master and destination axis. It can also be used as "gearing ratio" between a physical master and a destination axis.

BLOCK INPUT

The amplifier input is the numerical entrance for the flow of values which represent the profile generated by the previous blocks of the pipe. This input is connected to the previous pipe block.

BLOCK OUTPUT

The amplifier output is the numerical exit for the flow of amplified values. This output is connected to the next pipe block.

DECLARATION SYNTAX

AMPLIFIER <identifier> ;

GAIN = <gain value> ;

OFFSET= <offset value> ;

GAIN_SLOPE = <slew rate> ;

OFFSET_SLOPE= <slew rate> ;

END

<identifier> : the name of the amplifier.

<gain value> : (RW), the gain value.

<offset value> : (RW), the ***input*** *offset value.*

<slew rate> : (RW), sets the maximum rate of change at the pipe block output resulting from changes in ***GAIN*** *or* ***OFFSET*** *parameters. If slew rate =* ***MAX.****, the slew rate is infinite. Units are user units per second for* ***OFFSET_SLOPE****, and 1/seconds for* ***GAIN_SLOPE.***

i **GAIN_SLOPE** and **OFFSET_SLOPE** have no effect on amplifier response to dynamics in the flow of amplifier input data.

FUNCTIONS

Inquire Functions

– **value**

Returns the current numerical value coming out of the pipe block.

Example: `CRV_AmpOuput <- AMP_Example ? value ;`

– **ready**

Returns the current value of the ready variable.

5.5.1 PARAMETER MODIFICATIONS

5.5.1.1 GAIN

When **GAIN** is changed, the slew rate at the amplifier output is limited by **GAIN_SLOPE**. However, if the pipe block is not active, the change is immediate regardless of the **GAIN_SLOPE** setting.

If an amplifier pipe block is disactivated while the amplifier output is slewing to a new value following a **GAIN** change, the remaining portion of the output change is made instantly.

The ready flag is false during the interval while the amplifier output is slewing to a new value following a **GAIN** change.

i — Note that the returned value when reading **GAIN** is the instantaneous value which is changing at a rate determined by **GAIN_SLOPE** following modifications to **GAIN**.

5.5.1.2 OFFSET

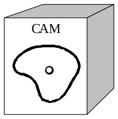
When **OFFSET** is changed, the slew rate at the amplifier output is limited by **OFFSET_SLOPE**. However, if the pipe block is not active, the change is immediate regardless of the **OFFSET_SLOPE** setting.

If an amplifier pipe block is disactivated while the amplifier output is slewing to a new value following an **OFFSET** change, the remaining portion of the output change is made instantly.

The ready flag is false during the interval while the amplifier output is slewing to a new value following an **OFFSET** change.

i — Note that the returned value when reading **OFFSET** is the instantaneous value which is changing at a rate determined by **OFFSET_SLOPE** following modifications to **OFFSET**.

5.6 CAM



PURPOSE

The cam block is used to generate profiles of any shape. The profile generally represents the position evolution of the system. The shape of the profile is represented by a table of numerical values. These values can be generated using software tools such as spreadsheets or specialised cam software.

BLOCK INPUT

The cam input is the numerical entrance for the flow of values which are generated by the previous blocks of the pipe. This input is connected to the previous pipe block. These values are the X[i] values in the cam transfer function.

BLOCK OUTPUT

The cam output is the numerical values exit for the flow of values. These are the Y[i] values of the cam transfer function. This exit is connected to the next pipe object.

5.6.1 DECLARATIONS

Separate cam and profile parameters declarations for the cam pipe block provides the capability to declare and prepare several different cam profiles then apply one of these dynamically to the cam pipe block. Profile switching may be done on the fly, without losing the synchronisation and without dead time.

All cam profile amplitude and offset parameters can be modified dynamically by the application and any parameter modification is immediately taken into account. The way to insure that modifications to several parameters of one profile are applied at the same time is to use an off-line profile and to switch it to the cam after the modification.

In addition, the periodicity of the cam output values can be specified when used with a periodic system.

5.6.1.1 CAM DECLARATION SYNTAX

CAM <cam identifier> ;

PROFILE = <profile name> ;

{ **VALUE_PERIOD** = <period value> |

VALUE_RANGE = <min. value> <max. value>} ;

END

<cam identifier> : name of the cam (string of characters).

<profile name> : (WO), name of the current profile assigned to the cam. It must be a declared profile object.

<period value> : (RW), value of the period of the cam output values expressed in user units, for a cyclic system.

<min. value>, <max. value> : (NA), the value range for linear systems expressed in user units. The max. value must be greater than min. value.

5.6.1.2 PROFILE DECLARATION SYNTAX

PROFILE <profile identifier> ;

FILE = <cam file> ;

INPUT_AMPLITUDE = <input amplitude value> ;

OUTPUT_AMPLITUDE = <output amplitude value> ;

INPUT_OFFSET = <input offset value> ;

OUTPUT_OFFSET = <output offset value> ;

END

<profile identifier> : name of the profile (string of characters).

<cam file> : (NA), name of a PAM cam file without path specification and extension. Only the eight first characters are considered. The cam file must by a file generated by the PAMCAM utility.

<input amplitude value> : (RW), A_{in}*, difference between the last position (where the cam is finished) and the first position (where the cam start) of the previous pipe bloc values expressed in the units of the previous pipe bloc.*

<output amplitude value> : (RW), A_{out}*, difference between the minimum position and the maximum position of the output cam values expressed in the same units as the next pipe bloc.*

<input offset value> : (RW), O_{in}*, position of the previous pipe bloc where the cam has to start expressed in the units of the previous pipe block.*

<output offset value> : (RW), O_{out}*, minimum position of the output cam values.*

INQUIRE FUNCTIONS

– **ready**

This function asks if the cam is ready according to function under execution. In this release, ready is always TRUE.

– **status** (Boolean) (not yet implemented)

This function tests if the cam status is corresponding to the specified status.

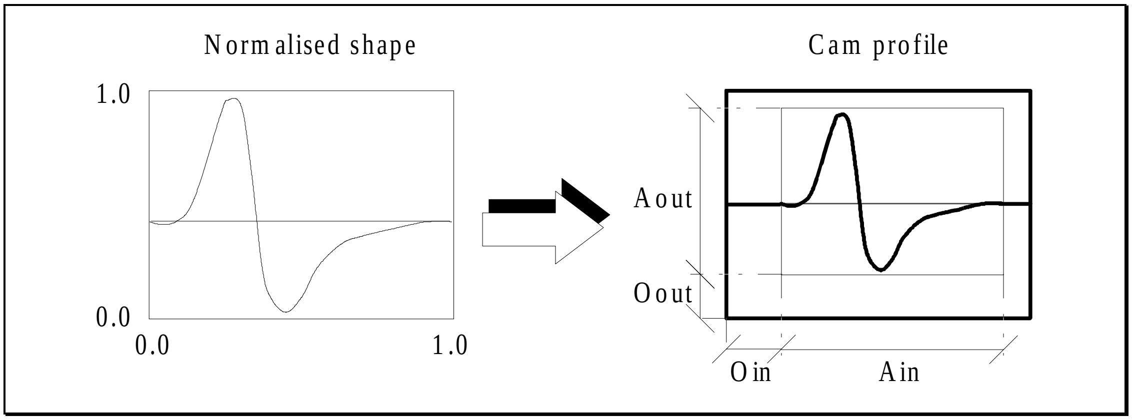
– **status** (not yet implemented)

This function returns the status of the cam.

- value

Return the current numerical value coming out of the pipe block.

Example: `CRV_CamOuput <- CAM_Example ? value ;`



Cam offset and amplitude parameters

5.6.2 CHANGING PROFILE PARAMETERS

If one or several parameters of a profile object are changed when the profile object is used in an active CAM pipe block, the **PROFILE** parameter of the cam must be reinitialised to activate the new profile parameters.

EXAMPLE

The cam profile parameter is re-initialised following changes to profile parameter values.

```
ROUTINES RGR_ProfilesControl ;

   ROUTINE RTN_ApplyProfileNb1 ;
      CAM_Example:PROFILE <- PRO_Nb1 ;   // active profile number1
   END_ROUTINE

   ROUTINE RTN_ModifyProfileNb1 ;
      PRO_Nb1:INPUT_AMPLITUDE <- 200 ;
      PRO_Nb1:INPUT_OFFSET <- 10 ;

      CAM_Example:PROFILE <- PRO_Nb1 ;   // activate new profile
parameters
   END_ROUTINE

END_ROUTINES
```

5.6.3 SHAPE SPECIFICATION

The shape of the cam profile must be processed by the CamMaker utility before it is usable by any PAM application. This utility normalises the shape and all values are bounded between 0.0 and 1.0. Offset and amplitude values given by CamMaker utility have meaning only if the units used to define the shape are the same as the units used in the target application.

The main advantage of normalisation of the shape is that the same shape file can be used any number of times in different cam pipe blocks and applications and scaled for the axis or application in the cam pipe block declaration.

For more information, refer to the CamMaker manual.

> ⚠ The more points used to define the profile shape, the higher their accuracy must be in order to obtain noise-free motion. Attention must be paid to second and third derivatives of the profile which must be very smooth (free of noise).
>
> If motion results are not satisfactory and higher accuracy is not possible, it is better to reduce the number of points in the given shape.

5.6.4 UNITS RULES

The cam pipe block is a non-linear transformer block. This has an important effect on the axis units and periodicity. In general, the output value units of the cam are related to the physical units of the destination axis and the input value units are related to the logical units of the input pipe block (TMP_generator for instance).

In the case where the cam block is not followed (in the pipe) by another cam or an amplifier, the output units of the cam correspond to the physical units of the axis. That means these units, including periodicity, are the units defined in the **AXIS** declaration. If the cam is followed by another cam or an amplifier, these units are only logical intermediate units with no particular meaning.

In the case where the cam block is not preceded in the pipe by another cam or an amplifier, the input value units of the cam correspond to the logical units of the generator or to the physical units of the source object of the pipe. That means these units, including periodicity, are the units used in the generator or defined in the source object. If the cam is preceded by an other cam or an amplifier, these units are only logical intermediate units with no particular meaning.

5.6.5 CAM'S INPUT-OUTPUT TRANSFER FUNCTION

The mathematical relationship of the cam output as a function of the input and the cam parameters is as follows:

$$\text{If} \quad O_{in} \le X_i \le O_{in} + A_{in} \quad \text{then} \quad Y_i = O_{out} + \left(fct\left(\frac{X_i - O_{in}}{A_{in}}\right) * A_{out} \right)$$

Within the stated limits the following functions apply:

$$\text{If} \quad X_i < O_{in} \quad \text{then} \quad Y_i = O_{out} + (fct(0.0) * A_{out})$$

$$\text{If} \quad X_i > O_{in} + A_{in} \quad \text{then} \quad Y_i = O_{out} + (fct(1.0) * A_{out})$$

with:

X_i :	Input value	Y_i :	Output value
O_{in} :	Input offset	O_{out} :	Output offset
A_{in} :	Input amplitude	A_{out} :	Output amplitude
fct :	the function defining the shape		

5.6.6 INTERPOLATION BETWEEN DATA POINTS

Interpolation of third polynomial order is used to compute points between two given points of the profile data. This means that the third derivative of the arc of curve binding two successive points is constant.

The properties of the interpolation method are as follows:

- The curve goes exactly through the given points.

- The acceleration of both segments of the curve at a given point is the same (acceleration continuity condition).

5.6.7 TYPICAL APPLICATIONS

Cams can be used similarly with periodic or no-periodic systems. A position cam for a non-periodic system generally has its ending point equal to its starting point. On the other hand, a cam for a cyclic system, which is running always in the same direction an infinite number of period, has its starting point equal to the first position of the period and its ending point equal to the last point of the period.

Figure 5-7 shows a typical cam shape for a repetitive motion executed on a non-periodic system. Its ending point is equal to its starting point and they must be equal to avoid shifting. When defining the shape table, the last table line, which has a destination position equal to the first one must be present.

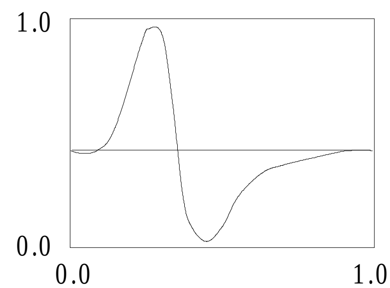


Figure 5-7 Cam Profile for Non-periodic system

Figure 5-8 shows a typical cam shape for a repetitive motion executed on a periodic system. The position is globally continuously progressing and bounded to the position period of the axis. The shape is defined for one period of the system. When defining the shape table, the last table line, which has a destination position equal to the last period position must be present.

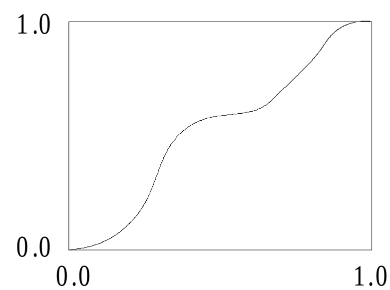
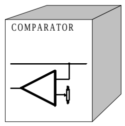


Figure 5-8 Cam Profile for Periodic System

5.7 COMPARATOR



PURPOSE

The purpose of the Comparator pipe block is to generate events when the pipe flow crosses particular values called references. The Comparator block does not modify flow values and it has no effect on the axis and its periodicity.

To cover all applications two working modes are defined:

- normal mode.
- through zero reference mode.

BLOCK INPUT

The Comparator input is the numerical entrance for the flow of values which represent the profile generated by the previous blocks of the pipe. This input is connected to the previous pipe block.

BLOCK OUTPUT

The Comparator output is the numerical exit for the flow of values through the block input. This output is connected to the next pipe block. Since the compactor pipe block does not modify flow values, output always equals input.

DECLARATION SYNTAX

COMPARATOR <block identifier> ;

{ **REFERENCE** = <reference value> ; |

THROUGH_ZERO_REFERENCE = < reference value> ; }

[**ROUTINE** = { <routine statement> **|** **NONE** } ;]

[**REVERSE_ROUTINE** = { <routine statement> } ;]

END

<block identifier> : the name of the Comparator (string of characters).

<reference value> : (RW), the reference value. If the input value of the Comparator is greater or equal than the reference value, the Comparator is ready.

<routine statement> : (WO), any routine statement including the name of a routine object (string of characters) and its parameters (only constants).

SAMPLE DECLARATION

```
COMPARATOR CMP_Leader;
   REFERENCE               = 0.0;
   ROUTINE                 = RTN_LeaderRef;
```

```
   REVERSE_ROUTINE          = NONE ;
END
```

FUNCTIONS

Inquire functions:

– **ready**

This function asks if the Comparator is ready (if the reference value of the Comparator is reached.

– **value**

Return the current numerical value coming out of the pipe block.

5.7.1 COMPARATOR MODES

In normal mode, which applies mainly to bounded motions, the Comparator's ready flag is false as long as the flow value is less than the reference and becomes true as soon as the flow value is greater than or equal to the reference.

The through zero reference mode is used to detect properly a periodic threshold crossing of motions on periodic axis where the flow values are always greater than or equal to zero but lower than the position period. In this mode, the flow values must first cross one period limit and then, as soon a value is greater than or equal to the reference, the ready flag becomes true.

Comparator mode, as well as reference value can be changed while an application is running by modifying the **REFERENCE** or **THROUGH_ZERO_REFERENCE** parameter.

A routine may be connected to the Comparator pipe block through the optional **ROUTINE** parameter. Every time the Comparator's ready flag become true, a routine specified by **ROUTINE** is automatically started, with a very short reaction time. The connected routine can be modified at any time by the application. Similarly, the true to false transition of the ready flag invokes the routine specified in the **REVERSE_ROUTINE** parameter.

5.7.2 EXAMPLES

In the following code portion *led_1* is set as soon as the pipe flow becomes greater than or equal to 1000.

```
CMP_Example:REFERENCE <- 1000.0 ;
CONDITION CMP_Example ? ready;
SBO_Led1 <- set;
```

The following code portion sets *led_1* as soon as the pipe flow becomes less than 500.

```
CMP_Example:REFERENCE <- 500.0 ;
CONDITION !(CMP_Example ? ready) ;
SBO_Led1 <- set;
```

In the following code portion *SBO_Led1* is set when the pipe flow crosses one period (zero crossing) and then as soon as it becomes greater than or equal to 326 (see paragraph 5.7.3).

```
CMP_Example:THROUGH_ZERO_REFERENCE <- 326.0 ;
CONDITION CMP_Example ? ready ;
SBO_Led1 <- set ;
```

The following statement sets Compoutput = current value of the Comparator named “MyComparitor”.

```
CRV_CompOuput <- CMP_Example ? value ;
```

5.7.3 USING THROUGH ZERO REFERENCE MODE

The necessity for using the through zero reference mode was illustrated in a previous example. Assume that the system is a periodic system with a position period of 500. The system is running in the positive direction (pipe flow values increases). Imagine that the position of the system is now 400 and we want to wait for the system to reach 326 again. If we ask for the Comparator to detect the 326 reference in normal mode, it will immediately set the ready flag at true (400 > 326) but this is not what we want. If we ask for the Comparator to detect the 326 value in through zero reference mode, it will wait for the system to cross one zero reference (cross the position value = 0) and then trigger the application on the correct condition.

5.7.4 COMPARATOR RESPONSE TIME CONSIDERATIONS

There is a big difference in response time when using a Boolean equation comparing a value with a reference, verses using a Comparator pipe block do to the same processing. With the Boolean equation, PAM periodically performs the comparison, ignoring any dynamics taking place between successive comparisons, resulting in delays in triggering sequences and possible loss of information when the pipe flow value crosses the reference momentarily between comparisons.

With a Comparator, the value of the ready flag is intrinsically updated each time a new pipe flow value is computed. Therefore, is it impossible to loose any transitions.

5.7.5 CONNECTING A ROUTINE

Two ways are possible to connect a routine to the Comparator. It is possible to declare the routine in the Comparator pipe block declaration. In this case <u>only constants</u> can be used as routine parameters.

It is also possible for the application to initialise or modify the **ROUTINE** parameter dynamically at run time. In this case any <u>expression</u> can be used as routine parameters.

⚠ The routine must have been previously defined in the application.

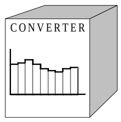
To connect a ROUTINE to the Comparator :

<block identifier>:**ROUTINE** **<-** <routine syntax> ;

EXAMPLE

```
CMP_Example:ROUTINE <- RTN_PainterStart ;
```

5.8 CONVERTER



PURPOSE

The converter block is necessary to define the connection between a pipe and a destination object. Incoming numerical values are converted to POSITION, SPEED or TORQUE set-points depending on Converter mode.. This conversion has no effect on the axis units and its periodicity. This block must be present at the end of a pipe.

BLOCK INPUT

The converter input is the numerical entrance for the flow of values which represents the profile generated by the previous blocks of the pipe. This input is connected to the previous pipe block.

BLOCK OUTPUT

The converter output is the position, speed or torque set-points for the flow of values representing the profile generated by the previous blocks of the pipe. This exit is connected to the destination object of the pipe (AXIS object for instance).

DECLARATION SYNTAX

CONVERTER <identifier> ;

DESTINATION = <destination object> ;

MODE = <converter mode> ;

END

<identifier> : name of the converter.

<destination object> : (NA), destination object of the pipe.

<converter mode> : (NA), one of the following modes, POSITION, SPEED, TORQUE or VALUE.

EXECUTIVE FUNCTIONS

Executive Functions:

– **disactivate**

This function deactivates the pipe ending with the converter.

EXECUTIVE FUNCTIONS FOR AXES SET DESTINATION

For converters with an axes set destination, the following functions are available:

– **connect**

Connects a specified axis of the axes set to the converter.

– **connect_all**

Connects all the axis of the axes set to the converter.

– **disconnect**

Disconnects a specified axis of the axes set from the converter.

– **disconnect_all**

Disconnects all the axis of the axes set from the converter.

– **change_ratio**

Changes the ratio used to control motion of the specified axis of the axes set through the converter (through this converter only).

– **change_all_ratios**

Changes the ratio used to control the motion of all the axis of the axes set through the converter (through this converter only).

INQUIRE FUNCTIONS

– **ready**

This function asks if the converter and all the pipe creation-activation is finished.

5.8.1 CONVERTER'S MODE AND DESTINATION

Theoretically any output object can be used as the destination of a converter in any mode. Practically, some configurations don't have any physical meaning, some others are not implemented. The possibilities are as follows:

	MODE			
DESTINATION	POSITION	SPEED	TORQUE	VALUE
AXES_SET	yes	yes	no	no
AXIS	yes	yes	yes	no
PAM_ANALOG_OUTPUT	no	no	no	yes

5.8.2 DESTINATION OBJECTS BEHAVIOUR

AXIS object, **MODE** = **POSITION**:

The values drive the position of the motor. At pipe activation, the current (axis) position is set to the first value given by the pipe by moving the motor. Speed and acceleration are derivatives of position. The torque is set according to the regulator needs. Units are the axis physical units.

AXIS object, **MODE** = **SPEED**:

The values drive the speed of the motor. At pipe activation, the current position is not affected. Position is the integral of speed, and acceleration is the derivative of speed. The torque is set according to the regulator needs. Units are the axis physical units per second.

AXIS object, **MODE** = **TORQUE**:

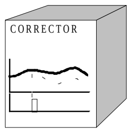
The values drive the limit torque of the motor. At pipe activation, the current position is not affected. The torque is limited if the motor is not at the required position. Position, speed and acceleration are set according to external forces.

Units are always Newton * meter [Nm].

PAM_ANALOG_OUTPUT object, **MODE** = **VALUE**:

The values drive the voltage of a PAM analogue output. Units are the PAM analogue output units.

5.9 CORRECTOR



PURPOSE

The purpose of the corrector block is to dynamically compute and add corrections to the flow of values. A common application of a corrector is in implementing an automatic registration function where corrections are applied to flow of values which represent the position dimension and the correction magnitude is small in comparison to the base position, but the correction magnitude is not limited. The following explanations are focused on position corrections. The corrector block has no effect on the axis units and its periodicity.

BLOCK INPUT

The corrector input is the numerical entrance for the flow of values which represent the profile generated by the previous blocks of the pipe. This input is connected to the previous pipe block.

BLOCK OUTPUT

The corrector output is the numerical exit for the flow of corrected values. This output is connected to the next pipe block.

DECLARATION SYNTAX

CORRECTOR <identifier> **;**

CORRECTION MODE **= <**correction mode> **;**

CORRECTION REFERENCE =<correction reference> ;

CORRECTION_SLOPE = <correction slope value> **;**

CORRECTION_LEVEL = <correction level value> **;**

TRIGGER_MODE = <trigger mode> **;**

TRIGGER_INPUT = <trigger input object> **;**

[VALUE_PERIOD = <period value> |

VALUE_RANGE = <min. value> <max. value>] **;**

DELAY_COMPENSATION = <sensor delay> **;**

END

<identifier> : name of the corrector.

<correction mode> : (NA), for selection of either ***IMMEDIATE*** *or* ***ON_REQUEST*** *mode of corrector operation.*

i | Selecting **CORRECTION_MODE** = **IMMEDIATE** makes corrector operation compatible with previous versions.

<correction reference> : (NA), for selection of either **INPUT** *or* **OUTPUT** *pipe values as source of reference position in correction computation.*

i | Selecting **CORRECTION_REFERENCE** = **INPUT** makes corrector operation compatible with previous versions.

<correction slope value> : (RW), rate of change of correction allowed (see Figure 5-9), given in user unit per square second. It correspond to the acceleration and deceleration of the trapezoidal correction profile generated by the correction generator when working in position mode.

<correction level value> : (RW), maximum level of correction (see Figure 5-9), given in user units per second. It correspond to the travel speed of the trapezoidal correction profile generated by the correction generator when working in position mode.

<trigger mode> : (NA), selection of the mode, ONCE or REPETITIVE.

<trigger input object> : (NA), identifier of a binary input object.

<period value> : (NA), value period for cyclic systems

<min. value>, <max. value> : (NA), value range for linear systems. The max. value must be greater than min. value.

i | If no **VALUE_PERIOD** | **VALUE_RANGE** value is specified, the default is infinite.

<sensor delay> : (RW) , reaction time of the sensor added with the delay introduced by the capacitor of the binary input. The value must be given in seconds.

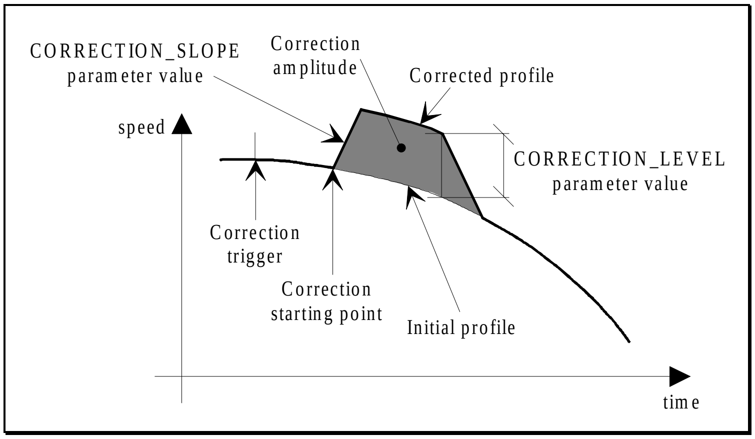


Figure 5-9 Corrector Parameters Impact on Trajectory

CORRECTOR SAMPLE DECLARATION

```
CORRECTOR COR_Detect;
   CORRECTION_SLOPE      = 3600.0 ;
   CORRECTION_LEVEL      = 3600.0 ;
   CORRECTION_REFERENCE = INPUT ;
   CORRECTION_MODE       = ON_REQUEST ;
   TRIGGER_MODE          = ONCE ;
   TRIGGER_INPUT         = SBI_DetectHigh ;
   VALUE_PERIOD          = 360.0 ;
   DELAY_COMPENSATION    = 0.00 ;
END
```

CORRECTOR FUNCTIONS

Executive Functions

– **trigger**

This function sets the "must be" value of the corrector. If the TRIGGER MODE of the corrector is ONCE, the trigger is rearmed.

– **trigger_off**

This function disables the corrector.

– **start correction**

This function commands the corrector to execute a correction (enable correction generator). If no parameter is specified, the corrective value computed by the corrector is used. When a parameter is included, the parameter value is used as corrective value. This function is active only when **CORRECTION_MODE** = **ON REQUEST**.

Examples:

```
COR_Example <- start_correction ;
COR_Example <- start_correction (-180) ;
```

INQUIRE FUNCTIONS

– **ready**

This function asks if the correction is done.

– **triggered**

This function asks if a correction is pending.

– **latched_value**

This function asks the "is" value, latched upon detection of the trigger event.

– **latched_d_value**

This function asks the derivative of the "is" value, latched upon detection of the trigger event.

– **latched_dd_value**

This function asks the second derivative of the "is" value, latched upon detection of the trigger event.

– **value**

Return the current numerical value coming out of the pipe block.

Example: `CRV_CorrOuput <- COR_Example ? value ;`

– **correction**

Returns corrective value (including sign) currently in use by corrector. The returned value must be interpreted differently depending on the setting of **CORRECTION_REFERENCE.** Note that if a corrective value was last specified via a **start_correction** command, this corrective value is returned.

Correction is valid only after the corrector is triggered and before the correction is started.

5.9.1 DETERMINING CORRECTION VALUES

At the moment a trigger input transition is detected (see Figure 5-10) the current position value (either input value or output value in the corrector block diagram) is extracted from pipe flow, delay compensation is applied, and the result ("is" value in as well as its first and second derivatives are latched. The selection of input value/output value is controlled by the **CORRECTION_REFERENCE** parameter.

The difference (corrective value) between the delay-compensated position value (“is” value) and the position reference ("must be" value) supplied as a parameter of the **trigger** function is computed and passed to the correction generator. The correction generator is then ordered to perform a correction of magnitude equal to the corrective value. The correction generator may begin implementing the correction immediately or upon receipt of a **start_correction** function depending on the **CORRECTION_MODE** parameter. If a corrective_value is included in the **start_correction** function, it replaces the corrective value computed by the corrector.

Finally, the corrective value is added to the flow of position values to produce a flow of corrected position values (Output value in. An application sequence can monitor corrector activity using it’s ready and triggered functions to determine status.

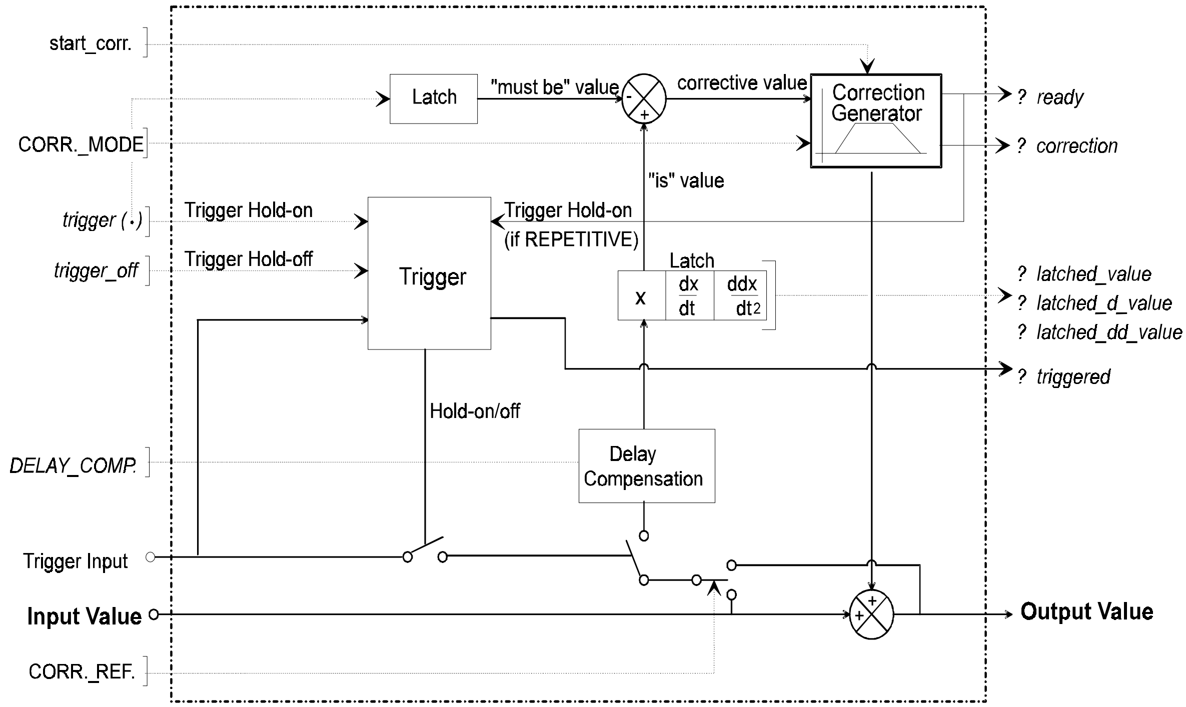


Figure 5-10 Corrector Block Diagram

5.9.2 CORRECTOR STATES DESCRIPTION.

Lets now examine the different states of the corrector in detail. The following discussion makes reference to the Corrector State Transition Diagram shown in Figure 5-11.

The corrector is initialised at pipe installation and is started in "STANDBY" state. At this point, it is not waiting for a trigger event, the position reference value is not yet initialised and no correction is in progress. The flow input values is passing through the corrector pipe block without modification.

Upon occurrence of the **trigger** function, the accompanying “must be” (position reference) value is latched, and the corrector enters the "WAIT EVENT" state awaiting the trigger event (EVENT). In this state, the ready flag is true, and the triggered flag is false (trigger is armed).

When the trigger event is detected, the current position (“is” value) is latched. Logical flow then follows one of two main paths depending on the **CORRECTION_MODE** parameter. If **CORRECTION_MODE** = **IMMEDIATE,** the corrector enters one of the "IN CORRECTION..." states depending on the **TRIGGER_MODE** parameter. When **TRIGGER MODE** = **ONCE**, IN CORRECTION/TRIGGER HOLDOFF state is entered where-upon the triggered status becomes true indicating the corrector has been triggered (and no further corrections may occur until the trigger is re-armed), and the ready status becomes false indicating the correction generator is active (a correction is in progress). When the correction is completed, the corrector returns to STANDBY state where another **trigger** function is required to initiate a new corrector cycle. When **TRIGGER_MODE = REPETITIVE**, IN CORRECTION/TRIGGER HOLDON state is entered. Here, triggered status remains false (indicating the trigger remains armed) and ready status is false

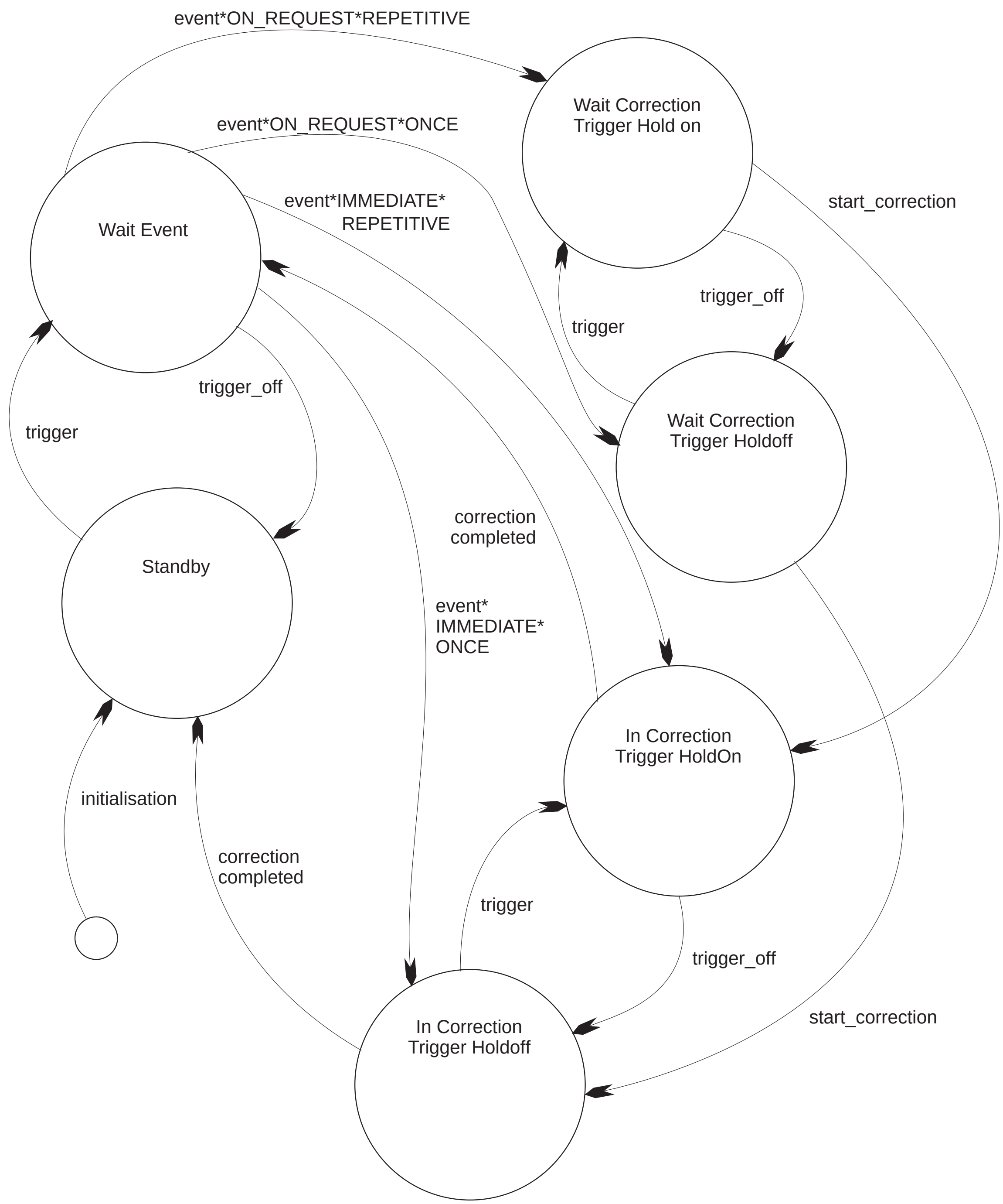


Figure 5-11 Corrector State Transition Diagram

while the correction generator is active**.** Upon completion of the correction, WAIT EVENT state is entered where the next occurrence of the trigger event initiates a new corrector cycle.

When **CORRECTION MODE** = **ON_REQUEST**, the corrector enters one of the WAIT CORRECTION states until occurrence of a **start_correction** command permits transition to the corresponding IN CORRECTION ... state where the function of the corrector is as described in the previous paragraph.

All other state transitions possibilities are shown in the corrector state transition block diagram for completeness.

5.9.3 DELAY COMPENSATION

Reaction time compensation for delays from the mechanical event to the comparison with the reference value are performed automatically by the "delay compensation" module. This “global” delay is composed of three parts:

- reaction time of the sensor including delay introduced by the binary input
- transmission time of the field bus
- reaction time of the corrector pipe block

The sensor delay may be specified by the application using the **DELAY_COMPENSATION** parameter. The transmission delay and the pipe block reaction time are automatically determined by PAM and can not be accessed by the application.

To compensate for reaction time, the "delay compensation" module make a retrospective computation of the actual input position value so that the position, speed and acceleration values latched are axis conditions at the time the mechanical event occurred. This compensation assumes that the acceleration is constant. The latched values are accessible by the application.

5.9.4 WORKING MODES

There are two operating modes for the trigger circuit. The mode is selected at the pipe block declaration level.

In ONCE mode, the corrector performs one correction cycle upon the next transition of the trigger.. In this mode it is necessary to have a sequence monitor pipe status and reactivate the trigger when necessary.

In REPETITIVE mode, which is the mode most commonly used, a correction cycle is initiated upon each transition of the trigger.

5.9.5 TYPICAL USES OF CORRECTORS

A system has slippage between the motor shaft and final mechanical parts but the final mechanical parts must be positioned accurately. If the parts can be detected somewhere in their motion by a sensor, the corrector can compensate for the slippage.

A system has a mechanical drive which is moving material with a non regular distribution. but the material in motion must be accurately positioned. If some reference point on the material in motion can be detected somewhere in it’s motion by a sensor, the corrector can compensate the distribution errors.

5.9.6 NUMERICAL EXAMPLES

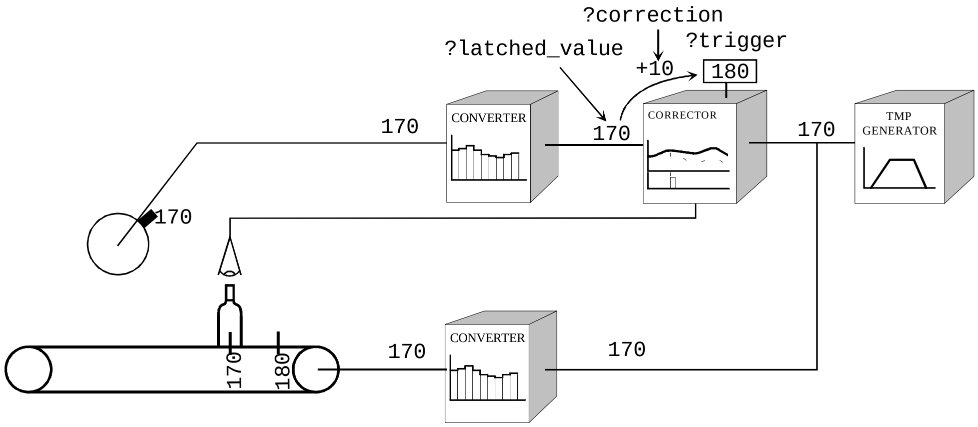
The following examples illustrate implementation of a registration system utilising a corrector where bottles on a conveyor belt must be synchronised with a bottle capping mechanism. Each example illustrates the following three situations:

- instant (t - ε): at bottle detection, immediately before the correction (assume no detection delay);
- instant (t + ε): immediately after the correction (assumes correction is instantaneous);
- instant (T + t): at bottle detection, in the next machine cycle, (bottle has exactly the same misalignment as last cycle so no correction is necessary).

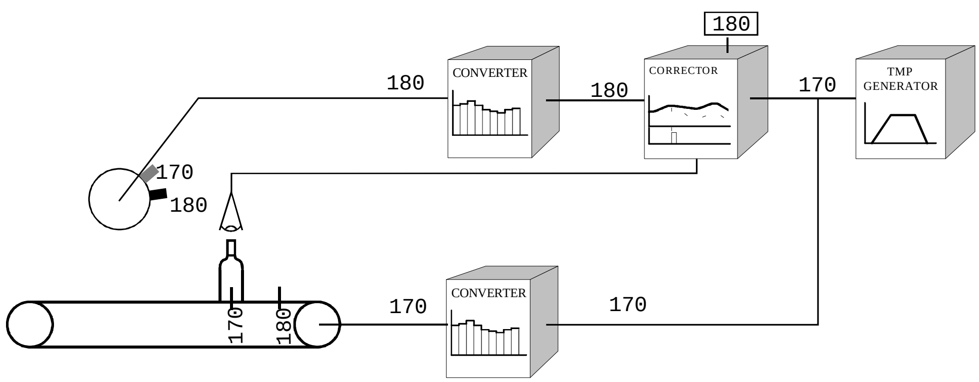
5.9.6.1 OUTPUT REFERENCE

This example (see Figure 5-12) shows (with some simple numerical values) a situation where corrections are applied to the capping mechanism to achieve synchronisation. In this example reference positions are taken from the corrector output. Synchronisation is achieved when the capping mechanism is at position 180 (?**trigger)** when bottle detection occurs.

Before correction



After instantaneous correction



Next cycle

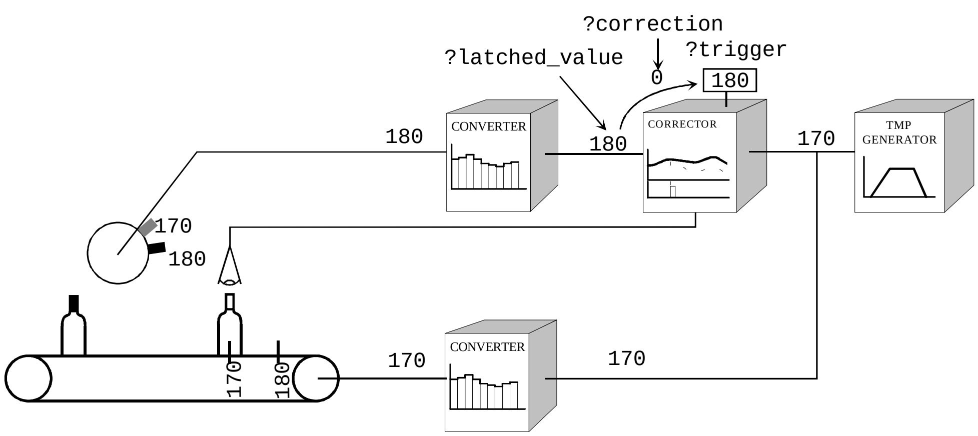


Figure 5-12 Corrector Operation with Output Reference

5.9.7 INPUT REFERENCE

In this example (see Figure 5-13) corrections are applied to the conveyor and reference positions are taken from the corrector input. Once again, synchronisation is achieved when the capping mechanism is at position 180 (**?trigger**) when bottle detection occurs.

5.9.8 ADJUSTING DELAY COMPENSATION TIME

The first step is to initialise the **DELAY_COMPENSATION** parameter with the theoretical reaction time of the sensor.

The second step is to run the system very slowly (speed A) and execute one correction cycle, then measure the positioning error (error A) while the system is stopped.

The third step is to execute one correction cycle running the system at nominal speed (speed B), then measure the positioning error (error B) while the system is stopped.

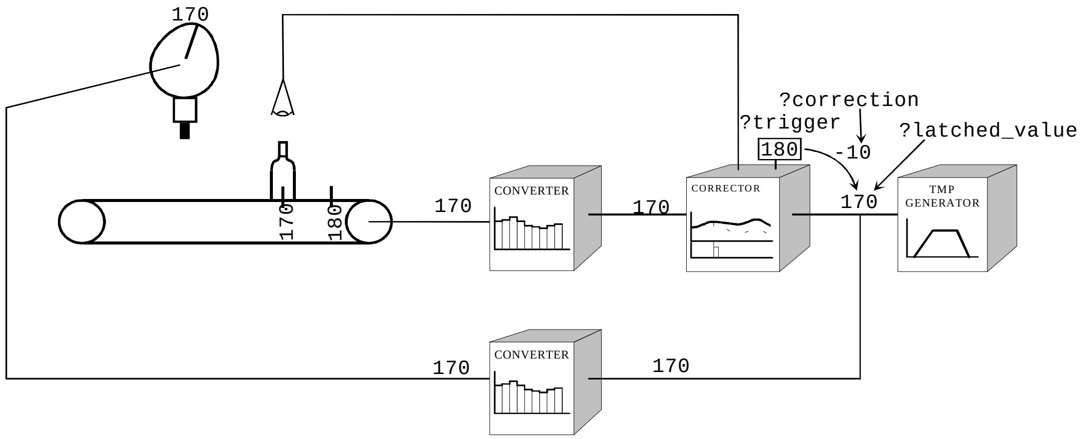
The delay compensation value is obtained using the following equation:

$$sensor\ delay = \frac{error\,B - error\,A}{speed\ B - speed\ A}$$

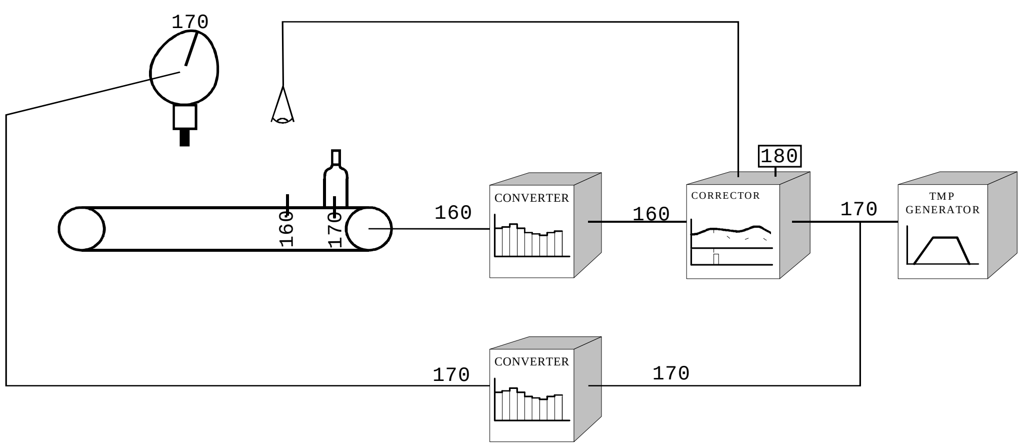
The value of the sensor delay must be expressed in seconds.

If the accuracy is not sufficient, the procedure can be repeated using the result of the first delay compensation in the first step instead of the theoretical reaction time of the sensor.

<u>Before correction</u>



<u>After instantaneous correction</u>



<u>Next cycle</u>

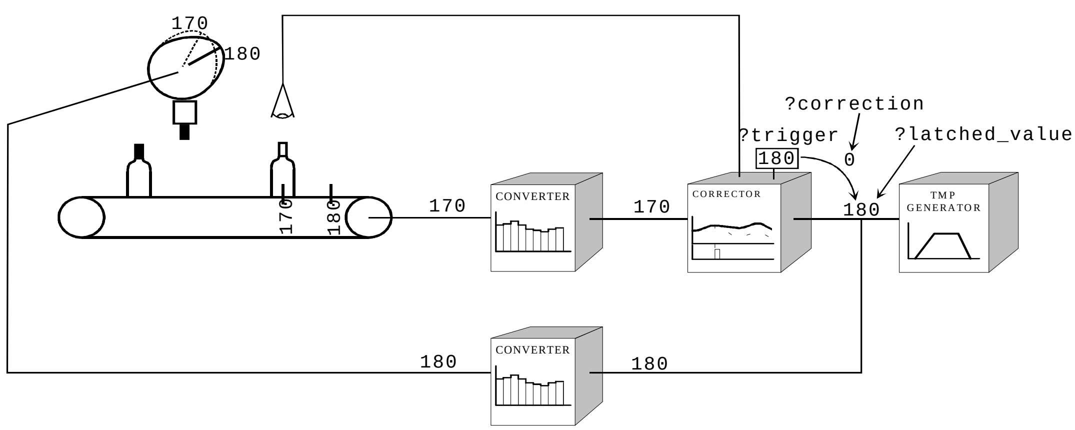
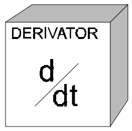


Figure 5-13 Corrector Operation with Input Reference

5.10 DERIVATOR



PURPOSE

The Derivator is a transformer pipe block whose purpose is to calculate the derivative of it's input values with respect to time.

Formally, the transfers function is:

$$y = \frac{dx}{dt}$$ with: x = input values, y = output values

For example, assuming the input value increases each millisecond by one (degree), the output value will be one thousand (degrees per second).

BLOCK INPUT

The Derivator input is the numerical entrance for the flow of values which represents the profile generated by the previous blocks of the pipe. This input is connected to the previous pipe block.

BLOCK OUTPUT

The Derivator output is the numerical exit for the flow of values derived from the block input. This output is connected to the next pipe block.

DECLARATION SYNTAX

DERIVATOR <identifier>;

{ **VALUE_PERIOD** = <period value> |

VALUE_RANGE = <min value> <max_value> };

END

<identifier>: name of the Derivator.

<period value> (RW): value of the period of a cyclic systems expressed in user units.

<min value>, <max_value> (NA): value range for linear systems expressed in user units. The max. value must be greater than the min value.

DECLARATION EXAMPLE

```
DERIVATOR DER_Example;
      VALUE_PERIOD   = 360.0;
   END
```

INQUIRE FUNCTIONS

– **? ready**

This function asks if the Derivator is ready according to function under execution. In this release the ready is always TRUE.

– **? value**

This function returns the current output value of the Derivator.

EXAMPLE

```
IF (DER_Example ? value) > 1250 THEN ...
```

5.10.1 VALUE_PERIOD PARAMETER

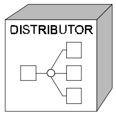
The parameter "**VALUE_PERIOD**" is defined to manage correctly the periodicity (modulus) of the input values. For example, if the input value increases each millisecond by one (degree) then the output value will be thousand (degrees per second). Now lets imagine that the input value skips suddenly from 359 to 0.

- If **VALUE PERIOD** = 360, the output will continue to indicate 1000 (degrees per second), indicating that roll-over into the next period has been properly handled.
- If **VALUE PERIOD** = 1000, the output will then indicate -359,000 (degrees per second), indicating that the input has incorrectly interpreted roll-over as a 359 degree change in input in one millisecond.

5.10.2 INITIAL BEHAVIOUR

The first calculation of a Derivator pipe block just after the pipe installation indicates zero regardless of the initial input value.

5.11 DISTRIBUTOR



PURPOSE

The distributor pipe block spreads the computations of all pipes of a pipes network over several pipes network periods. It becomes necessary to use a Distributor when the pipes network computation time is greater than the corresponding time available in a **BASIC_PAM_CYCLE**. Use of a Distributor results is less frequent sampling of those pipe blocks effected.

> ⚠ If PAM cannot complete all required pipe computations in the available time, it will stop in a fatal error condition.

BLOCK INPUT

The distributor input is the numerical entrance for the flow of values which represents the profile generated by the previous blocks of the pipe. This input is connected to the previous pipe block.

BLOCK OUTPUT

The distributor output is the numerical exit for the flow of values which is incoming through the block input. This output is connected to the next pipe block.

DECLARATION SYNTAX

DISTRIBUTOR <block identifier> ;

DIVISOR = <divisor value> ;

SHIFT = <shift value> ;

END

<block identifier>: name of the distributor block.

<divisor value>: (NA), number of time the pipes network sampling frequency is to be divided. Divisor value must be a positive integer number (≥ 0).

<shift value>: (NA), number of pipes network samples the pipe computing has to be shifted. Shift value must be a positive integer number (≥ 0) and must be lower than "divisor value".

INQUIRE FUNCTIONS

– **value**

Return the current numerical value coming out of the pipe block.

EXAMPLE:

```
CRV_DistOuput <- DIS_Example ? value ;
```

5.11.1 DISTRIBUTOR RULES

The following rules apply to Distributors used in a network:

- The distributor pipe block must be placed at the beginning of a pipe.
- The **DIVISOR** parameter values of all distributors used in the same pipes network should have the same value.

5.11.2 DISTRIBUTION PRINCIPLE

- The pipes network sampling period is given by the input block (generator or sampler).
- The period of a pipe without distributor is equal to the pipes network period.
- The period of a pipe with a distributor is equal to the pipes network period multiplied by the **DIVISOR** value.
- The computing point (in time) of a pipe with distributor is shifted a number of pipes network periods equal to it's **SHIFT** value.

5.11.3 EXAMPLE

Lets imagine a system controlled through a pipes network composed of 4 pipes. The pipes network sampling frequency must be 1 msec for pipe number 1, and can be 5 msec for the other pipes (2, 3 and 4). The total computing time of the whole pipes network exceeds the available time during one **BASIC_PAM_CYCLE**, so it is not possible to do all pipe network computations each millisecond.

The solution is to use Distributors for pipes 2, 3 and 4. Pipe 1 does not use a Distributor. The **BASIC_PAM_CYCLE** and the pipes network period are 1 millisecond.

The declaration is as follows:

```
DISTRIBUTOR DIS_Pipe2 ;
   DIVISOR     = 5 ;
   SHIFT = 0 ;       // executed first
END

DISTRIBUTOR DIS_Pipe3 ;
   DIVISOR     = 5 ;
   SHIFT = 1 ;       // executed second
END

DISTRIBUTOR DIS_Pipe4 ;
   DIVISOR     = 5 ;
   SHIFT = 2 ;       // executed third
END

// pipes creation-activation

CNV_Pipe1 << CAM_Pipe1 <<              TMP_VirtualMaster ;  // pipe 1

CNV_Pipe2 << CAM_Pipe2 << DIS_Pipe2 << TMP_VirtualMaster ;  // pipe 2

CNV_Pipe3 << CAM_Pipe3 << DIS_Pipe3 << TMP_VirtualMaster ;  // pipe 3

CNV_Pipe4 << CAM_Pipe4 << DIS_Pipe4 << TMP_VirtualMaster ;  // pipe 4
```

The result is as follows:

→ pipes network period ①: set-points for pipe 1 and pipe 2 computed

→ pipes network period ②: set-points for pipe 1 and pipe 3 computed

→ pipes network period ③: set-points for pipe 1 and pipe 4 computed

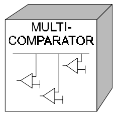
→ pipes network period ④: set-point for pipe 1 computed

→ pipes network period ⑤: set-point for pipe 1 computed

→ pipes network period ⑥: set-points for pipe 1 and pipe 2 computed

→ ...

5.12 MULTI-COMPARATOR



PURPOSE

The Multi-Comparator pipe block provides the capability for creating a fully auto-adaptive multi-channel trigger with independent delay compensation which, when triggered, starts a related **ROUTINE** with a very short reaction time. The Multi-Comparator block does not modify flow values and it has no effect on the axis and its periodicity.

BLOCK INPUT

The Multi-Comparator input is the numerical entrance for the flow of values which represent the profile generated by the previous blocks of the pipe. This input is connected to the previous pipe block.

BLOCK OUTPUT

The Multi-Comparator output is the numerical exit for the flow of values which are incoming through the block input. This output is connected to the next pipe block.

DECLARATION SYNTAX

MULTI_COMPARATOR <block identifier> ;
TIME_ORIGIN_SLOPE = <origin slope> ;
TIME_ORIGIN_COMPARE_MODE = <compare mode> ;
TIME_ORIGIN_REFERENCE = <origin value> ;
[**TIME_ORIGIN_ROUTINE** = { <routine statement> **|** **NONE** } ;]
{ **VALUE_PERIOD** = <period value> |
VALUE_RANGE = <min. value> <max. value>} **;**
{**TRACE** <trace identifier> ;
ROUTINE = { <routine statement> **|** **NONE** } ;
END}+
END

<block identifier> : (NA), the name of the Multi-Comparator object (string of characters).

*<origin slope> : (WO), indicates the sign of the first derivative (slope) of the pipe data flow at the origin position. The possibilities are: **POSITIVE**, **NEGATIVE**, **ZERO_MAXIMUM**, **ZERO_MINIMUM**.*

*<compare mode> : (WO), **IMMEDIATE** asks for activating the comparison with the origin reference immediately or in the current cycle for periodic systems. **NEXT_PERIOD** asks for activating the comparison only at the beginning of the next cycle for periodic systems.*

<origin value> : (RW), the origin value of the Multi-Comparator expressed in user units.

<routine statement> : (WO), any routine statement including the name of a routine object (string of characters) and its parameters (only constants).

<period value> : (RW), the value of the period of a cyclic system expressed in user units.

<min. value>, <max. value> : (WO), the value range for linear systems expressed in user units. The max. value must be greater than min. value.

<trace identifier> : (NA), the name of the trace sub-object (string of characters).

FUNCTIONS

The following executive functions are available for the Multi-Comparator:

– **learn**

Puts the Multi-Comparator in Learn Mode.

`Example:    MUL_Example <- learn ;`

– **execute**

Puts the multi Comparator in Execute Mode.

`Example:    MUL_Example <- execute ;`

The inquire functions available for the Multi-Comparator are as follows:

– **execute**

Test if the Multi-Comparator is in Execute Mode.

- TRUE if the Multi-Comparator is in Execute Mode.
- FALSE if the Multi-Comparator is in Learn Mode.

`Example:    IF (MUL_Example ? execute) THEN`

– **value**

Returns the current numerical value coming out of the pipe block.

Example: `IRV_MultiExampleOuput <- MUL_Example ? value ;`

5.12.1 OPERATING MODES

The Multi-Comparator works in two different modes:

- learn mode
- execute mode

Learn Mode, activates a self-calibration cycle during which the Multi-Comparator measures and records elapsed times from the time origin (reference event which initiates a Multi-Comparator cycle) to a set of references (trigger conditions) provided by the application and starts a series of **ROUTINES** linked to the references.

In Execute Mode, the Multi-Comparator utilises the elapsed times measured and recorded in Learn Mode to start the same series of **ROUTINES** each cycle.

Multi-compactor mode is selected using the **learn** and **execute** executive functions.

5.12.1.1 LEARN MODE OPERATION

In Learn Mode the Multi-Comparator monitors pipe flow values for concurrence of specific sets of pipe flow data characteristics (trigger conditions) called a references. References (see Figure 5-15) are defined by a position, a slope, and a comparison mode to localise exactly their position in the pipe data flow. When a reference becomes true, the Multi-compactor starts a related job (**ROUTINE**) with a very short reaction time and records the elapsed time relative to the Multi-compactor time origin (for subsequent use in Execute Mode).

References are stored in sub-objects called "Traces". Up to eight different references (but only one at a time) can be installed in a **TRACE** sub-object using the install_reference function. The number of **TRACE** sub-objects is only limited by the PAM internal memory. Only one reference per **TRACE** may be active at a time. The Multi-compactor simultaneously monitors pipe flow data for all active references and fires whenever any of the active references becomes true. Upon firing, execution of the **ROUTINE** connected to the trace that triggered the Multi-compactor is started.

References functions in a single-shot mode, so a reference is de-activated after it has fired . References are reactivated or modified by the application itself using the **install_reference** function.

A trace is connected to only one **ROUTINE** at a time, but the connected routine can be modified at any time by the application.

The Multi-Comparator must be referenced to an origin which is defined in the pipe block declaration. The origin also functions in single-shot mode, so the origin comparison is deactivated as soon as it has been triggered and started the origin routine. The origin comparison must be reactivated and eventually modified by the application itself.

5.12.1.2 EXECUTE MODE OPERATION

Execute Mode is similar to Learn Mode operation with the main difference being that elapsed times (measured in Learn Mode) instead of pipe flow data characteristics "fire" the Multi-compactor, and start **ROUTINES** connected to triggered Traces.

The actual trigger point (time) may be adjusted to compensate for actuator reaction time using the "delay compensation" parameter of a reference. In this case, the trigger time is adjusted to "elapsed time - delay compensation" (see Figure 5-15).

In Execute Mode, it is also possible to modify one or several parameters of any reference dynamically. In this case, the Multi-Comparator runs a partial learning cycle for the modified reference, then continues in Execute Mode. Other references are not affected during this partial learning process.

5.12.1.3 TIME ORIGIN REFERENCE

Upon occurrence of the set of pipe flow data characteristics described by the **TIME_ORIGIN** parameters, the Multi-compactor is triggered, establishing the time origin for the Multi-Comparator cycle (see Figure 5-15) and initiating execution of the **TIME_ORIGIN_ROUTINE**. The Multi-compactor also functions in single-shot mode and must be re-activated by the application after each cycle using the **learn** or **execute** functions.

Figure 5-14 shows the global principle of the Multi-Comparator and its ORIGIN and TRACE parameters.

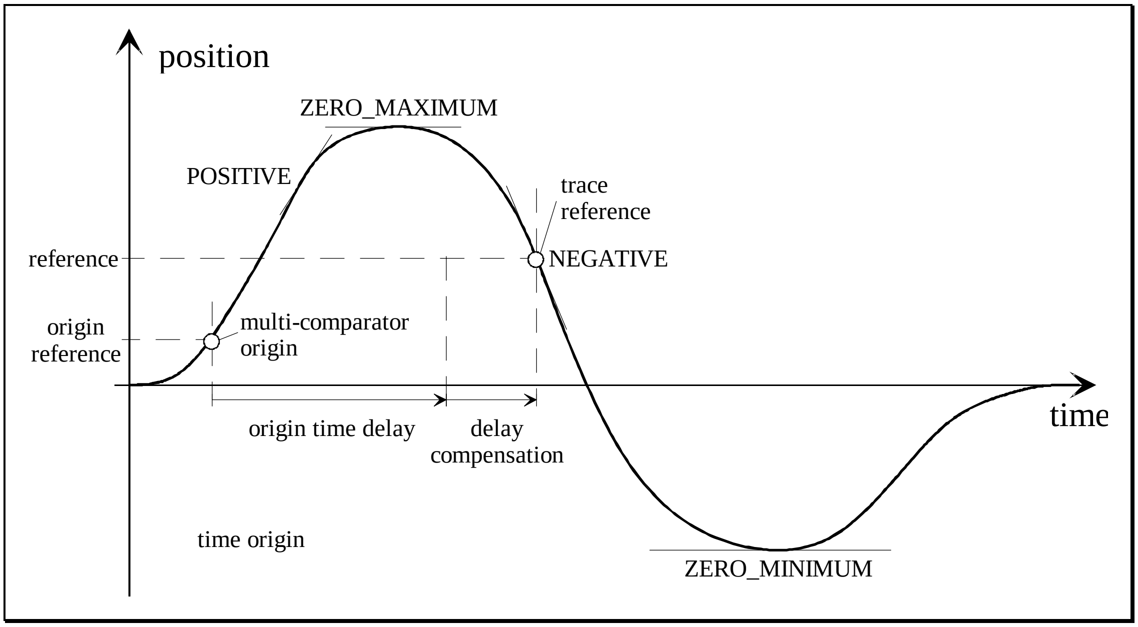


Figure 5-14 Reference and Time Origin Parameters Illustration

Figure 5-15 below shows comparison mode possibilities related to a periodic system.

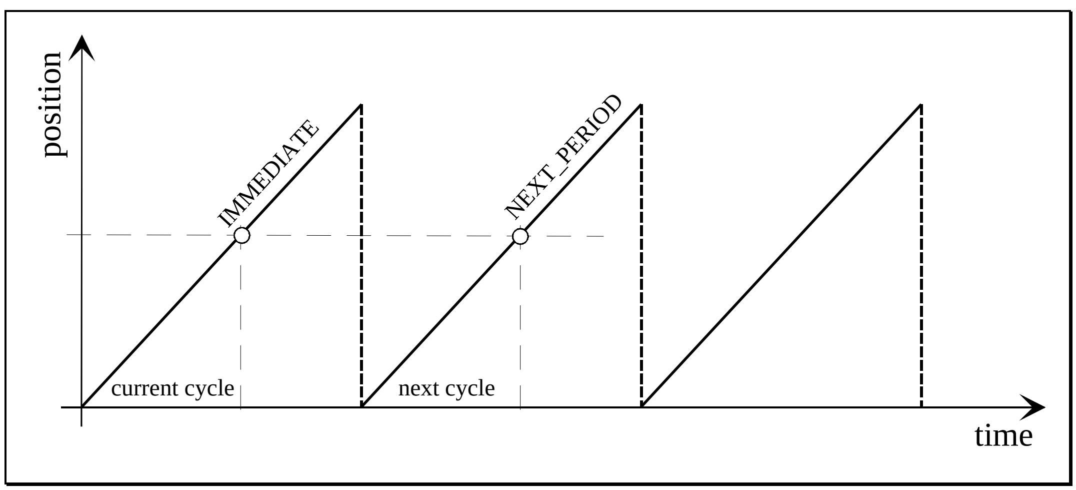


Figure 5-15 Compare_Mode Parameter

5.12.2 CONNECTING A ROUTINE

There are two ways to connect a **ROUTINE** to a trace and to a Multi-Comparator. The **ROUTINE** can be declared in the Multi-Comparator pipe block declaration. In this case <u>only constants</u> can be used as routine parameters.

It is also possible for the application to initialise or modify the **ROUTINE** trace sub-object parameter or the **TIME_ORIGIN_ROUTINE** MULTI-COMPARATOR parameter dynamically at run time. In this case any <u>expression</u> can be used as routine parameters.

The routine must have been previously defined in the application

SYNTAX

To connect a **ROUTINE** to the Multi-Comparator :

<block identifier>:**TIME_ORIGIN_ROUTINE** **<-** <routine syntax> ;

To connect a **ROUTINE** to a trace :

<trace identifier>:**ROUTINE** **<-** <routine syntax> ;

EXAMPLES

```
MUL_Example:TIME_ORIGIN_ROUTINE <- RTN_PainterStart ;
TRC_Example3:ROUTINE <- RTN_OpenDoor WITH CRV_DoorAmplitude ;
```

5.12.3 INSTALLING A REFERENCE

References are installed at run time by the application. Generally the first installation of references is done with the Multi-Comparator in Learn Mode to calibrate the time delay associated with each reference. After this first installation, the installation command is performed to reactivate the reference or to modify a reference parameter.

SYNTAX

<trace identifier> **<- install_reference (**<reference slope>, <compare mode>, <reference position>, <delay compensation>**) ;**

*<trace identifier> : (WO) name of the **TRACE** sub-object (string of characters).*

*<reference slope> : (WO) indicates the sign of the first derivative of the pipe data flow at the reference position. The possibilities are: **POSITIVE**, **NEGATIVE**, **ZERO_MAXIMUM**, **ZERO_MINIMUM**.*

<compare mode> : (WO) ***IMMEDIATE*** *activates the comparison immediately or in the current cycle for periodic systems.* ***NEXT_PERIOD*** *activates the comparison only at the beginning of the next cycle for periodic systems.*

<reference position> : (RW) the position of the reference in the pipe data flow expressed in user units.

<delay compensation> : (RW) the delay compensation value expressed in seconds. (for example, 0.052 gives 52 millisecond.)

5.12.4 HOW A MULTI-COMPARATOR WORKS

This description references the Multi-Comparator programming example listed in paragraph 5.12.5

First, to initiate the processing cycle, it is necessary to start (or restart) a Multi-Comparator. In sequence SystemEnable, the command:

```
MUL_Example <- learn
```

enables the Multi-compactor and places it in Learn Mode.

Now, the Multi-Comparator begins looking for the time origin reference. The traces are off and no trace references are defined.

To define a reference, the "install_reference" command is used. The statements:

```
TRC_Example1 <- install_reference(POSITIVE, IMMEDIATE,0.2, 0.010);
TRC_Example2 <- install_reference(POSITIVE, NEXT_PERIOD, 0.2, 0.010);
```

in sequence SystemEnable install the initial references in traces TRC_Example1 and TRC_Example2 and enable the references.

When the time origin is reached, the connected routine, RTN_ExampleOrigin, is executed, the traces are switched on and the Multi-compactor begins looking for the active references in traces TRC_Example1 and TRC_Example2.

Upon occurrence of a reference, the Multi-Comparator computes the origin time delay which is equal to the elapsed time (referenced to the origin) minus the delay compensation, then executes the **ROUTINE** connected to the **TRACE** where the next reference is installed and enabled. For example, the statement:

```
TRC_Example1<-install_reference(POSITIVE,IMMEDIATE, 0.4, 0.005);
```

installs and enables the next reference in trace TRC_Example1.

When the application determines that the learning cycle is complete, it switches the Multi-Comparator to Execute Mode and initiates a new Multi-compactor cycle. Now, the Multi-Comparator utilises the time delays previously learned to fire the references.

When a reference is reached, the **ROUTINE** connected to the **TRACE** is executed. To connect or change a **ROUTINE** connected to a **TRACE**, simply modify the **ROUTINE** parameter of the trace. For example:

```
TRC_Example1:ROUTINE <- RTN_ExampleTrace1 WITH 1.0;
```

When each reference of all traces are fired, the Multi-Comparator goes into the de-activated state. The application must re-activate the Multi-compactor and references for the next processing cycle. In sequence SystemEnable, the statements:

```
LOOP
     // wait that the last reference (second reference of Trace_2)
     // is reached
     CONDITION (CFV_FirstRun = 1);

     // when the last reference is found,
     // restart the multicomparator
     MUL_Example <- execute;
     CFV_FirstRun <- 0;
     TRC_Example1<-install_reference(POSITIVE,
                              IMMEDIATE,0.2,0.010);
     TRC_Example2<-install_reference(POSITIVE,
                              NEXT_PERIOD,0.2,0.010);
     ...;
  END_LOOP !CFV_EnableSystem;
```

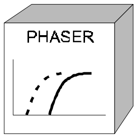
perform this function.

5.12.5 EXAMPLE

```
// routine attached with the multicomparator. It will be executed when
// the ORIGIN will be reached.

ROUTINES RGR_ExampleMain;
 ROUTINE RTN_ExampleOrigin;
    ...
 END_ROUTINE
END_ROUTINES

// declaration of the multi comparator.

MULTI_COMPARATOR MUL_Example ;
 TIME_ORIGIN_SLOPE = POSITIVE;
 TIME_ORIGIN_COMPARE_MODE= NEXT_PERIOD;
 TIME_ORIGIN_REFERENCE= 0.05;
 TIME_ORIGIN_ROUTINE  = RTN_ExampleOrigin;

 VALUE_PERIOD= 1.0;
 TRACE  TRC_Example1;
    ROUTINE  = NONE;
 END

 TRACE  TRC_Example2;
    ROUTINE  = NONE;
 END

END

// ROUTINE that will be attached to Trace_1 and Trace_2.

ROUTINES RGR_ExampleTraces;
 ROUTINE RTN_ExampleTrace1;
    IF CWV_CountTrace1 = 1 THEN
       // part for the first reference of the Trace_1
       ...
       CWV_CountTrace1 <- 2;

       // installation of the second reference the Traces_1.
       TRC_Example1<-install_reference(POSITIVE,
                              IMMEDIATE, 0.4, 0.005);
    ELSE IF CWV_CountTrace1 = 2 THEN
       // part for the second reference of the Trace_1
       ...
       CWV_CountTrace1 <- 1;
       END_IF
```

```
    END_IF
  END_ROUTINE

  ROUTINE RTN_ExampleTrace2;
    IF CWV_CountTrace2 = 1 THEN
      // part for the first reference of the Trace_2
      ...;
      CWV_CountTrace2 <- 2;

      // installation of the second reference the Traces_2.
      TRC_Example2<-install_reference(POSITIVE,
                              IMMEDIATE, 0.4, 0.005);
    ELSE IF CWV_CountTrace2 = 2 THEN
      // part for the second reference of the Trace_2
      ...;
      CWV_CountTrace2 <- 1;
      CFV_FirstRun <- 1;
      END_IF
    END_IF
  END_ROUTINE
END_ROUTINES


SEQUENCE SystemEnable ;
  CFV_FirstRun <- 0;
  CWV_CountTrace2 <- 1;
  CWV_CountTrace1 <- 1;
  TRC_Example1:ROUTINE <- RTN_ExampleTrace1;
  TRC_Example2:ROUTINE <- RTN_ExampleTrace2;

  // start the multicomparator in mode learn.
  MUL_Example <- learn;

  // put the multicomparator in a pipe.
  ... << MUL_Example  << ...;

  // installation of the first reference for the traces 1 and 2.
  TRC_Example1 <- install_reference(POSITIVE, IMMEDIATE,0.2, 0.010);
  TRC_Example2 <- install_reference(POSITIVE, NEXT_PERIOD, 0.2, 0.010);
  ...

  // loop in mode execute
  LOOP
    // wait that the last reference (second reference of Trace_2)
    // is reached
    CONDITION (CFV_FirstRun = 1);

    // when the last reference is found,
    // restart the multicomparator
    MUL_Example <- execute;
    CFV_FirstRun <- 0;
    TRC_Example1<-install_reference(POSITIVE,
                         IMMEDIATE,0.2,0.010);
    TRC_Example2<-install_reference(POSITIVE,
                         NEXT_PERIOD,0.2,0.010);
    ...;
  END_LOOP !CFV_EnableSystem;
  ...
END_SEQUENCE
```

5.13 PHASER



PURPOSE

The purpose of the phaser pipe block is to apply a phase shift to the values present at it's input. Changes to phase may be implemented in one step or at a specified rate. A phaser may also be started and stopped by command. When commanded to stop, the phaser assumes it's **STANDBY_VALUE**.

The phaser has some similarities with the amplifier (pipe block), however it's intended use is quite different. The typical application for a phaser pipe block is to drive a periodic system. That is a machine where the axes are globally increasing (or decreasing) their position. For this reason it has a **VALUE_PERIOD** parameter and other functions designed for continuously increasing (or decreasing) position. On the other hand, the amplifier pipe block, with **OFFSET** and **GAIN** parameters, is intended for bounded applications (applications where the integral of speed on a complete cycle is zero). Using the wrong one at the wrong place will cause unnecessary complications.

> ⚠ Please respect this rule during the design of your application. You will thereby avoid a number of problems while specifying the periodicity and other parameters specific to the application.

BLOCK INPUT

The phaser input is the numerical entrance for the flow of values which represent the profile generated by the previous blocks of the pipe. This input is connected to the previous pipe block.

BLOCK OUTPUT

The phaser output is the numerical output for the flow of phase shifted values. This output is connected to the next pipe block.

DECLARATION SYNTAX

PHASER <identifier> ;

PHASE = <phase value> ;

PHASE_SLOPE = <slope value> ;

STANDBY_VALUE = <standby value> ;

VALUE_PERIOD =<position period value>;

END

<identifier> : name of the phaser.

<phase value> : (RW), magnitude of the number added to the input value. Phase value may also be negative. A negative phase value is subtracted from the input value. Phase is expressed in user logical units.

*<slope value> : (RW), rate at which phase changes are implemented expressed in user logical units per second. A slope value of **MAX.** means that a phase change is fully implemented in a single step.*

<standby value> : (RW), value assumed by the phaser output when the phaser is in "stopped" condition, expressed in user logical units.

<position period value> : (RW), the position period for cyclic motion systems expressed in user logical units.

PHASER SAMPLE DECLARATION

```
PHASER PHA_Example ;
   PHASE         = 0.0 ;
   PHASE_SLOPE   = MAX ;
   STANDBY_VALUE = 0.0;
   VALUE_PERIOD  = 90.0 ;
END
```

EXECUTIVE FUNCTIONS

– **start**

When a phaser is started, it begins computing an output signal but its output remains at **STANDBY_VALUE** until the computed output signal next crosses the **STANDBY_VALUE**. At that point its output value is "connected to" the computed output.

– **stop**

When a phaser is stopped, it continues computing and outputting it's output value until the output value next crosses the **STANDBY_VALUE**, at which the output value is disconnected from the computed output and connected to the **STANDBY_VALUE.**

> ⚠ When the phaser is first used, without having received any command, it is started. To have it stopped, just add the stop command before any pipe activation.

When a phaser (in the started condition) receives a **stop** command, its **ready** variable is reset to false until it is fully stopped (its output value becomes **STANDBY_VALUE)**. At that time it becomes true again.

When a phaser (in the stopped condition) receives a **start** command, its **ready** variable is reset to false until it is fully started (its output value is "connected to" the computed output). At that time it becomes true again.

INQUIRE FUNCTIONS

– **value**

Returns current phaser output value.

– **ready**

Returns current value of ready variable.

5.13.1 PARAMETER MODIFICATIONS

5.13.1.1 PHASE

If the phaser is active and started when **PHASE** is changed, the phase change at the output is implemented at a rate determined by **PHASE SLOPE**. While the phase is changing, **ready** is false. If **PHASE_SLOPE** = **max.**, the phase change is fully implemented in one step.

If the phaser is active and stopped when **PHASE** is changed, the phase change is fully implemented in one step regardless of **PHASE SLOPE**.

If the phaser is not active a phase change is immediate regardless of **PHASE_SLOPE**. **Ready** remains true.

If the phaser is disactivated or stopped while a phase change is in process, any remaining portion of a phase change is immediately implemented then **ready** is set to true.

> i Note that the returned value when reading the phase parameter is the instantaneous value of **PHASE**. If the phase is changing, the returned value is between the old and the new values.

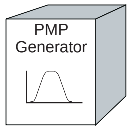
5.13.1.2 STANDBY VALUE

For proper calculations, it is necessary that **STANDBY_VALUE** is set according to the current value of **VALUE_PERIOD**. For this reason, the following manipulations of **STANDBY_VALUE** are performed by PAM:

- As long as the phaser is not activated, **STANDBY_VALUE** and **VALUE_PERIOD** remain independent.
- When the pipe (including the phaser) is activated, **STANDBY_VALUE** is reduced to the corresponding value modulo **VALUE_PERIOD**.
- If **STANDBY_VALUE** is modified while the phaser is active, its value is immediately reduced to the corresponding value modulo **VALUE_PERIOD**.
- This means, for example, that a **STANDBY_VALUE** equal to 1.5 times **VALUE_PERIOD** entered when the pipe is not activated, will read 0.5 times **VALUE_PERIOD** after the pipe is activated.

IF **STANDBY_VALUE** of an active phaser in stopped condition is changed, the new **STANDBY_VALUE** is immediately connected to the phaser output.

5.14 PMP GENERATOR



PURPOSE

The purpose of the PMP (Parabolic Motion Profile) generator block is to generate a flow of values with a second derivative (acceleration) which produces a trapezoidal trajectory. This generator is useful in applications where jerk (third derivative of the motion) limiting is necessary. These values are pure logical values, with generally no direct physical representation. It is a source of one or several pipes.

The PMP motion profile generator block can be considered as a virtual master for the system if several pipes are connected to it because it synchronises all axes which are linked to these pipes. Generally, a PMP generator is used to generate flow values with a position dimension.

USES

The PMP generator can generate a simple point-to-point profile. It can also generate a forward-backward profile with a non-stop zero transition. This profile (see Figure 5-16) can have different forward and backward distances and travel speeds (**FIRST_TRAVEL_SPEED** and **LAST_TRAVEL_SPEED**).

The PMP generator performs some pre-calculation before each movement, so it is not possible to modify the motion parameters on the fly during profile generation. If a command is sent to the PMP generator while computing or executing a profile, this command is ignored and lost.

If the PMP generator is executing a very short movement, it may not be able to achieve the specified travel speed, acceleration or jerk; however, the PMP profile shape will always be implemented.

BLOCK INPUT

The PMP generator has no input because it is an input block of a pipe.

BLOCK OUTPUT

The PMP generator output is the numerical exit for the flow of generated values. This output is connected to the next pipe block.

DECLARATION SYNTAX

PMP_GENERATOR <block identifier> ;

FIRST_TRAVEL_SPEED = <travel speed value> ;

LAST_TRAVEL_SPEED = <travel speed value> ;

ACCELERATION = <acceleration value> ;

JERK = <jerk value> ;

INITIAL_POSITION = <initial position value> ;

{ **POSITION_PERIOD** = <position period value> |

POSITION_RANGE = <min. position value> <max. position value>};

PERIOD = <sampling period value> ;

END

The **POSITION_RANGE** parameter values use is not yet implemented. But it must be specified with dummy values to declare a non-periodic system.

<block identifier> : name of the PMP generator.

<travel speed value> : (RW), travel speed value expressed in user logical units per second. This value must be greater than zero.

<acceleration value> : (RW), acceleration value expressed in user logical units per second squared. This value must be greater than zero.

<jerk value> : (RW), jerk value expressed in user logical units per second cubed. This value must be greater than zero.

<initial position value> : (RW), initial position value expressed in user logical units, used only at the pipe activation to initialise the position starting point.

<position period value> : (RW), position period for cyclic motion systems expressed in user logical units. This value must be greater than zero.

<min. position value> <max. position value> : (NA), position range for linear motion systems expressed in user logical units. This max. value must be greater than the min. value.

*<sampling period value> : (RO), sampling period of the generator expressed in seconds. This value must be greater than zero. PAM adjusts **PERIOD** to the closest multiple of **BASIC_PAM_CYCLE**, and this becomes the effective sampling period (i.e. if **BASIC_PAM_CYCLE** = 6, and **PERIOD** = 0.001 is specified in the TMP declaration, the effective TMP generator sampling period is 0.002 seconds). The value returned when reading **PERIOD** is the effective sampling period.*

*<anti-delay value> : (NA), enables or disables compensation for delays between time values are calculated by PAM and executed by axes. Anti-delay value is **YES** for anti-delay enabled, or **NO** for anti-delay disabled. For simple applications **ANTI-DELAY** = **NO** is recommended because anti-delay can tend to introduce some noise into the positions manipulated by the driven axis. In more critical applications where it is essential that the positions manipulated by axes are precisely the same at the same point in time as those calculated by PAM, **ANTI-DELAY** = **YES** should be used.*

> ⚠ The default value for **ANTI_DELAY** is **YES** for compatibility with previous versions. We strongly recommend initialising **ANTI_DELAY** = **NO** whenever possible.

PMP GENERATOR SAMPLE DECLARATION

```
PMP_GENERATOR PMP_Example;
   FIRST_TRAVEL_SPEED   =    6000.0  ;
   LAST_TRAVEL_SPEED    =    6000.0  ;
   ACCELERATION         =    6000.0  ;
   JERK                 = 2000000.0  ;
   INITIAL_POSITION     =       0.0  ;
   POSITION_PERIOD      =     360.0  ;
   PERIOD               =       0.001;
   ANTI_DELAY           =       YES  ;
END
```

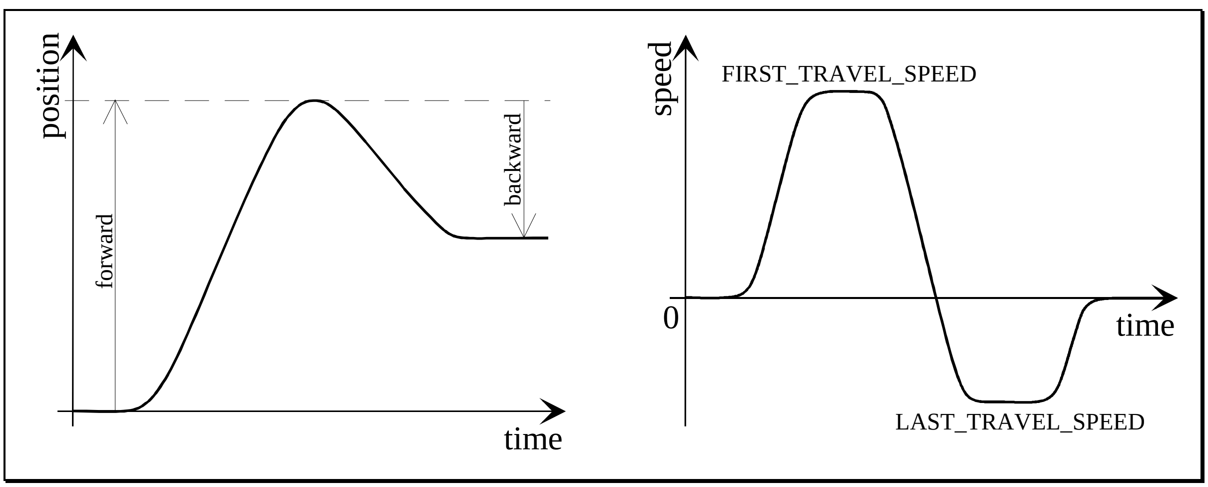


Figure 5-16 PMP Parameters Illustration

EXECUTIVE FUNCTIONS

– **position**

This function sets the current position of the PMP generator.

> ⚠ If the current PMP position is modified while the block is active, a step change to the new current position will occur at the pipe block output.

If the PMP block is not active (not used in any pipe at this time), the new position value will be used to set the initial position at subsequent PMP activation. Do not use this method for modifying initial position, rather change **INITIAL_POSITION** using a parameter modification command. This behaviour is retained only for compatibility with previous software versions.

Example

```
PMP_Example <- position (70.0) ;
```

– **absolute_move**

This function commands the generator to perform a point-to-point motion to an absolute position if one parameter is specified or a forward-backward motion between absolute positions if two parameters are specified.

Forward and backward displacements must be in opposite directions.

Examples:

```
PMP_Example <- absolute_move (12000.0) ;
PMP_Example <- absolute_move (12000.0, 8000.0) ;
```

– **relative_move**

This function asks the generator to perform a point-to-point motion to a relative position if one parameter is specified or a forward-backward motion to relative positions if two parameters are specified (the second relative position is related to the end of the first motion).

Forward and backward displacements must be in opposite directions.

Examples:

```
PMP_Example <- relative_move (4500.0) ;
PMP_Example <- relative_move (4500.0, -2000.0) ;
```

INQUIRE FUNCTIONS

– **position**

This function asks the current position of the generator.

– **speed**

This function asks the current speed of the generator.

– **ready**

This function asks if the generator is ready according to function under execution.

5.14.1 PARAMETERS MODIFICATION

5.14.1.1 TRAVEL_SPEED

Parameter access permits changes to the **FIRST_TRAVEL_SPEED** and **LAST_TRAVEL_SPEED** parameter values specified in the **PMP_GENERATOR** declaration. The travel speed values are always used to set the constant speed part of the motion profile.

The following rule is applied when the travel speed values are modified:

- If a PMP motion is in progress, the new travel speed value will be used only for subsequent PMP motions.

EXAMPLE

```
PMP_Example:FIRST_TRAVEL_SPEED <- (151.0) ;
```

5.14.1.2 ACCELERATION

Parameter access permits changes to the **ACCELERATION** parameter value specified in the **PMP_GENERATOR** declaration. The acceleration value (subject to constraints imposed by the **JERK** parameter) is always used to generate the portions of the motion profile where velocity is changing.

The following rules are applied when the acceleration value is modified:

- If the new acceleration value is less than or equal to zero, the current value is not replaced by the new value.
- If a PMP motion is in progress, the new acceleration value will be used only for subsequent PMP motions.

EXAMPLE

```
PMP_Example:ACCELERATION <- (1510.0) ;
```

5.14.1.3 JERK

Parameter access permits changes to the **JERK** parameter value specified in the **PMP_GENERATOR** declaration. The jerk value is used to generate rounded part of the speed motion profile. Jerk is the derivative of the acceleration, so it specifies the acceleration ramp.

The following rules are applied when the jerk value is modified:

- If the new jerk value is less than or equal to zero, the current value is not replaced by the new value.
- If a PMP motion is in progress, the new jerk value will be used only for subsequent PMP motions.

EXAMPLE

```
PMP_Example:JERK <- (15100.0) ;
```

5.14.1.4 INITIAL POSITION

The following rules are applied when **INITIAL POSITION** is modified:

- If the PMP block is not active (not used in any pipe at this time), the new initial position value will be used to set the initial position for subsequent PMP activation.
- If the PMP block is active, the new initial position value will be implemented at the next pipe activation.

5.14.1.5 EXAMPLES

This example shows a PMP_GENERATOR declaration and subsequent relative move function.

```
PMP_GENERATOR PMP_Example;
   FIRST_TRAVEL_SPEED     = 9000.0 ;  // in logical units/s
   LAST_TRAVEL_SPEED = 4500.0 ;  // in logical units/s
   ACCELERATION      = 45000.0 ; // in logical units/s2
   JERK  = 450000.0 ;      // in logical units/s3
   POSITION    = 0.0 ;     // in logical units
   POSITION_RANGE    = 0.0 10000.0 ;   // in logical units
   PERIOD      = 0.001 ;   // in second
END

   ...

// relative move of 4500 units
PMP_Example <- relative_move(4500.0) ;
```

Figure 5-17 shows the position, speed, acceleration and jerk profiles generated by the previous command.

i	The basic profile shape is illustrated in the velocity diagram, but the profile of the PMP block output flow values is illustrated by the position diagram.

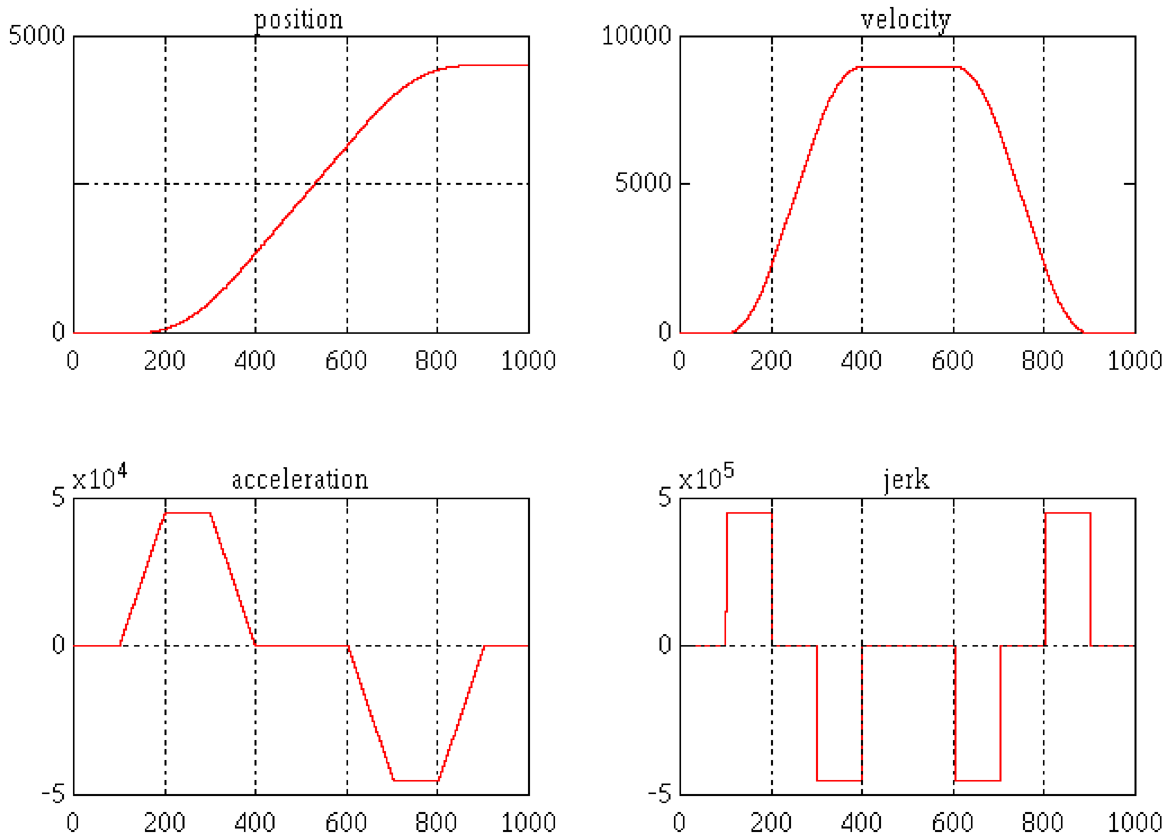


Figure 5-17 PMP Motion Profiles for a Relative Move

The following statement commands a move of 4500 units forward followed immediately by a backward move of 2000 units.

```
// relative move of 4500 units forward and 2000 backward
PMP_Example <- relative_move(4500.0 -2000.0) ;
```

Figure 5-18 shows the position, speed, acceleration and jerk profiles generated by the previous command.

i | The basic profile shape is the speed diagram, but the profile of the PMP block output flow values is illustrated by the position diagram..

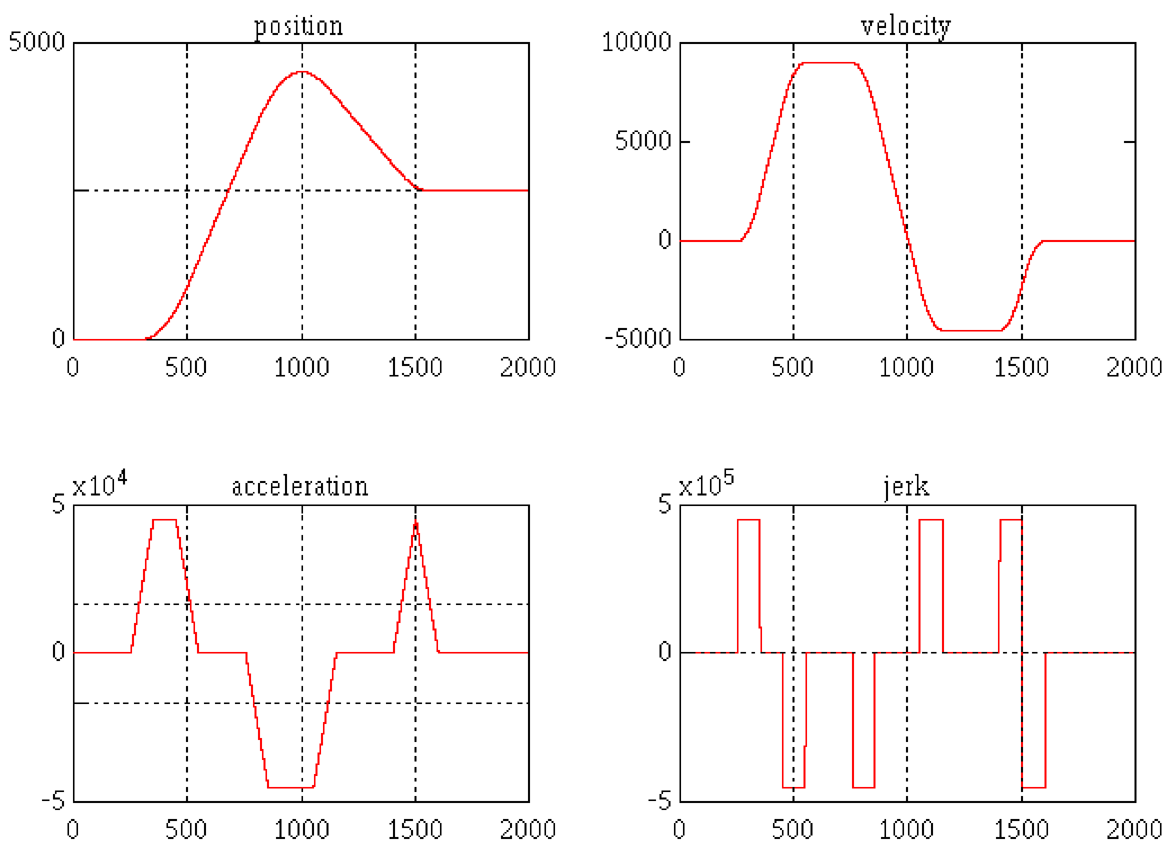
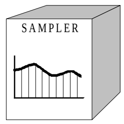


Figure 5-18 PMP Motion Profiles for a Forward-Backward Motion

5.15 SAMPLER



PURPOSE

The purpose of the sampler block is to periodically sample and place into a pipe some output of a source object. The sampled output might typically be the POSITION or SPEED of a source object measured by a resolver, an encoder or some other types of sensor.

The sampler implements the logical connection between an encoder on a physical master axis (the source object) and one or more pipes and performs the function of periodically sampling the source and placing the sampled values into the pipe.

BLOCK INPUT

The sampler has no input because it is an input block of a pipe.

BLOCK OUTPUT

The sampler output is the numerical exit for the flow of sampled values. This output is connected to the next pipe block.

DECLARATION SYNTAX

SAMPLER <identifier> ;

SOURCE = <source object> ;

MODE =<working mode> ;

PERIOD= <sampling period value> ;

END

<identifier> : name of the sampler.

<source object> : (NA), source object of the pipe.

<working mode> : (NA), working mode of the sampler. The available modes are: ***POSITION*** *and* ***SPEED****.*

<sampling period value> : (RO), period of the sampler expressed in seconds. This value must be greater than zero. PAM adjusts ***PERIOD*** *to the closest multiple of* ***BASIC_PAM_CYCLE****, and this becomes the effective sampling period (i.e. if* ***BASIC_PAM_CYCLE*** *= 6, and* ***PERIOD*** *= 0.001 is specified in the TMP declaration, the effective TMP generator sampling period is 0.002 seconds). The value returned when reading* ***PERIOD*** *is the effective sampling period.*

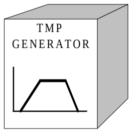
DECLARATION EXAMPLE

```
SAMPLER SMP_Leader;
   SOURCE   = ENC_Pos;
   MODE     = POSITION;
   PERIOD   = 0.01;
END
```

FUNCTIONS

There are no functions available for the sampler.

5.16 TRAPEZOIDAL MOTION PROFILE GENERATOR



PURPOSE

The purpose of the trapezoidal motion profile generator block is to generate a flow of values with a first derivative which produces a trapezoidal trajectory. These values are pure logical values, with generally no direct physical representation. It is a source block which frequently serves as a virtual master for a system comprised of several pipes. Generally, a trapezoidal motion profile generator is used to generate flow values with a position dimension.

BLOCK INPUT

The trapezoidal motion profile generator has no input because it is an input block of a pipe.

BLOCK OUTPUT

The trapezoidal motion profile generator output is the numerical exit for the flow of generated values. This output is connected to the next pipe block.

DECLARATION SYNTAX

TMP_GENERATOR <identifier> ;

TRAVEL_SPEED = <travel speed value> ;

ACCELERATION = <acceleration value> ;

[**DECELERATION** = <deceleration value> ;]

INITIAL_POSITION = <initial position value> ;

{ **POSITION_PERIOD** = <position period value> |

POSITION_RANGE = <min. position value> <max. position value>};

PERIOD = <sampling period value> ;

ANTI_DELAY = <anti-delay value> ;

END

stop — The **POSITION_RANGE** parameter values use is not yet implemented, but it must be specified with dummy values to declare a non-periodic system.

<identifier> : name of the trapezoidal motion profile generator.

<travel speed value> : (RW), travel speed value expressed in user logical units per second. This value must be greater than zero.

<acceleration value> : (RW), acceleration value expressed in user logical units per second squared. This value must be greater than zero.

<deceleration value> : (RW), deceleration value expressed in user logical units per second squared. This value must be greater than zero. If this value is omitted, the acceleration value is used.

<initial position value> : (RW), initial position value expressed in user logical units. Used only at the pipe activation to initialise the position starting point.

<position period value> : (RW), position period for cyclic motion systems expressed in user logical units. This value must be greater than zero.

<min. position value> <max. position value> : (NA), position range for linear motion systems expressed in user logical units. This max. value must be greater than the min. value.

*<sampling period value> : (RO), sampling period of the TMP generator expressed in seconds. This value must be greater than zero. PAM adjusts **PERIOD** to the closest multiple of **BASIC_PAM_CYCLE**, and this becomes the effective sampling period (i.e. if **BASIC_PAM_CYCLE** = 6, and **PERIOD** = 0.001 is specified in the TMP declaration, the effective TMP generator sampling period is 0.002 seconds). The value returned when reading **PERIOD** is the effective sampling period.*

*<anti-delay value> : (NA), enables or disables compensation for delays between time when values are calculated by PAM and executed by axes. Anti-delay value is **YES** for anti-delay enabled, or **NO** for anti-delay disabled. For simple applications **ANTI_DELAY** = **NO** is recommended because anti-delay can tend to introduce some noise into the positions manipulated by the driven axis. In more critical applications where it is essential that the positions manipulated by axes are precisely the same at the same point in time as those calculated by PAM, **ANTI_DELAY** = **YES** should be used.*

> ! The default value for **ANTI_DELAY** is **YES** for compatibility with previous versions. We strongly recommend initialising **ANTI_DELAY** = **NO** whenever possible.

TMP GENERATOR SAMPLE DECLARATION

```
TMP_GENERATOR TMP_Example;
   TRAVEL_SPEED     =   360.0   ;
   ACCELERATION     = 36000.0   ;
   INITIAL_POSITION =     0.0   ;
   POSITION_PERIOD  =   360.0   ;
   PERIOD           =     0.001 ;
   ANTI_DELAY       =     YES   ;
END
```

EXECUTIVE FUNCTIONS

– **position**

This function sets the current position of the TMP generator.

> If the current TMP position is modified while the block is active, a step change to the new current position will occur at the pipe block output.

> If the TMP block is not active (not used in any pipe at this time), the new position value will be used to set the initial position at subsequent TMP activation. Do not use this method for modifying initial position, rather change **INITIAL_POSITION** using a parameter modification command. This behaviour is retained only for compatibility with previous software versions.

Syntax

The syntax of the statement is as follows:

<object identifier> **<- position (**<position value>**)**

Example

```
TMP_Example <- position (70.0) ;
```

– **absolute_move**

This function asks the generator to perform a trapezoidal motion to an absolute position.

– **relative_move**

This function asks the generator to perform a trapezoidal motion to a position relative to current position.

– **run**

This function asks the generator to perform continuous motion at the specified speed.

INQUIRE FUNCTIONS

– **position**

This function asks the current position of the generator.

– **speed**

This function asks the current speed of the generator.

- acceleration

This function asks the current acceleration of the generator.

- ready

This function asks if the generator is ready according to function under execution.

5.16.1 TRAVEL_SPEED PARAMETER MODIFICATION

Parameter access permits changes to the **TRAVEL_SPEED** parameter value specified in the **TMP_GENERATOR** declaration. The travel speed value is always used to set the constant speed part of the trapezoidal motion profile.

The following rule is applied when the travel speed value is modified:

- If a trapezoidal motion is in progress, the new travel speed value will be used only for subsequent trapezoidal motions.

EXAMPLE

```
TMP_Example:TRAVEL_SPEED <- (3.22) ;
```

5.16.2 ACCELERATION PARAMETER MODIFICATION

Parameter access permits changes to the **ACCELERATION** parameter value specified in the **TMP_GENERATOR** declaration. The acceleration value is always used to generate the first part of the trapezoidal motion profile.

The following rules are applied when the acceleration value is modified:

- If the new acceleration value is less than or equal to zero, the current value is not replaced by the new value.
- If a trapezoidal motion is in progress, the new acceleration value will be used only for subsequent trapezoidal motions.

EXAMPLE

```
TMP_Example:ACCELERATION <- (3.22) ;
```

5.16.3 DECELERATION PARAMETER MODIFICATION

Parameter access permits changes to the **DECELERATION** parameter value specified in the **TMP_GENERATOR** declaration. The deceleration value is always used to generate the last part of the trapezoidal motion profile.

The following rules are applied when the deceleration value is modified:

- If the new deceleration value is less than or equal to zero, the current value is not replaced by the new value.
- If a trapezoidal motion is in progress, the new deceleration value will be used only for subsequent trapezoidal motions.

EXAMPLE

```
TMP_Example:DECELERATION <- (3.22) ;
```

5.16.4 INITIAL POSITION PARAMETER

The following rules are applied when **INITIAL POSITION** is modified:

- If the TMP block is not active (not used in any pipe at this time), the new initial position value will be used to set the initial position for subsequent TMP activation.
- If the TMP block is active, the new initial position value will be implemented at the next pipe activation.

5.16.5 EXAMPLE

This example shows a **TMP_GENERATOR** declaration followed by a **TRAVEL_SPEED** modification and relative move function.

```
TMP_GENERATOR TMP_Example;
   TRAVEL_SPEED   = 50.0 ;    // in logical units/s
   ACCELERATION   = 0.3 ;     // in logical units/s2
   DECELERATION   = 0.6 ;     // in logical units/s2
   INITIAL_POSITION    = 0.0 ;     // in logical units
   POSITION_PERIOD     = 678 ;     // in logical units
   PERIOD         = 0.001 ;   // in second
END
   ...
TMP_Example:TRAVEL_SPEED <- 6.0 ;  // new current travel speed
TMP_Example <- relative_move(450) ;// relative move of 450 units
```

Figure 5-19 illustrates the position, velocity and acceleration trajectories resulting from the **TMP_GENERATOR** declaration, parameter modification and subsequent function statement.

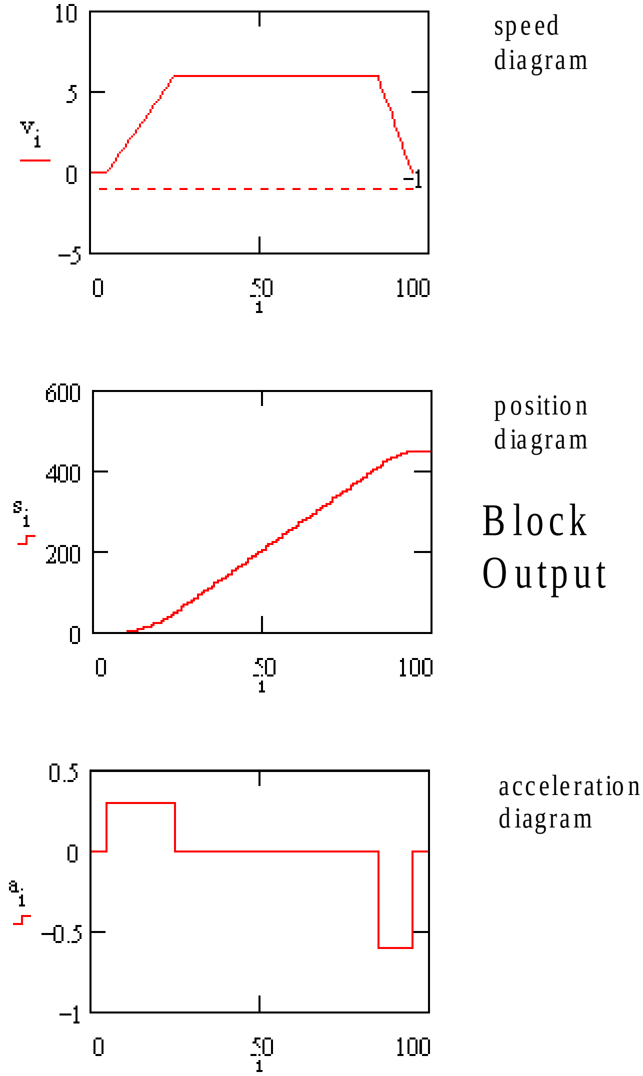


Figure 5-19 Trajectories resulting from Example Program

6 MATHEMATICAL FUNCTIONS AND OPERATORS

6.1 INTRODUCTION

This Chapter describes the mathematical functions and operators which are available for use in expressions and statements and presents examples of their use.

6.2 MATHEMATICAL FUNCTIONS

6.2.1 GENERAL SYNTAX

Mathematical functions are used like other inquire functions. The general syntax is as follows:

<argument> **?** <function>

6.2.2 EXAMPLES

Assign sin(π) to MyVariable:

```
IRV_MyVariable <- Pi ? sin ;
```

Assign cos (2*x+23) to MyVariable :

```
IRV_MyVariable <-  (2*IRV_x+23)  ? cos ;
```

Assign (2*x)+cos(23) to MyVariable :

```
IRV_MyVariable <- 2 * IRV_x + 23 ? cos ;
```

Assign tan (x / y) * 3 to MyVariable :

```
IRV_MyVariable <- (IRV_x / IRV_y) ? tan * 3 ;
```

Assign asin (2* sin(x)) to MyVariable :

```
IRV_MyVariable <- (IRV_x ? sin * 2) ? asin ;
```

6.2.3 ABS, CEIL, FLOOR

The function *Abs* computes the absolute value of the argument.

The function *Ceil* returns the smallest floating-point number not less than the argument whose value is an exact mathematical integer.

The function *floor* returns the largest floating-point number not greater than the argument whose value is an exact mathematical integer.

6.2.4 EXP, LN, LOG10, SQRT

The function *Exp* computes the exponential function of the argument; that is, e^{argument}, where *e* is the base of the natural logarithm. An error can occur for large arguments.

The function *Ln* computes the natural function of the argument. If the argument is negative, an error occurs. If the argument is zero or close to zero, an error may occur.

The function *Log10* computes the base-10 logarithm of the argument. If the argument is negative, an error occurs. If the argument is zero or close to zero, an error may occur.

The function Sqrt computes the non-negative square root of the argument. An error occurs if the argument is negative.

6.2.5 COS, SIN, TAN

The function *cos* computes the trigonometric cosine of the argument, which is taken to be in radians. If the argument is very large, the result may not be meaningful.

The function *sin* computes the trigonometric sine of the argument, which is taken to be in radians. If the argument is very large, the result may not be meaningful.

The function *tan* computes the trigonometric tangent of the argument, which is taken to be in radians. If the argument is very large, the result may not be meaningful. An error may occur if the argument is close to an odd multiple of $\pi/2$.

6.2.6 ACOS, ASIN, ATAN

The function *Acos* computes the principal value of the trigonometric arc cosine function of the argument. The result is in radians and lies between 0 and π. An error occurs if the argument is less than -1.0 or greater 1.0 .

The function *asin* computes the principal value of the trigonometric arc sine function of the argument. The result is in radians and lies between $-\pi/2$ and $\pi/2$. An error occurs if the argument is less than -1.0 or greater 1.0 .

The function *Atan* computes the principal value of the trigonometric arc tangent function of the argument. The result is in radians and lies between $-\pi/2$ and $\pi/2$.

6.2.7 COSH, SINH, TANH

The function *cosh* computes the hyperbolic cosine of the argument. An error can occur if the absolute value of the argument is large.

The function *Sinh* computes the hyperbolic sine of the argument. An error can occur if the absolute value of the argument is large.

The function *Tanh* computes the hyperbolic tangent of the argument.

6.3 MATHEMATICAL OPERATORS

The mathematical operators described below are available for use in expressions and statements.

6.3.1 GENERAL SYNTAX

<operand> **<operator>** <operand>

6.3.2 ADDITION, SUBTRACTION, MULTIPLICATION, DIVISION

EXAMPLES

Assign x + y to MyVariable

```
IRV_MyVariable <-  IRV_x + IRV_y ;
```

Assign x^y to MyVariable ;

```
IRV_MyVariable <-  IRV_x ^ IRV_y ;
```

Assign $(x*z)^y$ to MyVariable ;

```
IRV_MyVariable <- IRV_x * IRV_z ^ IRV_y ;
```

6.3.3 RAISING TO A POWER

The operator ^ computes the value of the first operand raised to the power of the value of the second operand. An error occurs if the first operand is zero and the second operand is not positive or equal to zero, or if the first operand is negative and the second operand is not an exact integer. If the first operand is zero and the second operand is positive, the result is zero. If the first operand is not zero and the second operand is zero, the result is 1.0 .

EXAMPLE

Assign $(x^y)*z$ to MyVariable ;

```
IRV_MyVariable <- IRV_x ^ IRV_y * IRV_z;
```

6.3.4 REMAINDER

The operator **%** computes the remainder of the integral division of the first operand by the second operand. The result has the same sign as the first operand. If the second operand is zero, the result is the first operand.

EXAMPLE

Assign the remainder of x / y to MyVariable ;

```
IRV_MyVariable <- IRV_x % IRV_y
```

6.4 MATHEMATICAL CONSTANTS

One mathematical constant is available.

Pi (π)

The value used for "Pi" is 3.141592653589793 .

6.5 PRECEDENCE AND ASSOCIATIVITY OF OPERATORS

Table 6-1 shows the operators in order of precedence:

CLASS OF OPERATOR	OPERATORS IN THAT CLASS	ASSOCIATIVITY	PRECEDENCE
primary	()	left-to-right	HIGHEST
unary	+ - !	left-to-right	
inquire	?	left-to-right	
multiplicative	* / % ^	left-to-right	
additive	+ -	left-to-right	
relational	< <= > >=	left-to-right	
equality	= <>	left-to-right	LOWEST

Table 6-1 Precedence and Associativity of Operators

7 PHYSICAL & APPL. OBJECT EXECUTIVE FUNCTIONS

This chapter describes each of the executive functions available for the various physical objects and application objects described in Chapters 2 and 3. Note that each of the executive functions applies only to the objects listed for the function. These functions are not applicable to pipe blocks.

7.1 COMPATIBILITY WITH PREVIOUS SOFTWARE VERSIONS

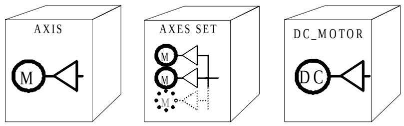
The executive functions listed in Table 7-1 are no longer available beginning with this version of the PAM software. The actions performed by these deleted functions may now be performed using parameter modification statements (see paragraph 4.3.4.2). The rules and considerations applicable to parameter modifications for each type of physical object are found in Chapter 2.

FUNCTION DELETED	AFFECTED OBJECTS
acceleration	axis, axes set
deceleration	axis, axes set
travel_speed	axis, axes set

Table 7-1 Executive Functions no longer available

7.2 ABSOLUTE_MOVE

APPLIES TO



PURPOSE

This function commands an axis, an axes set or a DC-motor to perform a trapezoidal motion to a specified absolute position. Several cases must be considered:

a) no motion is in progress,

b) a trapezoidal motion is in progress,

c) a continuous motion is in progress,

d) an arbitrary motion generated by a pipe is in progress.

If no motion is in progress when the absolute_move function is called, the object perform a trapezoidal motion to the specified position.

If a trapezoidal motion is in progress when the absolute_move function is called, the current trapezoidal motion is replaced by a new trapezoidal motion to the new position.

If a continuous motion is in progress, this motion is cancelled and the object perform a trapezoidal motion to the specified position.

If an arbitrary motion generated by a pipe is in progress, the trapezoidal motion is superimposed. The result is an absolute position shift. (possible only with an axis or an axes set).

> i — To stop this superimposed motion without stopping the motion generated by a pipe, perform the **run** function with a zero parameter value instead of the **stop** function.

SYNTAX

<object identifier> **<- absolute_move (**<absolute position value>**)**

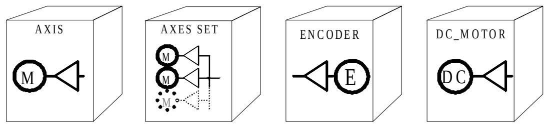
<absolute position value> : absolute position value expressed in user length units.

EXAMPLES

```
AXI_AnAxis <- absolute_move (3.22) ;
AXI_Axes[i] <- absolute_move (CRV_TargetPosition[i]) ;
AXI_Axes[all] <- absolute_move ((CRV_TargetPosition * 1.25) - 1) ;
```

7.3 POSITION

APPLIES TO



PURPOSE

This function performs the absolute reference position initialisation of an axis, an axes set, an encoder or a DC-motor <u>without making any movement</u>.

SYNTAX

<object identifier> **<- position (**<position value>**)**

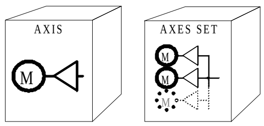
<position value> : the new position value expressed in user length units.

EXAMPLES

```
AXI_AnAxis <- position (0) ;
AXI_Axes[i] <- position (23) ;
AXI_Axes[all] <- position (0) ;
```

7.4 POWER_OFF

APPLIES TO



PURPOSE

This function performs the power-down of an axis or an axes set.

> i | If the axis was moving, a <u>stop</u> function is performed before the power-down.

SYNTAX

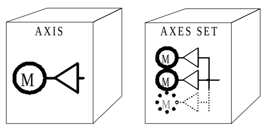
<axis identifier> **<- power_off**

EXAMPLES

```
AXI_AnAxis <- power_off ;
AXI_Axes[i] <- power_off ;
AXI_Axes[all] <- power_off ;
```

7.5 POWER_ON

APPLIES TO



PURPOSE

This function performs the power-up of an axis or an axes set.

i | The absolute position of the axis is set to zero but the axis doesn't move.

SYNTAX

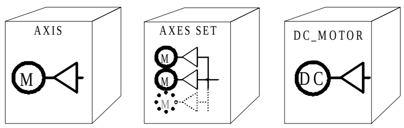
<axis identifier> **<- power_on**

EXAMPLES

```
AXI_AnAxis <- power_on ;
AXI_Axes[i] <- power_on ;
AXI_Axes[all] <- power_on ;
```

7.6 RELATIVE_MOVE

APPLIES TO



PURPOSE

This function commands an axis, an axes set or a DC-motor to perform a trapezoidal motion to a relative specified position.

> This motion can be superimposed to any motion generated by a pipe connected to the axis or the axes set.
>
> To stop this superimposed motion without stopping the motion generated by a pipe, perform the **run** function with a zero parameter value instead of the **stop** function.

SYNTAX

<object identifier> **<- relative_move (**<relative position value>**)**

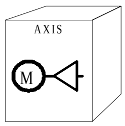
<relative position value> : the relative position value expressed in user length units.

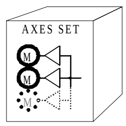
EXAMPLES

```
AXI_AnAxis <- relative_move (3.22) ;
AXI_Axes[i] <- relative_move (IRV_p[i]) ;
AXI_Axes[all] <- relative_move (CWV_b * 1.25 - 1) ;
```

7.7 RUN

APPLIES TO





PURPOSE

This function commands an axis or an axes set to perform continuous motion at constant speed.

This motion can be superimposed to any motion generated by a pipe connected to the axis or the axes set.

To stop this superimposed motion without stopping the motion generated by a pipe, perform the **run** function with a zero parameter value instead of the **stop** function.

SYNTAX

<object identifier> **<- run (**<speed value>**)**

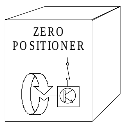
<speed value> : speed value expressed in user length units per second.

EXAMPLES

```
AXI_AnAxis <- run (3.22) ;
AXI_Axes[i] <- run (IRV_p[i]) ;
AXI_Axes[all] <- run (CWV_b * 1.25 - 1) ;
```

7.8 START

APPLIES TO



PURPOSE

This function starts the action of a zero positioner object. The different phases are executed in the following order: coarse phase, fine phase, resolver phase.

SYNTAX

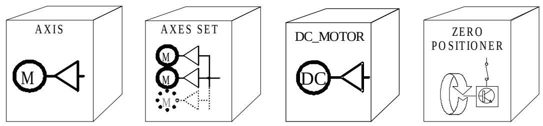
<zero positioner identifier> <**- start** ;

EXAMPLE

```
ZEP_MyZeroPositioner <- start ;
```

7.9 STOP

APPLIES TO



PURPOSE

This function commands an axis, axes set, a DC-motor or a Zero Positioner to perform a controlled stop or stops the action of a zero positioner object. By default, the deceleration used for the stop function is the current value of the axis **DECELERATION**. If a different deceleration is needed, this can be accomplished by modifying **DECELERATION** or the ST1 ASTOP parameter (see document 024.8060).

Both motion generated by a pipe and superimposed motion are stopped and a disconnect is performed on the related pipe.

If a stop is performed by an axis which is member of an axes set and if the axes set is connected to a pipe, a disconnect function will be also performed. The result is that all other axes of the axes set will also be stopped.

SYNTAX

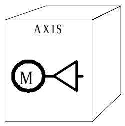
<object identifier> **<- stop**

EXAMPLES

```
AXI_AnAxis <- stop ;
AXI_Axes[i] <- stop ;
AXI_Axes[all] <- stop ;
ZEP_MyZeroPositioner <- stop ;
```

7.10 UPDATE STATUS

APPLIES TO



PURPOSE

This function commands an axis or an axes set to evaluate it's hardware status and to update it's corresponding status information.

SYNTAX

<axis identifier> **<- update_status**

EXAMPLES

```
AXI_AnAxis <- update_status ;
AXI_Axes[i] <- update_status ;
AXI_Axes[all] <- update_status ;
```

8 PIPE BLOCKS EXECUTIVE FUNCTIONS

This chapter describes each of the pipe block executive functions. Note that each of the pipe block functions applies only to the pipe objects listed for the function. These functions are not applicable to physical objects (i.e. axes).

8.1 COMPATIBILITY WITH PREVIOUS SOFTWARE VERSIONS

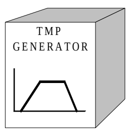
The pipe block executive functions listed in Table 8-1 are no longer available beginning with this version of the PAM software. The actions performed by these deleted functions may now be performed using parameter modification statements (see paragraph 4.3.4.2). The rules and considerations applicable to parameter modifications for each type of pipe block object are found in Chapter 5.

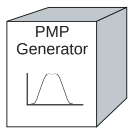
FUNCTION DELETED	AFFECTED OBJECTS
acceleration	PMP, TMP generator
deceleration	PMP, TMP generator
delay_compensation	corrector
reference	comparator
travel_speed	TMP, PMP generator
through_zero_reference	comparator

Table 8-1 Pipe Block Functions no longer available

8.2 ABSOLUTE_MOVE

APPLIES TO





PURPOSE

This function commands a PMP or TMP Generator pipe block to perform a point-to-point motion to an absolute position.

Several cases must be considered:

a) no motion in progress,

b) a point-to-point motion in progress,

c) a continuous motion in progress.

If no motion is in progress when an **absolute_move** is commanded, the object performs a point-to-point motion to the specified absolute position.

If a point-to-point motion is in progress when an **absolute_move** is commanded, the current point-to-point motion is superseded by a new point-to-point motion to the new absolute position.

If a continuous motion is in progress, when an **absolute_move** is commanded, the continuous motion is replaced by a point-to-point motion to the specified absolute position.

For the PMP Generator only, if two parameters are specified the generator perform a forward-backward motion between absolute positions.

> i The **absolute_move** function has slightly different syntax and operation for PMP Generator and TMP Generator; therefore, this function is separately described for each pipe block.

SYNTAX TMP GENERATOR

<tmp generator identifier> **<- absolute_move (**<absolute position value>**)**

<absolute position value> : the position value expressed in user position units.

TMP GENERATOR EXAMPLE

```
CNV_Main << TMP_Main ;
   ...
TMP_Main <- absolute_move (77.256+CRV_DeltaPos) ;
   ...
```

SYNTAX PMP GENERATOR

<pmp generator identifier> **<- absolute_move (**<first absolute position value> , [<second absolute position value>]**)**

<first absolute position value> : the position value of the first destination expressed in user position units.

<second absolute position value> : the position value of the second destination expressed in user position units.

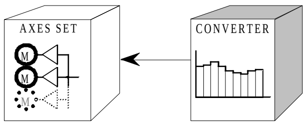
First and second position displacements must be in opposite directions.

PMP GENERATOR EXAMPLE

```
PMP_Example <- absolute_move (12000.0) ;
PMP_Example <- absolute_move (12000.0, 8000.0) ;
```

8.3 CHANGE_ALL_RATIOS

APPLIES TO



PURPOSE

This function changes the ratio used to control the motion of all axes of an axes set. This change is applied through the corresponding Converter pipe block.

SYNTAX

<converter identifier> **<- change_all_ratios (**<ratio value>**)**

<converter identifier> : name of the converter to which the axes set is connected.

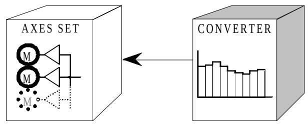
<ratio value> : desired ratio.

EXAMPLE

```
AXES_SET AXS_Table ;
   AXIS = AXI_X ;
   AXIS = AXI_Y ;
END


CONVERTER CNV_Table ;
   DESTINATION = AXS_Table ;
   MODE        = POSITION ;
END

   CNV_Table << CAM_Sinus << TMP_Main ;
   ...
   CNV_Table <- change_all_ratios (CRV_OldRatio * 11.89) ;
   ...
```

8.4 CHANGE_RATIO

APPLIES TO



PURPOSE

This function changes the ratio used to control the motion of a specified axis of an axes set, through the corresponding Converter pipe block.

SYNTAX

<converter identifier> **<- change_ratio (**<axis identifier>**,** <ratio value>**)**

<converter identifier> : name of the converter to which the axes set is connected.

<axis identifier> : name of one axis of the axes set.

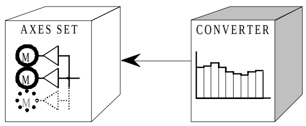
<ratio value> : desired ratio.

EXAMPLE

```
AXES_SET AXS_Table ;
   AXIS = AXI_X ;
   AXIS = AXI_Y ;
END


CONVERTER CNV_Table ;
   DESTINATION = AXS_Table ;
   MODE        = POSITION ;
END

CNV_Table << CAM_Sinus << TMP_Main ;
...
CNV_Table <- change_ratio (AXI_X, CRV_OldRatio * 2.0) ;
...
```

8.5 CONNECT

APPLIES TO



PURPOSE

This function connects a specified axis of an axes set to a specified converter pipe block. The connection is equivalent to closing a logical switch located just before the specified axis. If the current set-points of the converter and the axis are not equal, the axis will execute a jump to the converter's set-point at the maximum torque allowed by the axis parameters. To avoid this jump between set-points, the following rules must be applied:

Converter in POSITION mode	Converter and axis must have same position set-point at connection time.
Converter in SPEED mode	Converter and axis must have same speed set-point at connection time but not necessarily same position.
Converter in TORQUE mode	Converter and axis must have same torque set-point at connection time but not necessarily same position and speed.

SYNTAX

<converter identifier> **<- connect (**<axis identifier>**)**

<converter identifier> : name of the converter to which the axis is connected.

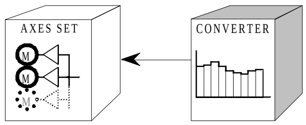
<axis identifier> : name of an axis of the axes set.

EXAMPLE

```
AXES_SET AXS_Table ;
   AXIS = AXI_X ;
   AXIS = AXI_Y ;
END


CONVERTER CNV_Table ;
   DESTINATION = AXS_Table ;
   MODE        = POSITION ;
END
CNV_Table <<- CAM_Sinus << TMP_Main ;
...
CNV_Table <- connect (AXI_Y) ;
...
```

8.6 CONNECT_ALL

APPLIES TO



PURPOSE

This function connects all axes of an axes set to a converter pipe block. The connection is equivalent to closing logical switches located just before all axes of the axes set. If the current set-points of the converter and all axes are not equal, all axes will jump to their new set-points at the maximum torque allowed by their parameters. To avoid this jump between set-points, the following rules must be applied:

Converter in POSITION mode	Converter and axis must have same position set-point at connection time.
Converter in SPEED mode	Converter and axis must have same speed set-point at connection time but not necessarily same position.
Converter in TORQUE mode	Converter and axis must have same torque set-point at connection time but not necessarily same position and speed.

SYNTAX

<converter identifier> **<- connect_all**

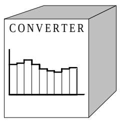
<converter identifier> : name of the converter to which the axes set is connected.

EXAMPLE

```
AXES_SET AXS_Table ;
   AXIS = AXI_X ;
   AXIS = AXI_Y ;
END


CONVERTER CNV_Table ;
   DESTINATION = AXS_Table ;
   MODE        = POSITION ;
END

CNV_Table << CAM_Sinus << TMP_Main ;
...
CNV_Table <- connect_all ;
...
```

8.7 DISACTIVATE

APPLIES TO



PURPOSE

This function deactivates the pipe ending at the specified converter pipe block. Dynamic data related to the pipe and its history is lost. The following rules apply at deactivation time:

Converter in POSITION mode	position stays stable at the last value given by the pipe
Converter in SPEED mode	speed goes to zero
Converter in TORQUE mode	maximum torque parameter value is reactivated

SYNTAX

<converter identifier> **<- disactivate**

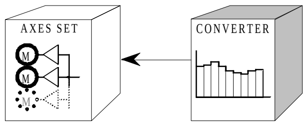
<converter identifier> : name of the converter at which the pipe to be deactivated ends.

EXAMPLE

```
CNV_Main << CAM_Cosinus << TMP_Main ;
...
CNV_Main <- disactivate ;
...
```

8.8 DISCONNECT

APPLIES TO



PURPOSE

This function disconnects a specified axis of an axis set from the specified converter pipe block. The disconnection is equivalent to <u>opening a logical switch</u> located just before the specified axis. The following rules are applied at disconnection time:

Converter in POSITION mode	position stays stable at the last value given by the pipe
Converter in SPEED mode	speed goes to zero
Converter in TORQUE mode	maximum torque parameter value is reactivated

When an axis is disconnected via the **<- disconnect** function, it remains disconnected until one connects it again; <u>even if the converter is disactivated</u>. In previous versions, it was reconnected at converter disactivation.

SYNTAX

<converter identifier> **<- disconnect (**<axis identifier>**)**

<converter identifier> : name of the converter to which the axes set is connected.

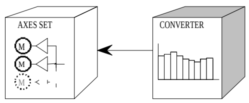
<axis identifier> : name of an axis in the converter's destination axes set.

EXAMPLE

```
AXES_SET AXS_Table ;
   AXIS = AXI_X ;
   AXIS = AXI_Y ;
END


CONVERTER CNV_Table ;
   DESTINATION = AXS_Table ;
   MODE        = POSITION ;
END

CNV_Table << CAM_Cosinus << TMP_Main ;
...
CNV_Table <- disconnect (AXI_X) ;
...
```

8.9 DISCONNECT_ALL

APPLIES TO



PURPOSE

This function disconnects all axes of an axis set from the converter pipe block. The disconnection is equivalent to <u>opening logical switches</u> located just before all axes of the axis set. The following rules are applied at the disconnection time:

Converter in POSITION mode	position stays stable at the last value given by the pipe
Converter in SPEED mode	speed goes to zero
Converter in TORQUE mode	maximum torque parameter value is reactivated

SYNTAX

<converter identifier> **<- disconnect_all**

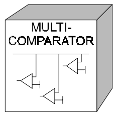
<converter identifier> : name of the converter to which the axes set is connected.

EXAMPLE

```
AXES_SET AXS_Table ;
   AXIS = AXI_X ;
   AXIS = AXI_Y ;
END


CONVERTER CNV_TAble ;
   DESTINATION = AXS_Table ;
   MODE        = POSITION ;
END

CNV_Table << CAM_Cosinus << TMP_Main ;
...
CNV_Table <- disconnect_all ;
...
```

8.10 EXECUTE

APPLIES TO



PURPOSE

Puts the multi-comparator into it's Execute mode.

SYNTAX

<multi-comparator identifier> **<- execute**

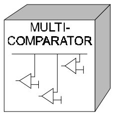
i | The **? execute** inquire function (see paragraph 10.3) may be used to interrogate the current multi-comparator mode.

EXAMPLE

```
MUL_Example <- execute ;
```

8.11 LEARN

APPLIES TO



PURPOSE

Puts the multi-comparator into it’s Learn mode.

SYNTAX

<multi-comparator identifier> **<- learn**

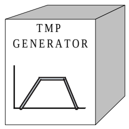
> i The **? execute** inquire function (see paragraph 10.3) may be used to interrogate the current multi-comparator mode.

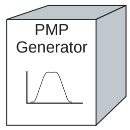
EXAMPLE

```
MUL_Example <- learn ;
```

8.12 POSITION

APPLIES TO





PURPOSE

This function performs the absolute reference position initialisation of a TMP or PMP generator pipe block without making any movement.

SYNTAX

<tmp/pmp generator identifier> **<- position (**<position value>**)**

<position value> : the position value expressed in user length units.

If the current TMP/PMP position is modified while the block is active, a step change to the new current position will occur at the pipe block output.

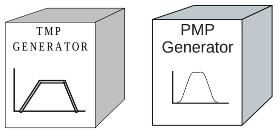
If the TMP/PMP block is not active (not used in any pipe at this time), the new position value will be used to set the initial position at subsequent TMP/PMP activation. Do not use this method for modifying initial position, rather change **INITIAL_POSITION** using a parameter modification command. This behaviour is retained only for compatibility with previous software versions.

EXAMPLE

```
CNV_Main << TMP_Main ;
...
TMP_Main <- position (IRV_NewPosition) ;
...
```

8.13 RELATIVE_MOVE

APPLIES TO



PURPOSE

This function commands a TMP or PMP Generator pipe block to perform a motion to a relative position.

Several cases must be considered:

a) no motion in progress,

b) a point-to-point motion in progress,

c) a continuous motion in progress.

If no motion is in progress when a **relative_move** is commanded, the object performs a point-to-point motion of the specified amount relative to current set-point position.

If a point-to-point motion is in progress when a **relative_move** is commanded, the current point-to-point motion destination is superseded by a new destination equal to current axis set-point plus the specified relative position amount.

If a continuous motion is in progress, when a **relative_move** is commanded, the continuous motion is replaced by a point-to-point motion with destination equal to current axis set-point plus the specified relative position amount.

SYNTAX

<tmp/pmp generator identifier> **<- relative_move (**<relative position value>**)**

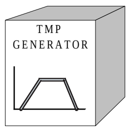
<relative position value> : relative position value expressed in user length units.

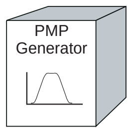
EXAMPLE

```
CNV_Main << TMP_Main ;
...
TMP_Main <- relative_move (CRV_DeltaPosition) ;
...
```

8.14 RUN

APPLIES TO





PURPOSE

This function commands a TMP or PMP Generator pipe block to perform a continuous motion at constant speed.

i | To stop this motion without stopping a superimposed motion, perform the run function with a zero parameter value.

SYNTAX

<tmp/pmp generator identifier> **<- run (**<speed value>**)**

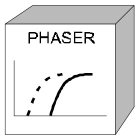
<speed value> : speed value expressed in user length units per second.

EXAMPLE

```
CNV_Main << TMP_Main ;
...
TMP_Main <- run (CRV_HighSpeed) ;
...
```

8.15 START

APPLIES TO



PURPOSE

This function commands a Phaser to start applying a phase value to it's output. When a Phaser is started, it begins computing an output signal but its output remains at **STANDBY_VALUE** until the computed output signal next crosses the **STANDBY_VALUE**. At that point its output value is "connected to" the computed output. When a Phaser (in the stopped condition) receives a **start** command, its **ready** variable is reset to false until it is fully started (its output value is "connected to" the computed output). At that time it becomes true again.

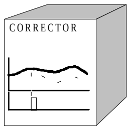
When the Phaser is first used, without having received any command, it is started. To have it stopped, just add the stop command before any pipe activation.

SYNTAX

<phaser identifier> <**- start** ;

8.16 START CORRECTION

APPLIES TO



PURPOSE

This function commands a corrector to execute a correction (enable correction generator)**.** If no parameter is specified, the corrective value computed by the corrector is used. When a parameter is included, the parameter value is used as corrective value. This function is active only when **CORRECTION_MODE** = **ON REQUEST**.

SYNTAX

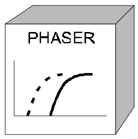
<corrector identifier> <**- start_correction** (corrective value) ;

EXAMPLES

```
COR_Example <- start_correction ;
COR_Example <- start_correction (-180) ;
```

8.17 STOP

APPLIES TO



PURPOSE

This function commands a Phaser to stop applying a phase value to it's output signal. When a Phaser is stopped, it continues computing and outputting it's output value until the output value next crosses the **STANDBY_VALUE**, at which the output value is disconnected from the computed output and connected to the **STANDBY_VALUE.** When a Phaser (in the started condition) receives a **stop** command, its **ready** variable is reset to false until it is fully stopped (its output value becomes **STANDBY_VALUE)**. At that time it becomes true again.

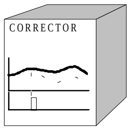
> ⚠ When the Phaser is first used, without having received any command, it is started. To have it stopped, just add the stop command before any pipe activation.

SYNTAX

<phaser identifier> <**- stop** ;

8.18 TRIGGER

APPLIES TO



PURPOSE

This function changes the "must be" value of a corrector pipe block. If the trigger mode of the corrector is ONCE, the trigger is rearmed.

SYNTAX

<corrector identifier> <**- trigger (**<trigger value>**)**

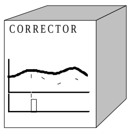
<trigger value> : the new trigger value.

EXAMPLE

```
CNV_PrintUnit << COR_PrintUnit << TMP_Main ;
...
COR_PrintUnit <- trigger (IRV_Level + 12.75) ;
...
```

8.19 TRIGGER_OFF

APPLIES TO



PURPOSE

This function disables the activity of a corrector pipe block.

SYNTAX

<corrector identifier> <**- trigger_off**

EXAMPLE

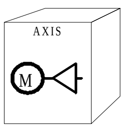
```
CNV_PrintUnit << COR_PrintUnit << TMP_Main
...
COR_PrintUnit <- trigger_off ;
...
```

9 AXES INQUIRE FUNCTIONS

This chapter describes each of the inquire functions available for axes, encoders, zero positioners, and DC motor objects. These physical and application objects are described in Chapters 2 and 3. Note that each of the inquire functions applies only to the objects listed for the function. These functions are not applicable to pipe blocks).

9.1 ERROR

APPLIES TO



PURPOSE

This Boolean inquire function is true when an error condition exists at the specified axis.

SYNTAX

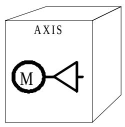
<axis identifier> **? error**

EXAMPLE

```
IF AXI_Main ? error THEN ...
EXCEPTION AXI_Axes[i] ? error SEQUENCE SEQ_AxesErrorHandling[i] ;
```

9.2 ERROR_CODE (BOOLEAN)

APPLIES TO



PURPOSE

This Boolean inquire function is true when the error code parameter matches an existing error condition on a specified axis.

SYNTAX

The statement syntax is as follows:

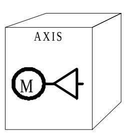
<axis identifier> **? error_code (**<error code>**)**

EXAMPLE

```
IF AXI_Main ? error_code (SERVO_ERROR) THEN ...
CONDITION AXI_Axes[i] ? error_code ...
```

9.3 ERROR_CODE (NUMERICAL)

APPLIES TO



PURPOSE

This inquire function returns the current error code(s) for an axis.

SYNTAX

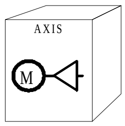
<axis identifier> **? error_code**

EXAMPLE

```
CWV_CrtErrCode <- AXI_Main ? error_code ;
IF AXI_Axes[i] ? error_code = SERVO_ERROR THEN ...
```

9.4 GENERATOR_POSITION

APPLIES TO



PURPOSE

This inquire function returns the current commanded destination position of an axis This position represent only the result of the motions related to the axis executive functions (absolute move, relative move or run) Superimposed pipe motions are not factored into the value returned by this function. The position returned is always a position value maintained internally by PAM, so the execution of this inquire function takes no time.

If an absolute move, relative move or run is in progress, the position information does not represent the real position set-point until this motion is finished. So the inquiring sequence must wait until the end of this motion for an accurate position ("? ready").

SYNTAX

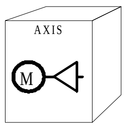
<axis identifier> **? generator_position**

EXAMPLE

```
CRV_CrtPosition <- AXI_Main ? generator_position ;
IF AXI_Axes[i] ? generator_position >= CRV_OldPositions[i] THEN ...
```

9.5 PIPE MOTIONLESS

APPLIES TO



PURPOSE

This boolean inquire function is used to determine when an axis or an axis set connected to a pipe is not in motion. This function becomes true when the last modification to axis (or axis set) position due to pipe motion is received at the axis node(s). As soon as this function becomes true, a driving pipe can be disactivated. Conversely, if a pipe block is disactivated while **? pipe_motionless** is false, axis position at the destination axis (or axes) may not be up to date.

Note that in the following special case:
<u>where</u> a given axis is a member of an axes_set;
<u>and</u> this axes_set is driven by a converter pipe block;
<u>and</u> this axes_set has the optional parameter « **OPTIMIZE = YES** »;
<u>and</u> if the conditions are met to group the given axis with others;
<u>then</u> the function « **? pipe_motionless** » is not available (its value remains true).

SYNTAX

<axis identifier> **? pipe_motionless**

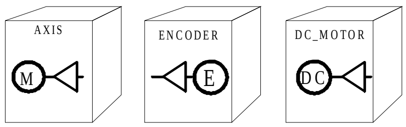
EXAMPLE

```
AXI_
   CNV_Leader   << DIS_Leader   << TMP_Virtual;
   CONDITION    (CNV_Leader ? ready);
   ....
   TMP_Virtual  <- relative_move(180);
   CONDITION TMP_Virtual ? ready;
   CONDITION AXI_Leader ? pipe_motionless ;
   CNV_Leader   <- disactivate;

AXES_SET
   CNV_Leader   << DIS_Leader   << TMP_Virtual;
   CNV_Follower << DIS_Leader   << TMP_Virtual;
   CONDITION  (CNV_Follower ? ready ) * (CNV_Leader ? ready);
          ...
       TMP_Virtual  <- relative_move(180);
       CONDITION TMP_Virtual ? ready;
       CONDITION SET_AllAxis ? pipe_motionless ;
```

```
CNV_Follower <- disactivate;
CNV_Leader   <- disactivate;
```

9.6 POSITION

APPLIES TO



PURPOSE

This inquire function returns the current (instantaneous) position set-point of an axis or DC-motor. For an encoder, the current value of the ST1 variable/parameter specified by the encoder **ADDRESS** parameter is returned.

For an axis:

- This is always a position set-point generated or managed internally by PAM, so the execution of this inquire function takes no time.
- The pipe's position set-point is returned when a pipe is generating set-points for the axis
- If a superimposed absolute move, relative move or run is in progress, the position information does not represent the real position set-point until the superimposed motion is finished. So the inquiring sequence must wait until the end of the superimposed motion to have an accurate position ("? ready"). The position at that point is the sum of the pipe's current position set-point with the superimposed motion position.

For an encoder:

- This inquire function returns the current value of the specified ST1 variable/parameter. If the encoder is connected to a sampler pipe block and if the corresponding pipe is active, the ST1 variable/parameter specified in the **ADDREESS** parameter is sampled continuously by PAM, so execution of this inquire function takes no time. Otherwise, PAM must perform a read cycle on the ST1 which takes six to seven × **BASIC_PAM_CYCLES** to complete. During this interval the sequence is suspended (see paragraph 3.10.7.1).

For a DC-motor:

- PAM must ask the Smart-IO for it’s DC-motor position through the PAM-Ring which takes some time to be executed, so the position is received after a delay.

SYNTAX

<axis identifier> **? position**

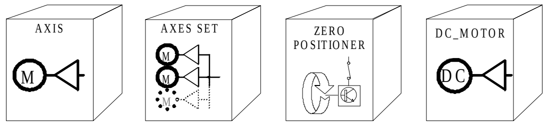
For an encoder, **? position** and **? value** are alternative syntaxes for the same function.

EXAMPLE

```
CRV_CrtPosition <- AXI_Main ? position ;
IF AXI_Axes[i] ? position >= CRV_OldPositions[i] THEN ...
```

9.7 READY

APPLIES TO



PURPOSE

This Boolean inquire function is true when an axis, an encoder, a zero-positioner or a DC-motor object has completed execution of it's last command(s).

SYNTAX

<object identifier> **? ready**

EXAMPLE

```
IF AXI_Main ? ready THEN ...
CONDITION AXI_Axes[i] ? ready  ;
CONDITION AXI_Axes[all*] ? ready ;
```

9.7.1 READY RULES FOR AXES

For commands or combination of commands executed by axes or axis sets, it is necessary to detail the process. There are two families of commands:

- Logical commands which include **connect**, **disconnect**, **change_ratio** and all inquire functions (statements with ?).
- physical commands which include **power_on**, **power_off**, **absolute_move**, **relative_move**, **run** and **stop**.

The rules for "<axis identifier> **? ready"** flag response are:

- logical commands do not affect the flag;
- physical commands set the flag to false during execution and produce a true setting upon completion of execution;
- when a logical command is superimposed on a physical command, the flag is not affected;
- when multiple physical commands are executing simultaneously, the flag is manipulated with respect to the resulting combination of commands.
- no command can be superimposed with a logical command because logical commands require almost no time for execution.

The commands which are tested for completion of using the inquire function "**? ready**" are:

absolute_move
relative_move
run
start
stop
power_on

EXAMPLE

In this example Led1 is set as soon as AXI_X starts its movement.

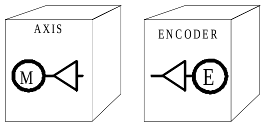
```
AXI_X <- relative_move (1000);
SBI_Led1  <- set;
```

In this example Led1 is set only when AXI_X has completed its movement.

```
AXI_X <- relative_move (1000);
CONDITION AXI_X ? ready;
SBI_Led1  <- set;
```

9.8 SPEED

APPLIES TO



PURPOSE

This inquire function returns the current speed set-point of an axis. For an encoder, the derivative of the specified ST1 variable/parameter is returned.

For an axis:

- This is always the speed set-point generated by a pipe and managed internally by PAM, so execution of this inquire function takes no time. The speed generated by a superimposed motion is not taken into account.

For an encoder:

- If the encoder is connected to a sampler pipe block and if the corresponding pipe is active, the ST1 variable/parameter specified in the **ADDREESS** parameter is sampled continuously by PAM, so execution of this inquire function takes no time.
- Otherwise, PAM must perform a read cycle on the ST1 which takes six to seven × **BASIC_PAM_CYCLE** to complete. During this interval the sequence is suspended (see paragraph 3.10.7.1).

SYNTAX

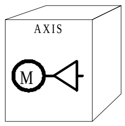
<object identifier> **? speed**

EXAMPLE

```
CRV_CrtSpeed <- AXI_Main ? speed ;
IF AXI_Axes[i] ? speed >= CRV_OldSpeeds[i]  THEN ...
```

9.9 STATUS (BOOLEAN)

APPLIES TO



PURPOSE

This Boolean inquire function is true if the parameter status code matches the status of the object.

SYNTAX

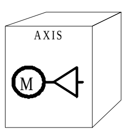
<object identifier> **? status (**<status code>**)**

EXAMPLE

```
IF AXI_Main ? status (POWER_OFF) THEN ...
EXCEPTION !(AXI_Axes[i] ?status (AXIS_MOVING)) SEQUENCE ...
```

9.10 STATUS (NUMERICAL)

APPLIES TO



PURPOSE

This inquire function returns the status code of an axis or an axis set.

SYNTAX

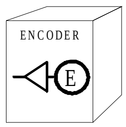
<object identifier> **? status**

EXAMPLE

```
CWV_CrtStatus_code <- AXI_Main ? status ;
IF AXI_Axes[i] ? status = POWER_OFF  THEN ...
```

9.11 VALUE

APPLIES TO



PURPOSE

This inquire function returns the current value of the specified ST1 variable/parameter. If the encoder is connected to a sampler pipe block and if the corresponding pipe is active, the ST1 variable/parameter specified in the **ADDREESS** parameter is sampled continuously by PAM, so execution of this inquire function takes no time. Otherwise, PAM must perform a read cycle on the ST1 which takes six to seven × **BASIC_PAM_CYCLE** to complete. During this interval the sequence is suspended (see paragraph 3.10.7.1).

SYNTAX

<encoder identifier> **? value**

> For an encoder, **? value** and **? position** are alternative syntaxes for the same function.

EXAMPLE

IWV_TempAxis1 is updated to the current value of the ST1 parameter/variable specified by the **ADDRESS** parameter of encoder ENC_Blanket.

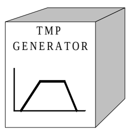
```
IWV_TempAxis1 <- ENC_Blanket ? value ;
```

10 PIPE BLOCKS INQUIRE FUNCTIONS

This chapter describes each of the pipe block inquire functions. Note that each of the pipe block functions applies only to the pipe objects listed for the function. These functions are not applicable to physical objects (i.e. axes).

10.1 ACCELERATION

APPLIES TO



PURPOSE

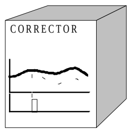
This inquire function returns the current acceleration value used by the TMP generator.

SYNTAX

<pmp/tmp generator identifier> **? acceleration**

10.2 CORRECTION

APPLIES TO



PURPOSE

This function returns the corrective value (including sign) currently in use by the corrector. The returned value must be interpreted differently depending on the setting of **CORRECTION_REFERENCE.** Note that if a corrective value was last specified via a **start_correction** command, this corrective value is returned.

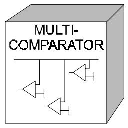
Correction is valid only after the corrector is triggered and before the correction is started.

SYNTAX

<corrector identifier> **? correction**

10.3 EXECUTE

APPLIES TO



PURPOSE

This function is used to test the multi-comparator's operating mode. The **? execute** function returns a boolean **TRUE** if the multi-comparator is in execute mode, and **FALSE** if the multi-comparator is in learn mode.

SYNTAX

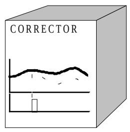
<multi-comparitor identifier> **? execute**

Example

```
IF (MUL_Example ? execute) THEN
```

10.4 LATCHED_DD_VALUE

APPLIES TO



PURPOSE

This function returns the second derivative of the "is" value, latched at the beginning of a corrector pipe block correction cycle.

SYNTAX

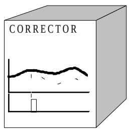
<corrector identifier> ? **latched_dd_value**

EXAMPLE

```
...
CNV_PrintUnit << COR_PrintUnit << TMP_Main ;
...
IRV_DDValue <- COR_PrintUnit ? latched_dd_value ;
...
```

10.5 LATCHED_D_VALUE

APPLIES TO



PURPOSE

This function returns the derivative of the "is" value, latched at the beginning of a corrector pipe block correction cycle.

SYNTAX

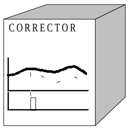
<corrector identifier> ? **latched_d_value**

EXAMPLE

```
...
CNV_PrintUnit << COR_PrintUnit << TMP_Main ;
...
IRV_DValue <- COR_PrintUnit ? latched_d_value ;
...
```

10.6 LATCHED_VALUE

APPLIES TO



PURPOSE

This function returns the "is" value, latched at the beginning of a corrector pipe block correction cycle.

SYNTAX

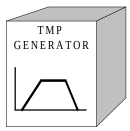
<corrector identifier> ? **latched_value**

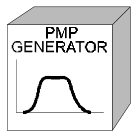
EXAMPLE

```
...
CNV_PrintUnit << COR_PrintUnit << TMP_Main ;
...
IRV_Value <- COR_PrintUnit ? latched_value ;
...
```

10.7 POSITION

APPLIES TO





PURPOSE

This inquire function returns the current (instantaneous) position set-point generated by a trapezoidal motion profile generator pipe block. If the pipe block is not active, zero is returned.

SYNTAX

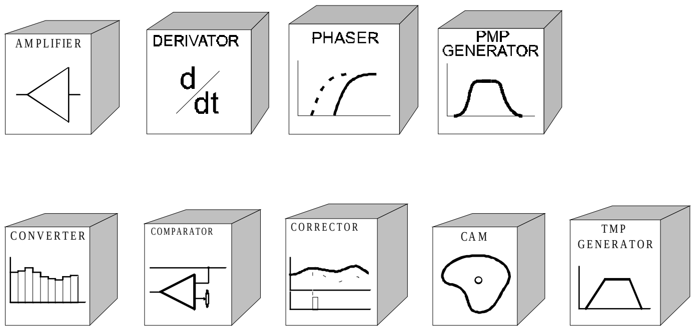
<tmp generator identifier> **? position**

EXAMPLE

```
CNV_Main << TMP_Main ;
...
IRV_CrtPosition <- TMP_Main ? position ;
...
```

10.8 READY

APPLIES TO



PURPOSE

This boolean inquire function is used for checking the ready status for the applicable pipe block types. In general, **ready** is true when a pipe block has completed execution of the last command(s). However, the exact meaning of **ready** varies depending on the pipe block type. Refer to the pipe block descriptions (see chapter 5) for details.

SYNTAX

<pipe block identifier> **? ready**

The following PMP/TMP generator motion functions may be tested for completion of execution using **? ready**:

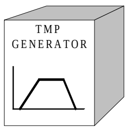
absolute_move
relative_move
run

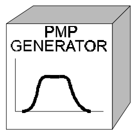
EXAMPLE

```
CNV_Conveyor << CMP_Conveyor << TMP_Main;
...
CONDITION CNV_Conveyor ? ready ;
...
CMP_Conveyor <- through_zero_reference (1256.89) ;
...
CONDITION CMP_Conveyor ? ready ;
...
CMP_Conveyor <- reference (34.75 * CRV_OldrRef) ;
...
CONDITION CMP_Conveyor ? ready ;
...
```

10.9 SPEED

APPLIES TO





PURPOSE

This inquire function returns the current (instantaneous) speed set-point generated by a TMP or PMP generator pipe block.

SYNTAX

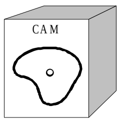
<TMP/PMP generator identifier> **? speed**

EXAMPLE

```
CNV_Main << TMP_Main ;
...
IRV_CrtSpeed <- TMP_Main ? speed ;
...
```

10.10 STATUS (BOOLEAN)

APPLIES TO



PURPOSE

This boolean inquire function is true if the parameter status code matches the status of the cam pipe block.

NOT YET IMPLEMENTED

SYNTAX

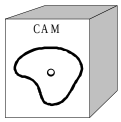
<cam identifier> **? status (**<status code>**)**

EXAMPLE

```
CNV_Main << CAM_Sinus << TMP_Main ;
...
CONDITION CAM_Sinus ? status (SATURATED) ;
```

10.11 STATUS (NUMERICAL)

APPLIES TO



PURPOSE

This inquire function returns the status code of a cam pipe block.

NOT YET IMPLEMENTED

SYNTAX

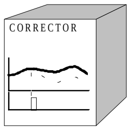
<cam identifier> **? status**

EXAMPLE

```
CNV_Main << CAM_Sinus << TMP_Main ;
...
IWV_CamStatus <- CAM_Sinus ? status ;
```

10.12 TRIGGERED

APPLIES TO



PURPOSE

This function asks a corrector pipe block if a correction is pending. Refer to the Corrector description (paragraph 5.9) for details on the “triggered” status indicator.

SYNTAX

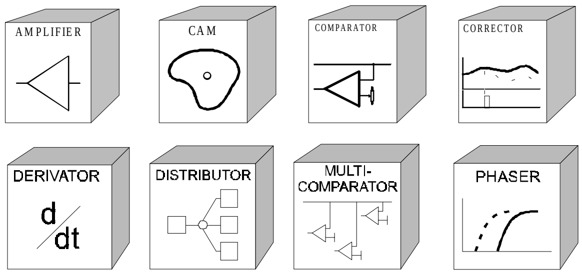
<corrector identifier> ? **triggered**

EXAMPLE

```
CNV_PrintUnit << COR_PrintUnit << TMP_Main ;
...
CONDITION   COR_PrintUnit ? triggered ;
...
```

10.13 VALUE

APPLIES TO



PURPOSE

This function returns the current numerical value coming out of the pipe block.

SYNTAX

<pipe block identifier> ? **value**

EXAMPLE

```
CRV_AmpOuput <- AMP_Example ? value ;
```

11 DISPLAY FUNCTIONS

This chapter describes those functions used with LEDs, seven segment displays and the eight digit display located on the PAM.

11.1 BLINK

This function, applicable to the LED and seven segment display outputs only, enables automatic blinking of the output (performed at the node level). The blinking frequency is not user configurable.

SYNTAX

<identifier> <**- blink** ;

< identifier> : name of an LED output or display output.

EXAMPLE

```
...
D7S_MyDisplay <- blink ;   // blink the current value on the display.
LED_MyLed     <- blink ;   // made the led blink.
...
```

11.2 DISPLAY

This function sends to the seven segment display output a character string or value to be displayed.

11.2.1 DISPLAY OF CHARACTER STRING

SYNTAX

<display identifier> <**- display("**<string>**")** ;

<display identifier> : name of a display output.

<*string>* **:** *string can be composed of any character of the selected font. Only the HEXA font is available (refer to Appendix C : Display output table). With the display output on a SMART_IO node, the string is limited to 1 character.*

EXAMPLE

```
...
D7S_MyDisp <- display("P") ;    // display a P.
...
```

11.2.2 DISPLAY OF VALUE

SYNTAX

<display identifier> <**- display(<**value>**,** <digit number>**)**;

<display identifier> : the name of the display output.

<digit number> *: the number of digits to display, (only 1 digit is possible with Smart_IO node display).*

<*value> : an integer expression, integer object or integer constant. The value displayed is the value given in the statement, truncated to the size specified by <digit number>. With 1 digit it is possible to display values ranging from 0 to #F.*

EXAMPLES

```
...
D7S_MyDisp <- display(7,1) ;    // display the value 7.
...
CWV_Test1<- 8000 ;
D7S_MyDisp <- display(CWV_Test1 / 1000, 1) ;// display 8.
...
CWV_Test1<- 4660 ;
D7S_MyDisp <- display(CWV_Test1, 1) ;
// display 4, because 4660 truncated to 1 digit, gives 4.
```

11.3 NO_BLINK

This function, applicable only to LED and seven segment display outputs, disables automatic blinking of the output (performed at the node level). When a **no_blink** statement is executed, the output, if set, reverts back to the "steady on" condition.

SYNTAX:

<identifier> <- **no_blink** ;

< identifier> : name of an led output or seven segment display output.

EXAMPLE

```
...
LED_MyLed <- set ;        // turn on "LED_MyLed".
LED_MyLed <- blink ;      // blink "LED_MyLed".
WAIT_TIME 5000 ;
LED_MyLed <- no_blink           // stop blinking and restore on state
```

11.4 INVERT

This function, applicable to all Boolean output objects, applies the Boolean **not** function to the object.

SYNTAX

<identifier> <- **invert** ;

EXAMPLE

```
...
// binary output blink loop
LOOP
   SBO_8 <- invert ; // invert output state.
   WAIT_TIME 500 ;
END_LOOP BOL_StopKey ;
...
```

11.5 RESET

This function, applicable to all Boolean output objects, forces the object to the Boolean value "false". "False" corresponds to 0 (led off ,output inactive).

SYNTAX

<identifier> <**- reset** ;

EXAMPLE

```
...
SBO_7[all] <- reset ;     // reset all item of SBO_7.
LED_2[i]         <- reset ;     // reset item i of LED_2 (led off).
...
```

11.6 SET

This function, applicable to all Boolean output objects, forces the object to the Boolean value "true". "True" corresponds to 1 (led on, output active).

SYNTAX

<identifier> <**- set** ;

EXAMPLE

```
...
SBO_BinaryOut7[all] <- set ;    // set all item of SBO_7.
LED_2[i]          <- set ;// item i of LED_2 turned true (on)
...
```

11.7 PAM DISPLAY

11.7.1 INTRODUCTION

This eight digit hexadecimal display called the PamDisplay is located on the PAM front panel. The PamDisplay is accessible to the application. It can be used for application level messages and value monitoring. The PamDisplay object is pre-declared, so it need not be declared by the application.

A set of PamDisplay functions (see paragraph 11.8) used by the application to place messages on the PamDisplay provide the following capabilities:

- Send a general status message to the PamDisplay and to the PAM Debugger.
- Send a warning message to the PamDisplay and to the PAM Debugger.
- Send an end of warning message to the PamDisplay and to the PAM Debugger.
- Send an error message to the PamDisplay and to the PAM Debugger.
- Send an end of error message to the PamDisplay and to the PAM Debugger.
- Display value of a monitored variable on the PamDisplay.

> i | PAM also sends system-generated warning, error and status messages to the PamDisplay.

GENERAL SYNTAX

PAMDISPLAY <- <statement type> **(**<message number>**,** <character string>**,** <parameter expression>**) ;**

<statement type> : ***MESSAGE*** *|* ***WARNING*** *|* ***END_WARNING*** *|****ERROR*** *|****END_ERROR****. The statement type is encoded into the "T" character position of the PAM message code (see paragraph 11.8).*

<message number> : three digit hexadecimal code number assigned to the message. This number is displayed in the "N" character positions of the PAM message code (see paragraph 11.7.2), and on the PAM debugger.

<character string> : the message string. This is the message text. This string is displayed only on the PAM debugger's message window and must be not more than 40 characters long. <character string> must be enclosed in quotation marks.

<parameter expression> : the message parameter. This parameter contains some dynamic value related to the message. This value is displayed both on the PAM display (in the data fields) and the PAM debugger.

Value monitoring is performed by assigning an expression (any application variable) to the PamDisplay object. Then the variable (current value) is periodically displayed on the PamDisplay.

i The contents of all PamDisplay functions (except monitoring) is duplicated on the PAM debugger's message window.

11.7.2 PAM DISPLAY MESSAGE CODES

The 8 hexadecimal digits of PAM message codes displayed on the Pam Display are encoded as follows:

O	O	R	X	T	N	N	N

where:

OO indicates the origin of the message:
00 = application messages
All other origin codes are system messages and errors and are listed in appendixes

R gives additional information:
0 = means non real time message
8 = real time message

X not used (its value is 0)

T indicates the type of message:
M = message
W = warning message
K = end of warning (stored in the error list)
F = fatal error (stored in the error list)
E = error message (stored in the error list)
S = end of error (stored in the error list).

NNN indicates the message code number (in hexadecimal)

11.8 PAMDISPLAY FUNCTIONS

11.8.1 MESSAGE

Message functions generally are used to indicates progression of the application, or some normal system state related to the application.

SYNTAX

PAMDISPLAY <- message (<message number >**,** <character string>**,** <parameter expression>**) ;**

<message number> : three digit hexadecimal code number assigned to the message.

<character string> : the message text. This string is displayed only in the PAM debugger's message window and must be not more than 40 characters long.

<parameter expression> : the message parameter. This parameter contains some dynamic value related to the message.

EXAMPLE

```
...
#define  DEF_TracePointNb  12    // Debug trace point reached
...
IWV_TracePoint <- 33 ;     // Some operation done
...
PamDisplay <- message (DEF_TracePointNb,
                       "Reached trace point No:",
                       TracePoint) ;
...
```

Following execution of this example, the message code on the Pam Display is “0080M00C”.

11.8.2 WARNING

A warning (message) is generally used to indicate a non fatal error, or some unusual system state related to the application.

SYNTAX

PAMDISPLAY <- warning (<number expression>**,** <character string>**,** <parameter expression>**) ;**

<message number> : three digit hexadecimal code number assigned to the warning.

<character string> : the warning message text. This string is displayed only in the PAM debugger's message window and must be not more than 40 characters long.

<parameter expression> : the warning parameter. This parameter contains some dynamic value related to the warning.

EXAMPLE

A monitored axis has stopped.

```
...
#define  DEF_ZeroValue     23
...
IF IRV_Speed = 0.0 THEN
   CWV_Location <- 127 ;
   PamDisplay <- warning (DEF_ZeroValue,
                          "Speed is nul at location:",
                          CWV_Location) ;
END_IF
...
```

11.8.3 END_WARNING

An end_warning (message) is generally used to indicate that the condition for which a warning (message) was generated no longer exists.

SYNTAX

PAMDISPLAY <- end_warning (<number expression>**,** <character string>**,** <parameter expression>**) ;**

<message number> : three digit hexadecimal code number assigned to the end_warning. This must be the same code number assigned to the corresponding warning.

<character string> : the end_warning message text. This string is displayed only in the PAM debugger's message window and must be not more than 40 characters long.

<parameter expression> : the end_warning parameter. This parameter contains some dynamic value related to the end_warning.

EXAMPLE

Speed now back into allowable range; therefore, cancel warning.

```
IF IRV_Speed > 10.0 THEN
      CWV_Location <- 127 ;
      PamDisplay <- end_warning (DEF_ZeroValue,
                             "Speed is okay at location:",
                             CWV_Location) ;
   END_IF
   ...
```

11.8.3.1 HANDLING OF END_WARNING

If a warning is displayed, when an **END_WARNING** message is generated, the following sequence occurs:

- A "warning cancelled " message is displayed on the PAMDEBUGGER screen
- The PAM display scrolls the application name and date after a delay of 2 seconds.

11.8.4 ERROR

An error (message) indicates a fatal error, a non fatal error or an erroneous system state related to the application.

SYNTAX

PAMDISPLAY <- error (<number expression>**,** <character string>**,** <parameter expression>**) ;**

<message number> : three digit hexadecimal code number assigned to the error.

<character string> : the error message text. This string is displayed only in the PAM debugger's message window and must be not more than 40 characters long.

<parameter expression> : the message parameter. This parameter contains some dynamic value related to the error.

EXAMPLE

```
...
#define  DEF_ZeroDiv 6
...
IF IRV_Ratio = 0.0 THEN
   CWV_Location <- 56 ;
   PamDisplay <- error (DEF_ZeroDiv,
                        "Division by zero at location:",
                        CWV_Location) ;
   IRV_Ratio <- 1.0 ; // Dummy value
END_IF
   IRV_Position <- IRV_Position / IRV_Ratio ;
...
```

11.8.5 END_ERROR

An end_error (message) is generally used to indicate the end of a non fatal error.

SYNTAX

PAMDISPLAY <- end_error (<number expression>**,** <character string>**,** <parameter expression>**) ;**

<message number> : three digit hexadecimal code number assigned to the end_error. This must be the same code number assigned to the corresponding error

<character string> : the end_error message text. This string is displayed only in the PAM debugger's message window and must be not more than 40 characters long.

<parameter expression> : the end_error parameter. This parameter contains some dynamic value related to the end_error.

EXAMPLE

```
...
#define     DEF_EndShortCircuit          7
...
PamDisplay <- end_error (DEF_EndShortCircuit,
                         "CC end on output No:",
                         IWV_OutputNumber) ;
...
```

11.8.5.1 HANDLING OF END_ERROR

If a currently displayed error is a non fatal error, the following sequence occurs when an **END_ERROR** message with the same error code is generated:

- An "error cancelled " message is displayed on the PAMDEBUGGER screen.
- The PAM display scrolls the application name and date.

EXAMPLE

If PAM is started with an application containing an RS422 Port declaration without running host communication software, the following error is displayed on the PAM debugger screen :

```
DD.MM.YY  HH:MM:SS [0780E039] real time Error
ges_rs422   RS422 : comm. disconnected
```

The PAM display blinks : 0780E039

After the host communication software is started, the following message is displayed on the PAM debugger screen :

```
DD.MM.YY  HH:MM:SS [0780S039] real time Error Canceled
ges_rs422   RS422 : comm. disconnected
```

The PAM display now scrolls the application name and version.

11.8.6 MONITORING

The PamDisplay can be used to monitor any application value located in any type of object. The display uses floating point format.

SYNTAX

PAMDISPLAY <- <expression> **;**

<expression> : the expression value to monitor. The expression must be real (in floating point format).

EXAMPLE

```
...
PamDisplay <- 3453.56 * IRV_SomeOtherVariable ;
PamDisplay <- AXI_Roller[i] ;
...
```

12 ERROR AND STATUS FUNCTIONS

This chapter describes error and status management functions, the fatal errors panel and tuning of sequence workspaces.

12.1 ERROR FUNCTIONS

Two inquire functions, **? error** (see paragraph 12.1.2) and **? error_code** (see paragraph 12.1.1), are used to test and monitor error status on axis and DC motor objects. The error status of a Smart IO peripheral is also accessed using these same error functions. The syntax and use of these functions varies somewhat depending on the object; therefore, these error functions are described separately for each of the applicable object types.

12.1.1 ERROR_CODE

12.1.1.1 FOR AN ST1 NODE

The **error_code** inquire function is not available for ST1 nodes.

12.1.1.2 FOR AN AXIS

Two forms of the **error_code** function are available to return either the current axis numeric error code or a boolean value after filtering through a mask parameter.

SYNTAX (NUMERIC FUNCTION)

<axis identifier> **? error_code** ;

This statement returns a 32 bit value comprised of error bits corresponding to the status ABCD of the ST1 servo amplifier (see document # 024.8068) masked by the value 0xd0f8ffff with the part C+D of the ST1 status already masked by ST1 parameter CMASKS.

SYNTAX (BOOLEAN FUNCTION)

<axis identifier> **? error_code** (<error code mask identifier>)

<error code mask identifier> : mask applied to the axis error code. A set of standard masks (see Table 12-1) are defined in the file ST1ERROR.SYS.

This expression returns a Boolean value based on a logical AND of axis error code with the specified mask.

MASK VALUE	MASK NAME	ERROR IF **ERROR_CODE** = TRUE (ST1 STATUS BIT REFERENCE)
0x80000000	POWER_STAGE_DISABLED	power stage is disabled (bit 7 status A)
0x40000000	CHANGE_IN_CD	change in status C+D
0x10000000	SERVO_LAG_ERROR	lag error (bit 4 status A)
hardware errors		
0x00800000	RESOLVER_FAILURE	resolver failure (bit 7 status B)
0x00400000	OVER_VOLTAGE	over voltage on UA (bit 6 status B)
0x00200000	POWER_STAGE_OVERLOAD	power stage overloaded (bit 5 status B)
0x00100000	INT_SUPPLY_FAILURE	internal supply failure (bit 3 status B)
0x00080000	AUX_SUPPLY_FAILURE	auxiliary supply failure (bit 3 status B)
0x00001000	COMMUNICATION_FAILURE	communication failure (bit 4 status C)
other		
0x00000080	MOTOR_TEMPERATURE	motor over temperature (bit 7 status D)
0x00000040	CURRENT_LIMITED	output current limited (bit 6 status D)
0x00000010	OUT_OF_LIMITS	out of limits (bit 4 status D)
0x00000002	SERVO_RESET	servo reset is over (bit 1 status D)

Table 12-1 Axis Error Code Masks

\What is the significance of "x" in the above mask values??\

EXAMPLE

Testing for a servo lag error:

```
IF MyAxis ? error_code(SERVO_LAG_ERROR)
   THEN  ...
   ...        treatment for servo lag error
```

12.1.1.3 DEFINING SPECIAL MASKS

The user may define a global mask for a particular group of error conditions by including the mask definition in ST1ERROR.SYS.

EXAMPLE

(Definition included in ST1ERROR.SYS) :

```
#set HARD_ERROR     RESOLVER_FAILURE | OVER_VOLTAGE |\
         POWER_STAGE_OVERLOAD | INT_SUPPLY_FAILURE |\
         AUX_SUPPLY_FAILURE | COMMUNICATION_FAILURE
```

An application-specific mask may also be defined in an application (after including ST1ERROR.SYS).

EXAMPLE

(Definition included in application file)

```
#set SUPPLY_ERROR  INT_SUPPLY_FAILURE |AUX_SUPPLY_FAILURE
```

12.1.1.4 FOR A SMART_IO NODE

Two forms of the **error_code** function are available to return either the current Smart_IO numeric error code, or a boolean value after filtering through a mask parameter.

SYNTAX (NUMERIC FUNCTION)

<smart_io node name> **? error_code**

This function returns a 32 bit value which is a collection of error bits representing the Smart_IO error status.

SYNTAX (BOOLEAN FUNCTION)

<smart_io node name> **? error_code** (<error code mask identifier>) ;

This expression return a Boolean value based on a logical AND of the Smart_IO error code with the specified mask.

<error code mask identifier> : mask applied to the Smart_IO error code. A set of standard masks (see Table 12-2) are defined in the file SMARTERR.SYS.

MASK VALUE	SMART CODE	MASK NAME	ERROR IF **ERROR_CODE** = TRUE
0x000001	0x01	HEAP_OVERFLOW	heap memory overflow
0x000002	0x02	FIFOS_ERRORS	one command fifo is full
0x000004	0x03	CRC_ERROR	CRC error on received frame
0x000008	0x04	FRAME_TYPE	bad frame type received
0x000010	0x06	OUT_OVERTEMP_1	heat sink over temperature on module 1
0x000020	0x06	OUT_OVERTEMP_2	heat sink over temperature on module 2
0x000040	0x06	OUT_OVERTEMP_3	heat sink over temperature on module 3
0x000080	0x06	OUT_OVERTEMP_4	heat sink over temperature on module 4
0x000100	0x06	OUT_OVERTEMP_5	heat sink over temperature on module 5
0x000200	0x06	OUT_OVERTEMP_6	heat sink over temperature on module 6
0x000400	0x06	OUT_OVERTEMP_7	heat sink over temperature on module 7
0x000800	0x06	OUT_OVERTEMP_8	heat sink over temperature on module 8
0x001000	0x08	AIR_OVERTEMP	ambient air over temperature
0x002000	0x0F	DC_MOTORS_POSITION _LOST	position lost for one or more of the DC Motors of this node
0x004000	0X10	DC_MOTORS_UNEXPEC TED_MOVE	unexpected rotation for one or more of the DC Motors of this node
	0x11	DC_MOTORS_DO_NOT_ STOP	rotation keep going after stop for one or more of the DC Motors of this node
0x010000	0x09	UNKNOWN_CMD	unknown command received
0x020000	0x0A	UNALLOWED_CMD	unallowed command received
0x040000	0x0B	UNDECLARED_PERIPH	request on undeclared I/O
0x080000	0x0C	UNKNOW_PERIPH	request on unknown I/O type

Table 12-2 Smart_IO Error Code Masks

EXAMPLE

```
IF MySmartIoNode ? error_code(AIR_OVERTEMP)
      THEN    ...
         ...        treatment for overtemperature
```

12.1.1.5 DEFINING SPECIAL MASKS

The user may define a global mask for a group of error conditions by including the mask definition in SMARTERR.SYS.

EXAMPLE

(Definition included in SMARTERR.SYS)

```
#set OUT_OVERTEMP OUT_OVERTEMP_1 | OUT_OVERTEMP_2
```

A mask definition may be redefined in an application (after including SMARTERR.SYS).

EXAMPLE

(Definition redefined in application file)

```
#undef OUT_OVERTEMP
#set OUT_OVERTEMP  OUT_OVERTEMP_1 | OUT_OVERTEMP_2 | OUT_OVERTEMT_3
```

i | To redefine a mask use #undef <mask name> before #set <mask name>.

12.1.1.6 BINARY OUTPUT ON SMART_IO NODE

There are no error bits in the **SMART_IO** node error code for binary outputs; but in the event of a short circuit in a binary output, an error message, [0400000E], is sent to the PAM display. The short circuit information defines the output group (group 1 : OUT 1..4, group 2 : OUT 5..8, group 3 : OUT 9..12) in which the short was detected. Refer to Appendix B, error [0400000E].

12.1.1.7 FOR A DC MOTOR

Two forms of the **error_code** function are available to return either the current DC Motor numeric error code, or a boolean value, after filtering through a mask parameter.

SYNTAX (NUMERIC FUNCTION)

<DC Motor identifier> **? error_code** ;

This function returns a 32 bit error code representing DC Motor error status.

SYNTAX (BOOLEAN FUNCTION)

<DC Motor identifier> **? error_code** (<error code mask identifier>) ;

< error code mask identifier> mask applied to the DC Motor error code. A set of standard masks (see Table 12-3) are defined in the file SMARTERR.SYS.

This function returns a Boolean value based on a logical AND of the DC Motor error code with the specified mask.

EXAMPLE

```
IF MyDcMotor ? error_code(DC_MOTORS_POSITION_LOST)
   THEN  ...
      ...           traitement for position lost
```

MASK VALUE	SMART CODE	MASK NAME	ERROR IF ERROR_CODE = TRUE
0x000001	0x07	DCM_TIMOUT	counting time-out
0x000002	0x0D	DCM_LOWER_LIMIT	lower limit reached
0x000004	0x0E	DCM_UPPER_LIMIT	lower limit reached
0x000008	0x12	DCM_INQ_POS_TIMOUT	inquire position answer time-out
0x002000	0x0F	DC_MOTORS_POSITION_LOST	position lost for one or more of the DC Motors of this node
0x004000	0X10	DC_MOTORS_UNEXPECTED_MOVE	unexpected rotation for one or more of the DC Motors of this node
	0x11	DC_MOTORS_DO_NOT_STOP	rotation keep going after stop for one or more of the DC Motors of this node

Table 12-3   DC Motor Error Code Masks

12.1.1.8 DEFINING SPECIAL MASKS

The user may also define a global mask for a group of DC Motor error conditions using the procedure described in paragraph 12.1.1.5.

EXAMPLE

Definition included in SMARTERR.SYS) :

```
    #set DCMOTOR_ABORT_ERROR DCM_TIMOUT | DCM_LOWER_LIMIT |
DCM_UPPER_LIMIT
```

A mask definition specific to an application may also be defined (after including SMARTERR.SYS).

EXAMPLE

(Definition added in application file) :

```
#undef OUT_OVERTEMP
#set OUT_OVERTEMP  OUT_OVERTEMP_1 | OUT_OVERTEMP_2 | OUT_OVERTEMT_3
```

i | All DC Motor errors are reported into the error code of the corresponding Smart IO.

12.1.2 ERROR

The **error** function returns a boolean true if one or more bits in the object error code is set; otherwise, the result is false. **Error** is available for axis and DC Motor objects and the Smart_IO peripheral.

> stop | The **error** function not available for ST1 nodes

SYNTAX

<object identifier> ? error ;

<object identifier> : name of a Smart_IO, an axis or DC Motor.

12.2 STATUS FUNCTION

The **status** inquire function returns the motion status of an axis object. Two alternate forms of the **status** function syntax are defined in paragraphs 12.2.1 and 12.2.2.

The function <object identifier> ? status is available only for axis objects.

12.2.1 STATUS (NUMERIC FUNCTION)

This form of **status** returns a numeric value representative of axis motion-related status.

SYNTAX

<axis identifier> **? status** ;

12.2.2 STATUS (BOOLEAN FUNCTION)

This form of **status** returns a Boolean value based on the logical AND of the axis motion status code with the specified status mask.

SYNTAX

<axis identifier> **? status** (<status mask identifier>) ;

<status mask identifier> : mask applied to the axis motion status code. A set of standard masks (see Table 12-4) are defined in the file ST1ERROR.SYS.

MASK VALUE	MASK NAME	STATUS
1	SYNCHRONISED	axis synchronised
2	IN_MOVE	axis in absolute or relative movement.
4	IN_RUN	axis in run.
16	POWER_OFF	axis with power off
32	STOPPED	axis stopped

Table 12-4 PAM Axis Status Masks

EXAMPLE

```
IF MyAxis ? status (POWER_OFF)
   THEN  ...
      ...            treatment for axis power off
```

12.3 FATAL ERROR PANEL

A PAM fatal error is a type of error which renders continuation of application execution impossible. The fatal error panel is an area of the DualPort used to communicate this type of error to a host controller outside of PAM. In the event of a fatal error, PAM places a message providing details concerning the first fatal error detected into the fatal error panel area of Dual-Port memory.

If a VME or a Simatic Dual-Port is used, the fatal error panel is located in Dual-Port memory. When the RS422 communication channel is used in parallel with a VME or Simatic Dual-Port, the fatal error panel is located in Simatic/VME Dual-Port memory.

The contents of the fatal error panel are valid only after PAM has stopped servicing the watchdog in response to a fatal error condition, and it should only be read by a Dual Port user at that time.

Fatal errors also produce message codes on the PamDisplay (see paragraph 11.7) and on the PAM Debugger.

12.3.1 FATAL ERROR MESSAGE FORMAT

Table 12-5 shows the format of a fatal error message in the fatal error panel. Each cell is 16 bits wide. The “OO” and “NNN” fields referenced in Table 12-5 refer to the general PAM error code structure (see Table 12-6).

0	Message origin (OO field value)
2	Fatal error number (NNN field value)
4	DATE : 6 bits delta year from 1990, 4 bits month, 5 bits day, 1 bit AM/PM (0=AM)
6	TIME : 4 bits hour (0-12), 6 bits minute, 6 bits second
8	Info TAG 0 = no info, 1 = node addr.
10	Info field 1 (node addr)
12	Info field 2 (reserved)

Table 12-5 Fatal Error Message Format

General PAM error code structure :

O	O	R	X	T	N	N	N

Table 12-6 PAM Error Code Structure

"R", "X", "T" fields are not placed in the fatal error panel.

12.3.2 SIMATIC FATAL ERROR PANEL LOCATION

In a Simatic DualPort (see chapter 14) the fatal error panel is located before the watch dog variable.

EXAMPLE

If the watch dog is located at relative address #FFE, the first cell (16 bits) of the fatal error panel is located at #FFE - #0E = #FF0 (size of panel is 14 bytes).

In this example, the panel can be read from PAM address #00701FE0 using the PAM debugger (#00700000 + (2 * #FF0)).

⚠ A space of At least 14 bytes must be left between last variable address and watch dog location.

12.3.3 VME FATAL ERROR PANEL LOCATION

In a VME DualPort (see chapter 13) the fatal error panel is located between the command port and the body of the input FIFO:

The S_dual_vme structure is:

```
/* VME dualport structure */
/* ---------------------- */

typedef struct {
  S_header_fifo_dp_vme  header_fifo_in;
  S_header_fifo_dp_vme  header_fifo_out;
  short                 watchdog;
  short                 synchro;
  S_port_cmd_host       cmd_port;
  S_error_panel         fatal_error_panel;
} S_dual_vme;
```

With the fatal error panel structure defined as follows:

```
typedef struct {
  short          mess_origin;
  short          fatal_err_nb;
  short          date;
  short          time;
  short          info_tag;
  short          info_1;
  short          info_2;
} S_error_panel;
```

The Panel offsets from top are as follows:

```
#define VME_FATAL_ERROR_MESS_ORIGIN          44
#define VME_FATAL_ERROR_MESS_NUMBER          46
```

```
#define VME_FATAL_ERROR_DATE            48
#define VME_FATAL_ERROR_TIME            50
#define VME_FATAL_ERROR_INFO_TAG        52
#define VME_FATAL_ERROR_INFO_1          54
#define VME_FATAL_ERROR_INFO_2          56
```

The fatal error panel can be read from PAM address #00700058 using the PAM debugger.

i | VME user may refer to PAM_DEF.H file.

12.3.4 SERIAL LINE FATAL ERROR PANEL LOCATION

In a Serial-Line Port (see chapter 15) the fatal error panel is located at relative Dual Port address #FF0. The panel can be read from PAM address #00701FE0 using the PAM debugger.

12.3.5 LIST OF FATAL ERRORS

This paragraph includes a listing of fatal errors which is organized by error category. Additional details on fatal errors (as well as non-fatal errors) may be found in the appendicies.

RING ERRORS

PAM LED	ON PAM	VALUES PLACED IN FATAL ERROR PANEL				
	DISPLAY	ORIGIN	ERR_NB	INFO_TAG	INFO_1	INFO_2
ON	0680F002	06	002	1	node add.	0
ON	0680F003	06	003	1	node add.	0
ON	0680F004	06	004	1	node add.	0
ON	0680F005	06	005	1	node add.	0
ON	0680F007	06	007	1	node add.	0
ON	0680F008	06	008	1	node add.	0
ON	0680F009	06	009	1	node add.	0
ON	0680F016	06	016	1	node add.	0
ON	0680F006	06	006	2	periph nb.	0
ON	0680F022	06	022	2	periph nb.	0

Table 12-7 Fatal Error Panel Contents for Ring Errors

FAULT HANDLER ERRORS

PAM LED	ON PAM	VALUES PLACED IN FATAL ERROR PANEL				
	DISPLAY	ORIGIN	ERR_NB	INFO_TAG	INFO_1	INFO_2
BLINK	0200FFF0	02	FF0	0	0	0
BLINK	0200FFF1	02	FF1	0	0	0
BLINK	0200FFF2	02	FF2	0	0	0
BLINK	0200FFF3	02	FF3	0	0	0

PAM LED	ON PAM	VALUES PLACED IN FATAL ERROR PANEL				
	DISPLAY	ORIGIN	ERR_NB	INFO_TAG	INFO_1	INFO_2
BLINK	0200FFF4	02	FF4	0	0	0
BLINK	0200FFF5	02	FF5	0	0	0
BLINK	0200FFF6	02	FF6	0	0	0
BLINK	0200FFF7	02	FF7	0	0	0
BLINK	0200FFF8	02	FF8	0	0	0

Table 12-8 Fatal Error Panel Contents for Fault Handler Errors

HIGH PRIORITY CYCLE DURATION ERROR

PAM LED	ON PAM	VALUES PLACED IN FATAL ERROR PANEL				
	DISPLAY	ORIGIN	ERR_NB	INFO_TAG	INFO_1	INFO_2
BLINK	0280F060	02	060	0	0	0

Table 12-9 Fatal Error Panel Contents for High Priority Cycle Duration Error

WORKSPACE ERROR

PAM LED	ON PAM	VALUES PLACED IN FATAL ERROR PANEL				
	DISPLAY	ORIGIN	ERR_NB	INFO_TAG	INFO_1	INFO_2
BLINK	0380F02C	03	02C	0	0	0

Table 12-10 Fatal Error Panel Contents for Workspace Errors

HOST NOT READY DURING PAM INITIALIZATION

PAM LED	ON PAM	VALUES PLACED IN FATAL ERROR PANEL				
	DISPLAY	ORIGIN	ERR_NB	INFO_TAG	INFO_1	INFO_2
ON	0780F020	07	020	0	0	0

Table 12-11 Fatal Error Panel Contents for Host not Ready Error

OTHER ERRORS THAT ABORT PAM INITIALISATION PHASE

PAM LED	ON PAM	VALUES PLACED IN FATAL ERROR PANEL				
	DISPLAY	ORIGIN	ERR_NB	INFO_TAG	INFO_1	INFO_2
ON	0780F0xx	07	0xx	0	0	0

Table 12-12 Fatal Error Panel Contents for Aborted PAM Initialization

12.4 MANAGING SEQUENCE WORKSPACES

Sequence workspace is the amount of internal memory which PAM allocates for a sequence each time it starts sequence execution. The default workspace size can limit the execution of a large number of simple sequences by over-allocating memory and possibly exceeding PAM's physical memory limit.

It is possible to individually specify the workspace for each sequence in an application. This feature permits an increase in the number of sequences that can be executed in parallel in very large applications

The procedure employed by PAM in assigning sequence workspace is as follows:

1. If the **WORKSPACE** parameter in the **SPECS** section of the sequence declaration is present, this value is used. Otherwise...

2. If the **DEFAULT_SEQUENCE_WORKSPACE** parameter in the **SPECS** section of the task declaration is present, this value is used. Otherwise...

3. If the **DEFAULT_TASK_WORKSPACE** parameter in application information section is present, this value is used. Otherwise...

4. By default, the internal workspace size (4096 bytes) is used.

EXAMPLE

This example illustrates assignment of sequence workspaces. Table 12-13 lists the workspace sizes implemented following execution of this example.

```
APPLICATION
   ...
   DEFAULT_TASK_WORKSPACE  = 3000 ;
END


   TASK Task1 ;
         SPECS
         DEFAULT_SEQUENCE_WORKSPACE = 4000 ;
      ...
      SEQUENCE Task1Sequence1 ;
         SPECS
         WORKSPACE = 1000 ;
      ...
      SEQUENCE Task1Sequence2 ;
      ...
END_TASK


TASK Task2 ;
   ...
   SEQUENCE Task2Sequence1 ;
      SPECS
      WORKSPACE = 1500 ;
   ...
   SEQUENCE Task2Sequence2 ;
   ...
END_TASK
```

The result is shown in the following table:

SEQUENCE	WORKSPACE SIZE
Task1Sequence1	1000
Task1Sequence2	4000
Task2Sequence1	1500
Task2Sequence2	3000

Table 12-13 Sequence Workspace Assignments for Example Program

13 VME BUS DUALPORT

13.1 INTRODUCTION

DualPort is the name given to the interface between PAM and a VME bus master through which system level variables are passed. DualPort variables are those variables which are passed through the DualPort. This interface is based on a dual port memory.

The DualPort serves only as the pick-up/drop-off point for exchanging variable values in this communication channel. DualPort variables do not reside in the dual port memory. Memory is allocated for each DualPort variable in the internal PAM memory and should likewise be allocated in the VME bus master memory.

In order to effectively utilise variable values passed back and forth across the DualPort, both the PAM application program and VME bus master program must have knowledge about each variable in the DualPort variable set, including the variable type (input or output), class (real, integer, boolean, etc.), and it's identifier (name). A C language INCLUDE file created during compilation of the PAM application program provides a list of all DualPort variables declared by the PAM application program. This file is intended to facilitate creation of an identical DualPort variable structure within the VME master program. During start-up PAM and the VME bus master verify that their definitions of each DualPort variable match.

Prior to reaching the point where variables can actually be exchanged, PAM and the VME bus master must go through a start-up sequence (See paragraph 13.4) which includes a start-up handshake, initial synchronisation, a configuration phase and initialisation phase.

13.2 GENERAL CONCEPT FOR EXCHANGING VARIABLES

There are two types of DualPort variables: input variables and output variables. Within each category, a DualPort variable may be any of the standard variable classes (see paragraph 13.8). When referring to any DualPort variable, direction (input or output) is always with respect to PAM. Therefore, a DualPort input variable is an input to PAM and a DualPort output variable is a PAM output. All DualPort variables (including input and output variables of all types) must be declared in the PAM application program. The general concept for exchanging variable values via the DualPort is summarised below.

13.2.1 INPUT VARIABLES

The VME bus master places an input variable value, along with it's PAM key (unique number defined and used by PAM to link data from the FIFO to the corresponding DualPort input variable) into the Input FIFO (portion of DualPort reserved for "dropping off" DualPort input variables). PAM removes the incoming value from the input FIFO and updates the designated variable to the new value.

13.2.2 OUTPUT VARIABLES

Whenever the value of a DualPort output variable changes, PAM places the new value along with it's VME Master key (unique number defined and used by the VME bus master to link data from the FIFO to the corresponding DualPort output variable) onto the Output FIFO (portion of dual port memory reserved for “picking up” DualPort output variables. The VME bus master removes the value from the output FIFO and updates the designated variable to the new value.

> Since identifiers for DualPort variables may be quite lengthy, key values (unique numbers which represent identifiers) are substituted for identifiers in FIFO blocks (data blocks passed through the FIFOs) to reduce block size and transmission time.

13.3 DUALPORT STRUCTURE

Figure 13-1 illustrates partitioning of dual port memory into the functional areas utilised by the DualPort. The function of each component of the DualPort is described in the following paragraphs. Dual port memory size is 4096 bytes, organised as 2048 sixteen bit words.

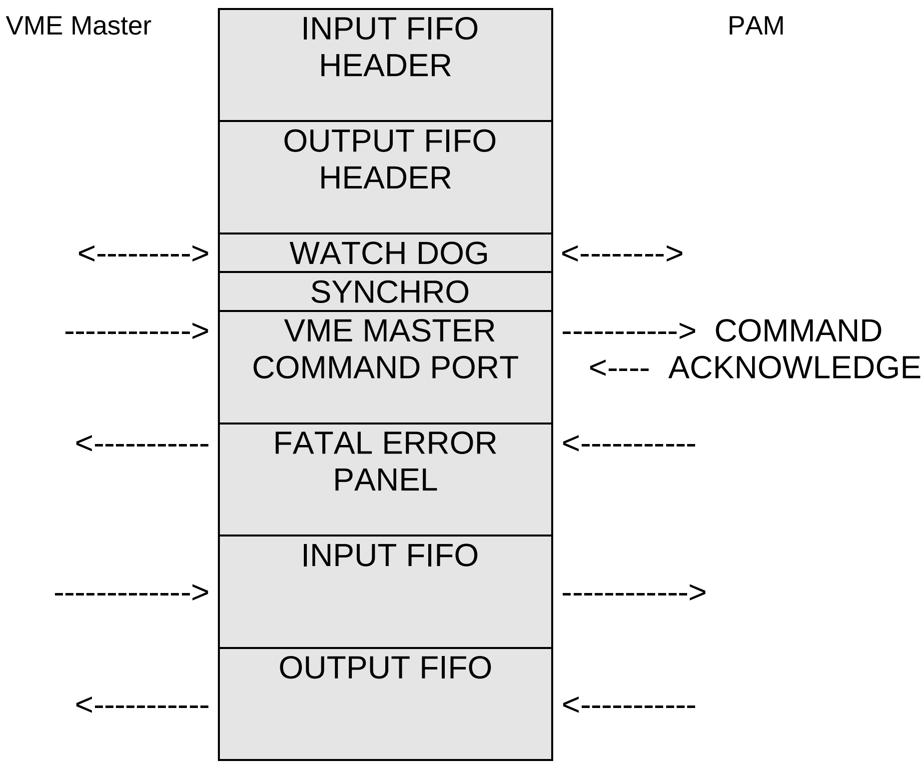


Figure 13-1 DualPort Memory Partitioning

13.3.1 INPUT FIFO HEADER

The input FIFO header (see Figure 13-2) contains pointers and indexes to data in the input FIFO. The FIFO Header is initialised by PAM. During data exchange the VME bus master and PAM manipulate the pointers using a specific protocol (see paragraph 13.6) as variable values are placed onto and removed from the FIFO.

FIFO OFFSET	offset (in bytes) from top of dual port memory to beginning of FIFO
HEAD (WRITE POINTER)	index to next empty block when writing into FIFO
TAIL (READ POINTER)	index to next block when reading FIFO
MAX BLOC NUMBER	number of blocs in the FIFO body
BLOC SIZE	size (in bytes) of FIFO data blocks

Figure 13-2 FIFO Header Arrangement

13.3.2 OUTPUT FIFO HEADER

Performs the same function for the Output FIFO as the input FIFO header performs for the Input FIFO (see paragraph 13.3.1).

13.3.3 WATCHDOG

This cell is reserved for a watchdog timer function implemented by PAM and VME bus master software (see paragraph 13.6.3).

13.3.4 SYNCHRO

This cell is used in a power-on synchronisation sequence implemented by PAM and VME bus master software (see paragraph 13.4.2).

13.3.5 VME MASTER COMMAND PORT

This segment of the DualPort (see Figure 13-3) is used by the VME bus master and PAM for passing commands and acknowledgements during DualPort start-up and configuration phases. The VME bus master places commands and parameters for PAM into the first four cells, and PAM places acknowledgements and parameters in cells five through ten. Table 13-1 lists the complete set of command and acknowledgement codes used with the VME Master Command Port.

From VME Master -------->	**VME COMMAND** (16 bits)	
(16 bits)	VME parameter 1	
(16 bits)	VME parameter 2	
(16 bits)	VME parameter 3	
(16 bits)	**PAM ACKNOWLEDGE**	**<---------- FROM PAM**
(16 bits)	PAM parameter 1	
(16 bits)	PAM parameter 2	
(16 bits)	PAM parameter 3	
(16 bits)	PAM parameter 4	
(16 bits)	PAM parameter 5	

Figure 13-3 VME Master Command Port Arrangement

CODE	VME COMMAND	PAM ACKNOWLEDGEMENT
0	**NOP**	**NO_ACK**
1		**ACK_OK**
2	**ASK_DEF_IN**	**BAD_CDE**
3	**DEF_OUT**	**CDE_REJECTED**
4	**END_DEF_OUT**	**END_DEF_IN**
5	**INIT_IN**	**BAD_CRC**
6	**END_INIT_IN**	
7	**START**	

Table 13-1 Command and Acknowledgement Codes

13.3.6 FATAL ERROR PANEL

This segment of the DualPort is reserved for communicating fatal errors detected during application execution to the VME Master. Refer to paragraph 12.3 for details on the fatal error panel.

13.3.7 INPUT FIFO

The Input FIFO is the area of the DualPort where the VME bus master places input variable values. A software protocol (see paragraph 13.6) for placing data onto and removing data from DualPort memory emulates the function of a hardware FIFO (first in first out) register. Input FIFO blocks (input variable data placed onto the input FIFO) use seven words in a format (see Table 13-8) which varies depending on the variable class. Capacity of the input FIFO is 144 blocks.

13.3.8 OUTPUT FIFO

The Output FIFO is the area of the DualPort where PAM places output variable values. Output FIFO blocks (output variable data placed onto the output FIFO) use seven words in a format (see Table 13-9) which varies depending on the variable class. In all other respects the Output FIFO is identical to the Input FIFO.

13.4 START-UP

The general sequence of events which takes place during start-up of VME Master/PAM communications via the DualPort includes start-up pre-set, synchronisation, configuration and initialisation phases. This sequence is executed upon power-up of the PAM and following a hardware reset. Each step in the DualPort start-up sequence is described in the remainder of this section.

13.4.1 START-UP PRE-SET

Start-up precept is the first step before establishing communications via the DualPort. The operations performed by the VME Bus Master and PAM are illustrated in Figure 13-4. **HEAD** and **TAIL** are set = 0 and the remaining fields in the FIFO headers are initialised as required by the application configuration. There is no required sequence and no timeout associated with completion of start-up pre-set. Upon completion of it's operations, PAM proceeds to the synchronisation phase.

VME Bus Master Pre-set	**PAM Pre-set**
NOP (Code = 0) → VME COMMAND 0 → VME PARAMETER 1	**ACK_OK** (Code = 1) → PAM ACKNOWLEDGE 0 → PAM PARAMETER 1 Initialise input & output FIFO headers.

Figure 13-4 Start-up pre-set

13.4.2 SYNCHRONISATION

During synchronisation, the VME bus master and PAM use the **SYNCHRO** location in DualPort memory (see Figure 13-1). To avoid deadlock during synchronisation no pre-set is done to the **SYNCHRO** location. Synchronisation is initiated by the first system (VME bus master or PAM) finding a true condition of **SYNCHRO**, and is completed when the series of manipulations of **SYNCHRO** illustrated in Figure 13-5 is completed. Note that a Master Ready Timeout (see paragraph 13.4.2.1 will occur if the delay between successive steps in the sequence exceeds the **MASTER_READY_TIMEOUT** parameter of DualPort Header declaration (see paragraph 13.7.1).

VME Master action	**PAM response**
wait until **SYNCHRO** <> 0 (with time-out), then 0 → **SYNCHRO**	wait until **SYNCHRO** = 0 (with time-out), then 1 → **SYNCHRO**
wait until **SYNCHRO** = 1 (with time-out), then 0 → **SYNCHRO**	wait until **SYNCHRO** = 0 (with timeout), then 1 → **SYNCHRO**

VME Master action	PAM response
	wait for first command of Configuration phase (see paragraph 13.4.3).

Figure 13-5 Synchronisation Sequence

13.4.2.1 MASTER READY TIMEOUT

ERROR CODE

[07xxF020] Real Time Fatal Error HOST NOT READY

CAUSE

Interval between successive steps in synchronisation sequence exceeds **MASTER_READY_TIMEOUT** parameter of DualPort Header declaration (see paragraph 13.7.1).

PAM ACTION

- Initialisation aborted, PAM stops and must then be restarted by performing a hardware reset.
- Error code stored in fatal error panel.

13.4.3 CONFIGURATION PHASE

13.4.3.1 INTRODUCTION

During the configuration phase, the VME master and PAM exchange details on DualPort variables via the VME Master Command Port using a pre-defined set of commands, acknowledgements and parameter structures. The purpose of the configuration phase is to verify that both PAM and VME bus master's descriptions of each DualPort variable match, and to enable PAM and the VME bus master to cross-reference keys to corresponding DualPort variables. To facilitate this, the PAM compiler, during compilation of the application, produces a C language INCLUDE file (see paragraph 13.4.5) containing the CRC (cyclic redundancy check) value for each DualPort variable declared in the application along with it's identifier. This file, which is intended for inclusion in the VME bus master program, establishes the link between CRC values and corresponding DualPort variables.

> i The inherent characteristics of CRC codes guarantees that different DualPort variable names will never produce the same CRC code.

The entire configuration sequence, which begins with an exchange of DualPort input variable definitions followed by output variable definitions, is controlled by the VME bus master.

13.4.3.2 INPUT VARIABLES CONFIGURATION

Configuration of input variables is initiated by the VME bus master when it places the command code for the **ASK_DEF_IN** (ask definition of input variable) command in the VME COMMAND location of the VME Master Command Port (see Figure 13-6).

VME Master --> to request next input variable definition

ASK_DEF_IN
0
0
0

ACK_OK
PAM CRC
PAM KEY
PAM VARIABLE CLASS
MAX. NUMBER of ITEMS
PAM PROCESSING PERIOD

<-- **PAM** input variable definition

END_DEF_IN
0
0
0
0
0

<-- **PAM** To signal end of input definitions

Figure 13-6 Command Port During Input Variables Configuration

PAM then places the first DualPort input variable definition in the PAM PARAMETERS locations and acknowledges by placing the **ACK_OK** acknowledgement code in the PAM ACKNOWLEDGEMENT location.

Each DualPort input variable definition includes a set of parameters (see Figure 13-6) which specify important characteristics of the variable. These parameter values are determined by PAM based on the variable declaration. Table 13-2 provides details on the meaning and format of each parameter.

INPUT DEFINITION PARAMETER	DESCRIPTION

PAM CRC	CRC for variable, computed by PAM from DualPort variable declaration
PAM KEY	Key value assigned by PAM
PAM VARIABLE CLASS	Code which defines the type of variable. The include file VMECLASS.H contents DualPort variable types definitions. This file is located in sub directory VME\VME_USER after PAM tools installation.
NUMBER OF ITEMS	Maximum number of variables from declaration: >1 for variables which are multiple 1 for variable which is single
PAM PROCESSING PERIOD	PAM scanning period from declaration (in msec.)

Table 13-2 Input Variable Definition Parameters

This sequence is repeated until PAM responds with an **END_DEF_IN** acknowledgement after all input variable definitions have been sent. Table 13-3 illustrates the sequence of command/acknowledgement possibilities for normal and abnormal situations. A Master Configuration Timeout (see paragraph 13.4.3.4) will occur if the duration between successive commands and acknowledgements exceeds the **MASTER_CONFIGURATION_TIMEOUT** parameter in the DualPort Header declaration.

VME MASTER ACTION	PAM RESPONSE
ASK_DEF_IN Code = 2 VME master asks PAM to place an input variable definition in VME Master Command Port. VME bus master fetches input variable definition from VME Master Command Port. VME bus master must verify CRC code exists in its own list, then verify variable class and number of items against it's own declaration. When everything matches, VME master stores PAM key. If CRC code does not match, VME master cannot assign PAM key. Error may be reported; however, VME master must continue configuration. If variable class or number of items does not match, VME master should denote mismatch by using "-1" for key code. Error may be reported; however, VME master must continue configuration. IF timeout, abort start-up.	**ACK_OK** Code = 1 PAM has placed an input variable definition in the VME Master Command Port. IF Timeout, abort start-up.
	END_DEF_IN Code = 4 PAM has sent All input variable definitions.
VME bus master must verify that PAM keys have been assigned to all input variable definitions in it's list. VME bus master should denote definitions with no PAM key using a key value of "-1". Error may be reported.	

Table 13-3 Input Variables Configuration Sequence

13.4.3.3 OUTPUT VARIABLES CONFIGURATION

Configuration of output variables is initiated by the VME bus master when it places the first output variable definition parameters in VME PARAMETER locations of the VME Master Command Port, then places the command code for the **DEF_OUT** (send output variable definition) command in the VME COMMAND location (see Figure 13-7). PAM then places the VARIABLE CLASS and NUMBER OF ITEMS parameters from it's output variable definition in the PAM PARAMETER locations and acknowledges by placing the **ACK_OK** acknowledgement code in the PAM ACKNOWLEDGEMENT location.

Each DualPort output variable definition sent by the VME bus master includes a set of parameters (see Figure 13-7) required by PAM to identify and cross-reference the variable within its DualPort variable structure. Table 13-4 provides details on the meaning and format of each parameter.

VME Master to send next output variable definition -->

DEF_OUT
VME MASTER CRC
MASTER KEY (lsw)
MASTER KEY (msw)

<-- **PAM** to acknowledge receiving definition

ACK_OK
0
0
PAM VARIABLE CLASS
NUMBER OF ITEMS
0

OR

<-- **PAM** when it can’t match CRC

BAD_CRC
0
0
0
0
0

VME Master signal end of output variable definitions -->

END_DEF_OUT
0
0
0

Figure 13-7 Command Port during Output Variables Configuration

OUTPUT DEFINITION PARAMETER	DESCRIPTION
VME MASTER CRC	CRC for variable, computed by PAM from DualPort variable declaration and passed to VME bus master in INCLUDE file.
MASTER KEY (lsw)	Key value assigned by VME bus master (bit 0 - 15)
MASTER KEY (msw)	Key value (bit 16 - 31)

Table 13-4 Output Variable Definition Parameters

This sequence is repeated until the VME bus master sends an **END_DEF_OUT** command after all output variable definitions have been sent. Table 13-5 illustrates the sequence of command/acknowledgement possibilities for normal and abnormal situations. A Master Configuration Timeout (see paragraph 13.4.3.4) will occur if the duration between successive commands and acknowledgements exceeds the **MASTER_CONFIGURATION_TIMEOUT** parameter in the DualPort Header declaration.

VME Master action	**PAM response**
DEF_OUT Code = 3 VME master places an output variable definition in VME Master Command Port. When VME master receives **ACK_OK**, it checks VARIABLE CLASS and NUMBER OF ITEMS parameters against it's own definition. If definitions correspond, VME master sends next definition. If not, master must change its own value of key to -1 in order to ignore this variable when PAM announces a change. Error may be reported. VME master sends the next definition. IF timeout, abort start-up.	**ACK_OK** Code = 1 PAM has matched CRC with it's own list, stored the master key, and placed VARIABLE CLASS and NUMBER OF ITEMS parameters from it's definition in the VME Master Command Port. Pam initialises the output variable to zero (0) . IF timeout, abort start-up.
When VME master receives **BAD_CRC**, error may be reported. VME master sends the next definition.	**BAD_CRC** Code = 5 PAM could not find the CRC in it's own list. PAM displays error [07xxE028] (this output doesn't exist for PAM).

VME Master action	PAM response
END_DEF_OUT Code = 4 VME master sends this command when all output variable definitions have been sent.	When PAM receives **END_DEF_OUT**, it must check its own list for output definitions with no master key assigned. The key value of these "extra" definitions is set to -1 to cancel them. No errors are displayed. IF timeout, abort start-up.

Table 13-5 Output Variables Configuration Sequence

13.4.3.4 MASTER CONFIGURATION TIMEOUT

ERROR CODE

[07xxF020] Real Time Fatal Error HOST NOT READY

CAUSE

Interval between a command and acknowledgement exceeds the value of **MASTER_READY_TIMEOUT** parameter in DualPort Header declaration (see paragraph 13.7.1).

PAM ACTION

- Initialisation aborted, PAM stops and must then be restarted by performing a hardware reset.
- Error code stored in fatal error panel.

13.4.4 PAM ERROR MESSAGES DURING CONFIGURATION

The following error condition related to configuring the DualPort produces an error message on the PAM display:

ERROR CODE

[07xxE028] "Bad CRC value for a VME DualPort Output variable" with :

D1 : CRC value received

D2 : KEY value received

D3 : 0

CAUSE

CRC code in output variable definition does not exist in PAM's list.

PAM ACTION

- PAM unable to update changes in output variable for which it cannot cross-reference CRC code.

13.4.5 CONFIGURATION FILES FOR INCLUSION IN VME MASTER

All needed definitions are provided in the following three include files :

PAM_DEF.H	This file contains the description of dualport memory from VME master view.
VMECLASS.H	This file contains the definitions corresponding to the variable class (type).
PAM_VME.H	This file contains a list of CRC values and corresponding DualPort variable identifiers compiled from the DualPort variable declarations. List entries are in the following format: #define <DualPort Variable identifier> <CRC value> PAM_VME.H is automatically generated during execution of PAMCOMP, so this file is application dependent. This file is placed into the \AGL sub directory of the compiled application.

Their purpose is to simplify set-up of the DualPort variables structure in the VME bus master. During compilation of the application program, the C language "INCLUDE" file PAM_VME.H is produced by the PAM compiler for inclusion in the VME master application. The two other INCLUDE files may be found (after PAM TOOLS installation) in the sub-directory VME\VME_USER.

13.4.6 INITIALISATION

13.4.6.1 INTRODUCTION

During Initialisation, the starting values for DualPort variables are established. Initialisation takes place during start-up, prior beginning application program execution.

13.4.6.2 OUTPUT VARIABLES

PAM initialises all DualPort output variables to zero. Thereafter, PAM sends only changes in value to the DualPort. The VME master must also initialise the DualPort output variables in its own memory to zero.

13.4.6.3 INPUT VARIABLES

DualPort input variable initial values are supplied by the VME bus master. The VME bus master indicates it is ready to begin sending initial values by placing the **INIT_IN** command code into the VME COMMAND location of the VME Master Command Port. Next, the VME master begins placing initial values of all input variables onto the Input FIFO. The complete initialisation sequence is detailed in Table 13-6. To prevent a Timeout error, the initialisation sequence must proceed at a rate which exceeds the **MASTER_CONFIGURATION_TIMEOUT** parameter in the DualPort Header declaration (see paragraph 13.7.1).

VME Master action	PAM action
INIT_IN Code = 5 VME bus master informs PAM it is starting to place initial input variable values on the Input FIFO.	**ACK_OK** Code = 1 PAM begins looking for input variable values (with timeout).
VME bus master starts placing initial values of input variables onto Input FIFO in standard FIFO Input Block format (see Table 13-8).	PAM pulls initial values off Input FIFO and updates it's corresponding DualPort input variables (with timeout). If the PAM key in a FIFO block does not match a key stored during configuration, PAM displays error [07xxE029] and continues. If the item index value in a FIFO block is out of range , PAM displays error [07xxE02A] and continues.
END_INIT_IN Code = 6 VME bus master informs PAM it has placed all input variable initial values onto the Input FIFO.	**ACK_OK** Code = 1 PAM acknowledges when it has emptied the Input FIFO.

Table 13-6 Input Variables Initialisation Sequence

13.4.6.4 PAM ERROR MESSAGES DURING VARIABLE INITIALISATION

The following error condition related to initialising DualPort variables produce an error message on the PAM display:

ERROR CODE

[07xxE029] "Bad KEY value for a VME Input variable" with :

D1 : KEY value received

D2 : corresponding KEY value in PAM or FFFFFFFF

D3 : 0

CAUSE

PAM KEY in an input FIFO block does not match PAM key defined during configuration.

PAM ACTION

- FIFO block with bad key is discarded.
- PAM continues.

ERROR CODE

[07xxE02A] "Bad VME variable index" with :

D1 : value of PAM KEY	(hexadecimal)
D2 : PAM compiler ID of the variable	(hexadecimal)
D3 : index received	(hexadecimal)

CAUSE

ITEM INDEX in input FIFO block is out of range when compared with NUMBER OF ITEMS parameter in variable definition.

PAM ACTION

- FIFO block with bad key is discarded.
- PAM continues.

13.5 EXECUTION PHASE

13.5.1 INTRODUCTION

During the execution phase PAM is executing the resident application program. The Input FIFO is checked each PAM cycle for new input variable values and output variable changes (new values) are placed on the Output FIFO as they are detected. The Watchdog (see paragraph 13.6.3) is also active during execution phase. The execution phase is started by command from the VME bus master (see paragraph 13.5.2).

13.5.2 STARTING PAM APPLICATION EXECUTION

The sequence for starting application program execution is detailed in Table 13-7.

VME Master action	PAM action
START Code = 7 VME bus master commands PAM to start application execution. VME master may check for **ACK_OK**.	**ACK_OK** Code = 1 If all phases of configuration are completed, PAM starts placing into Output FIFO new values of DualPort outputs, starts scanning Input FIFO and starts the watchdog.
VME master may report error.	**CMD_REJECTED** Code = 3 Sent if configuration phase has not been completed; however, PAM does begin application execution.

Table 13-7 Sequence for Starting Program Execution

13.5.3 CHANGING DUALPORT VARIABLE VALUES

13.5.3.1 INPUT VARIABLE CHANGES

When the VME bus master wishes to change the value of a DualPort input variable, it must get the index of the first free input FIFO block and fill it with the necessary data in the specified format for the variable class (see Table 13-8), then update the Input FIFO header. Paragraph 13.6.1 describes the general procedure to be followed by the VME bus master when writing into the Input FIFO. The following definitions apply to the Input FIFO Block components listed in Table 13-8:

PAM Key	cross-reference to DualPort input variable. Assigned by PAM during configuration
item index	designates specific variable when variable is multiple: 0 if variable is single 1 - (NUMBER-1) to designate specific item in multiple variable -1 to designate all items in a multiple variable
flag value	0 for false 1 for true
value	variable value in binary: for word variables = 16 bit signed integer for long variables = 32 bit signed integer
mantissa	48 bit binary fraction per IEEE 754-1985
exponent	16 bit binary exponent per IEEE 754-1985

byte	**flag variables**	**word variables**	**long variable**	**floating variable**
0	PAM Key	PAM Key	PAM Key	PAM Key
2	0	0	0	0
4	item index	item index	item index	item index
6	flag value	value	value (bit 0-15)	mantissa (bit 0-15)
8			value (bit 16-31)	mantissa (bit 16-31)
10				mantissa (bit 32-47)
12				exponent (bit 48-63)

Table 13-8 Input FIFO Block Formats

During each PAM cycle, PAM extracts FIFO blocks from the Input FIFO. Using the PAM Key as a cross-reference to the input variable, PAM places new variable values into an internal queue for subsequent processing. Depending on activity level and the quantity of data in the FIFO, the Input FIFO may or may not be emptied during a PAM cycle.

i

Internally, PAM updates DualPort input variables at the interval specified by the **PERIOD** parameter in the variable declaration, or if not specified, at the interval specified by the **DEFAULT_PERIOD** parameter of DualPort header . Therefore, DualPort input variable changes (and events linked to input variable changes) occur once per **PERIOD**, regardless of the number (or frequency) of changes to a given variable placed onto the FIFO by the VME master.

If PAM cannot cross-reference the PAM Key to an input variable or if the item index is out of range, PAM discards the errant FIFO block.

13.5.3.2 OUTPUT VARIABLE CHANGES

When PAM detect a change in value of a DualPort output variable, PAM fills the first available Output FIFO block with the necessary data in the specified format for the variable class (see Table 13-9), then updates the Output FIFO header. The following definitions apply to the Output FIFO Block components listed in Table 13-9:

i

Changes of DualPort Output values are immediately placed onto the output FIFO (provided that the output FIFO is not full). The PERIOD qualifier for DualPort Outputs is used only to have a period value when a DualPort output is used as term of a boolean equation.

Master Key — cross-reference to DualPort output variable. Assigned by VME bus master during configuration.

item index — designates specific variable when variable is multiple:
0 if variable is single
1 - (NUMBER-1) to designate specific item in multiple variable
-1 to designate all items in a multiple variable

flag value — 0 for false

1 for true

value — variable value in binary:
for word variables = 16 bit signed integer
for long variables = 32 bit signed integer

mantissa — 48 bit binary fraction per IEEE 754-1985

exponent — 16 bit binary exponent per IEEE 754-1985

byte	**flag variables**	**word variables**	**long variable**	**floating variable**
0	MASTER Key (bit 0 - 15)	MASTER Key (bit 0 - 15)	MASTER Key (bit 0 - 15)	MASTER Key (bit 0 - 15)
2	MASTER Key (bit 16 - 31)	MASTER Key (bit 16 - 31)	MASTER Key (bit 16 - 31)	MASTER Key (bit 16 - 31)
4	item index	item index	item index	item index
6	flag value	value	value (bit 0-15)	mantissa (bit 0-15)

byte	flag variables	word variables	long variable	floating variable
8			value (bit 16-31)	mantissa (bit 16-31)
10				mantissa (bit 32-47)
12				exponent (bit 48-63)

Table 13-9 Output FIFO Block Formats

The VME master must cyclically check the Output FIFO for new data. When the VME master find a new FIFO block, it must remove the block from the Output FIFO and update it's copy of the DualPort output variable. Paragraph 13.6.2 describes the general procedure to be followed by the VME bus master when checking or reading the Output FIFO.

If the Master Key received cannot be referenced to a DualPort output variable or the item index is invalid, the VME master should discard the data and signal an error.

13.6 FIFO READING AND WRITING PROTOCOL

13.6.1 WRITING INTO INPUT FIFO

The following principle is used by the VME bus master when writing into the Input FIFO:

1) compute new HEAD

    new_head = HEAD + 1

2) testing for end of FIFO

    if new_head >= MAX_BLOC_NUMBER

        THEN new_head = 0

3) testing for FIFO full

    if new_head = TAIL

        THEN FIFO is full !

4) compute write address in FIFO

    P_write = TOP_DUALPORT_ADDRESS + FIFO OFFSET +(HEAD * BLOCK_SIZE)

5) fill the block at P_write address

    P_write->key= KEY

    P_write->item_index= INDEX

    P_write->value= VALUE

6) update HEAD

    HEAD = new_head

i

HEAD, TAIL, MAX_BLOC_NUMBER, FIFO OFFSET and BLOC_SIZE are fields of the Input FIFO Header (see Figure 13-2).

HEAD & TAIL are indexes to the FIFO body. The FIFO body may be viewed as an array of blocks.

The value of TOP_DUALPORT_ADDRESS is related to VME bus configuration and jumper settings on the PAM board (see VME technical manual 006.8020).

13.6.2 READING FROM OUTPUT FIFO

The following principle is used by the VME bus master when reading from the Output FIFO:

1) testing for FIFO empty

    if TAIL = HEAD

        THEN FIFO is empty !

2) compute bloc read address

    P_read = TOP_DUALPORT_ADDRESS + FIFO OFFSET + (TAIL * BLOC SIZE)

3) read the bloc contents at P_read address

KEY = P_read->key

INDEX = P_read->item_index

VALUE = P_read->value

4) update TAIL

new_tail = TAIL + 1

if new_tail >= MAX_BLOC_NUMBER

THEN new_tail = 0

TAIL = new_tail

HEAD, TAIL, MAX_BLOC_NUMBER, FIFO OFFSET and BLOC_SIZE are fields of the Output FIFO Header (see Figure 13-2).

HEAD & TAIL are indexes to the FIFO body. The FIFO body may be viewed as an array of blocks.

The value of TOP_DUALPORT_ADDRESS is related to VME bus configuration and jumper settings on the PAM board (see VME technical manual 006.8020).

13.6.3 WATCHDOG

The DualPort register WATCHDOG (see Figure 13-1) is utilised as follows by PAM and the VME master to verify that each is still active (alive).

VME MASTER action : (at each VME MASTER cycle (10 ms for example))

(cycle_nb = PAM_timeout / cycle duration)

IF WATCHDOG = 0

THEN (* OK *) write 1 into WATCHDOG

cycle_count = 0

ELSE cycle_count = cycle_count + 1

IF cycle_count > cycle_nb

THEN PAM is not running

PAM action : (at each PAM cycle of 10 ms))

(cycle_nb = VME MASTER_timeout / cycle duration)

IF WATCHDOG = 1

THEN(* OK *) write 0 into WATCHDOG

cycle_count = 0

ELSE cycle_count = cycle_count + 1

IF cycle_count > cycle_nb

THEN VME MASTER is not running

13.6.3.1 WATCHDOG TIMEOUT ERROR

ERROR CODE

[07xxE01E] “Dual port watch dog !”

CAUSE

VME master has not written into **WATCHDOG** for an interval exceeding **MASTER_INACTIVITY_TIMEOUT** parameter of DualPort declaration.

PAM ACTION

- The DualPort error status becomes true (**DualPort ? error = true**)
- PAM stops acknowledging the watch dog to the VME master
- Application continue to be executed, but PAM stops scanning of DualPort inputs

13.7 VME DUALPORT DECLARATIONS

The ***DualPort*** object is pre-declared, so it does not need to be declared in the application.

The declaration is divided into three sequential parts :

- VME Dual Port Header
- List of individual variable declarations
- **END_DUALPORT** key word

13.7.1 VME DUALPORT HEADER

SYNTAX

```
VME DUALPORT
        SPECS
                DEFAULT_PERIOD                       =  <scanning period> ;
                MASTER_READY_TIMEOUT                 = <ready timeout> ;
        MASTER_CONFIGURATION_TIMEOUT   = <config. timeout>
        MASTER_INACTIVITY_TIMEOUT      = <inactive timeout> ;
        END_SPECS

            {<individual DualPort variable declarations>)

END_DUALPORT
```

<scanning period> : (NA), default scanning period (1, 5, 10, 20, 50 PAM basic cycles) for all DualPort variables.

Different scanning intervals for individual DualPort variables may be specified in their individual declarations.

<ready timeout > : (RO), timeout value in msec used by PAM when waiting for synchronisation with VME Master. ***NONE*** *deactivates the timeout.*

<config timeout > : (RO), timeout value in msec used during configuration phase, when waiting for master commands or acknowledgements. ***NONE*** *deactivates the timeout.*

<inactive timeout > : (RW), timeout value in msec used during execution phase. The VME master is considered inactive when it has not responded via the Watchdog within the inactive timeout interval. ***NONE*** *deactivates the timeout.*

SAMPLE DECLARATION

```
VME DUALPORT
    SPECS
    DEFAULT_PERIOD = 20 ;
        MASTER_READY_TIMEOUT        = 20000;
        MASTER_CONFIGURATION_TIMEOUT =  5000;
        MASTER_INACTIVITY_TIMEOUT    =   500;
    END_SPECS
```

FUNCTIONS

– **error**

This Boolean inquire function is true if an error condition exists in the DualPort.

SYNTAX

DualPort ? error

EXAMPLES

```
IF DualPort ? error THEN ...
EXCEPTION DualPort ? error SEQUENCE SEQ_DualportErrorHandling ;
```

– **error_code (Boolean)**

This Boolean inquire function is true if the error code parameter matches the error state of the DualPort.

SYNTAX

DualPort ? error_code(<error code>**)**

EXAMPLES

```
IF DualPort ? error_code(DUALPORT_MASTER_STOPPED)THEN ...
EXCEPTION DualPort ? error_code(DUALPORT_MASTER_STOPPED) SEQUENCE
....
```

i | The INCLUDE file "PORTERR.SYS" contains the DualPort error code definitions.

– **error_code (numerical)**

This inquire function returns the current error code of the DualPort.

SYNTAX

DualPort ? error_code

EXAMPLE

```
IF DualPort ? error_code = DUALPORT_MASTER_OK THEN ...
```

13.7.2 DUALPORT PARAMETER ACCESS

DualPort parameters may be read or modified, subject to each parameter's access level using the standard parameter access syntax (see paragraph 4.3.4).

EXAMPLES

```
/* wait till initial value is elapsed */
WAIT_TIME( DualPort:MASTER_INACTIVITY_TIMEOUT);
```

```
/* reduce the timeout value */
DualPort:MASTER_INACTIVITY_TIMEOUT <- 500;
```

13.8 DUALPORT VARIABLE DECLARATION

SYNTAX

The general declaration syntax for DualPort variables is listed below. The declaration syntax and functions for each type and class of DualPort variable are listed in subsequent paragraphs of this section.

<direction> <class > <identifier> **;**
  [{ **NUMBER** = <number> ¦
  NODES_GROUP = <nodes group identifier> } ;]
  [**PERIOD** **=** <scanning period> ;]
[**END]**

<direction> : (NA) specifies the direction of exchange (input or output).

<class> : (NA) specifies the class of the variable (flag, word, long, real).

<identifier>: (NA) specifies the variable name.

<number> : (NA) defines the number of items in a multiple variable.

<nodes group identifier> : (NA) the name of a node group used to define the number of duplicates in a multiple variable using the <number> parameter in the nodes group declaration.

> Variables associated with a Nodes Group via the **NODES_GROUP_IDENTIFIER** parameter are not serviced (scanned) when the associated component is inactive.

<scanning period>: (NA) scanning interval in msec (i.e. 1, 5, 10, 20, 50 msec). For individual DualPort variables this parameter overrides the default scanning period parameter in the DualPort Header declaration. If omitted, the default scanning interval is used.

> Internally, PAM updates DualPort input variables at the interval specified by the **PERIOD** parameter in the variable declaration or, if not specified in the variable declaration, at the interval specified by the **DEFAULT_PERIOD** parameter of the **DUALPORT HEADER.**
>
> This means it is possible to define a default scanning period for all DualPort input variables and specify, for some variables, a scanning period other than the default. The value of the given period [msec] is rounded up to the closest value of the cycle list (1,5,10,20,50) of **BASIC_PAM_CYCLE**.
>
> The **PERIOD** parameter for DualPort Outputs is used only to have a period value when a DualPort output is used as term of a boolean equation. DualPort output variable changes are immediately placed onto the output FIFO.

13.8.1 DUALPORT INPUT FLAG VARIABLE

SYNTAX

INPUT FLAG_VAR <identifier>
[{ **NUMBER** = <number> ¦
NODES_GROUP = <nodes group identifier> } ;]
[**PERIOD** **=** <scanning period in ms> ;]
[**END**]

FUNCTIONS

FUNCTION NAME	SET/ INQUIRE	PARAMETERS	RETURN VALUE
<- (read value)	INQUIRE	-	BOOLEAN

13.8.2 DUALPORT OUTPUT FLAG VARIABLE

SYNTAX

OUTPUT FLAG_VAR <identifier>
[{ **NUMBER** = <number> ¦
NODES_GROUP = <nodes group identifier> } ;]
[**PERIOD** **=** <scanning period in ms> ;]
[**END**]

FUNCTIONS

FUNCTION NAME	SET/ INQUIRE	PARAMETERS	RETURN VALUE
<- (assign value)	SET	- boolean expression	-
invert	SET	-	-
reset	SET	-	-
set	SET	-	-
<- (read value)	INQUIRE	-	BOOLEAN

13.8.3 DUALPORT INPUT WORD VARIABLE

SYNTAX

INPUT WORD_VAR <identifier> ;
[{ **NUMBER** = <number> ¦
NODES_GROUP = <nodes group identifier> } ;]
[**PERIOD** = <scanning period in ms> ;]
[**END**]

FUNCTIONS

FUNCTION NAME	SET/ INQUIRE	PARAMETERS	RETURN VALUE
<- (read value)	INQUIRE	-	INTEGER

WORD inputs are 16 bits signed integer !

13.8.4 DUALPORT OUTPUT WORD VARIABLE

SYNTAX

OUTPUT WORD_VAR <identifier> ;
[{ **NUMBER** = <number> ¦
NODES_GROUP = <nodes group identifier> } ;]
[**PERIOD** = <scanning period in ms> ;]
[**END**]

FUNCTIONS

FUNCTION NAME	SET/ INQUIRE	PARAMETERS	RETURN VALUE
<- (assign value)	SET	- numerical expression	-
<- (read value)	INQUIRE	-	INTEGER

WORD outputs are 16 bits signed integers !

13.8.5 DUALPORT INPUT LONG VARIABLE

SYNTAX

INPUT LONG_VAR <identifier> ;
[{ **NUMBER** = <number> ¦
NODES_GROUP = <nodes group identifier> } ;]
[**PERIOD** = <scanning period in ms> ;]
[**END**]

FUNCTIONS

FUNCTION NAME	SET/ INQUIRE	PARAMETERS	RETURN VALUE
<- (read value)	INQUIRE	-	INTEGER

LONG inputs are 32 bits signed integers !

13.8.6 DUALPORT OUTPUT LONG VARIABLE

SYNTAX

OUTPUT LONG_VAR <identifier> ;
[{ **NUMBER** = <number> ¦
NODES_GROUP = <nodes group identifier> } ;]
[**PERIOD** = <scanning period in ms> ;]
[**END**]

FUNCTIONS

FUNCTION NAME	SET/ INQUIRE	PARAMETERS	RETURN VALUE
<- (assign value)	SET	- numerical expression	-
<- (read value)	INQUIRE	-	INTEGER

LONG outputs are 32 bits signed integers !

13.8.7 DUALPORT INPUT REAL VARIABLE

SYNTAX

INPUT REAL_VAR <identifier> ;
[{ **NUMBER** = <number> ¦
NODES_GROUP = <nodes group identifier> } ;]
[**PERIOD** = <scanning period in ms> ;]
[**END**]

FUNCTIONS

FUNCTION NAME	SET/ INQUIRE	PARAMETERS	RETURN VALUE
<- (read value)	INQUIRE	-	REAL

REAL inputs are 64 bits floating ! (IEEE 754-1985)

13.8.8 DUALPORT OUTPUT REAL VARIABLE

SYNTAX

OUTPUT REAL_VAR <identifier> ;
[{ **NUMBER** = <number> ¦
NODES_GROUP = <nodes group identifier> } ;]
[**PERIOD** = <scanning period in ms> ;]
[**END**]

FUNCTIONS

FUNCTION NAME	SET/ INQUIRE	PARAMETERS	RETURN VALUE
<- (assign value)	SET	- numerical expression	-
<- (read value)	INQUIRE	-	REAL

REAL outputs are 64 bits floating ! (IEEE 754-1985)

13.8.9 DUALPORT VARIABLES DECLARATION EXAMPLE

```
VME DUALPORT
   SPECS
      DEFAULT_PERIOD = 20 ;
      MASTER_READY_TIMEOUT = 500 ;
      MASTER_CONFIGURATION_TIMEOUT = NONE ;
      MASTER_INACTIVITY_TIMEOUT = 500 ;
   END_SPECS

   INPUT FLAG_VAR  DFI_MachRunning ;  // single variable

   INPUT FLAG_VAR  DFI_MachAlarm ;    // single variable with period
spec.
      PERIOD = 10 ;
   END

   INPUT FLAG_VAR  DFI_MachHeadReady ;// multiple variable with dynanic
      NODES_GROUP = NGR_PrintHeads ;  // configuration
   END

   INPUT FLAG_VAR  DFI_MachHeadStopped ;    // multiple variable with
      NUMBER = 6 ;                                // static
configuration
   END

   INPUT WORD_VAR  DWI_MachPressure ;
      PERIOD = 5 ;
   END

   INPUT LONG_VAR  DLI_MachHeadSpeed ;
      NUMBER  = 6 ;
   END

   INPUT REAL_VAR  DRI_MachHeadFactor ;
      NUMBER  = 6 ;
   END

   OUTPUT FLAG_VAR  DFO_MachStopped ;

   OUTPUT WORD_VAR  DWO_MachCurrentPressure ;
      PERIOD = 50 ;
   END

   OUTPUT LONG_VAR  DLO_MachHeadPosition ;
      NODES_GROUP = PrintHeads ;
   END

   OUTPUT REAL_VAR  DRO_MachHeadLevel ;
      NODES_GROUP = NGR_PrintHeads ;
   END

END_DUALPORT
```

14 SIMATIC S5 DUALPORT

14.1 INTRODUCTION

DualPort is the name given to the interface between PAM and a Siemens Simatic S5 PLC (Programmable Logic Controller), because this interface is based on a DualPort memory. DualPort variables are those variables which are passed through the DualPort. DualPort variables reside at specific locations in DualPort memory which are defined by an **ADDRESS** parameter in each DualPort variable's declaration. The PLC program must use addresses corresponding to the **ADDRESS** parameter in the variable declaration.

In addition to DualPort variable declarations, a **SIMATIC DUALPORT HEADER** declaration provide information about the organisation of dualport memory and some DualPort system variables used for communications between PAM and the PLC.

14.2 GENERAL CONCEPT FOR EXCHANGING VARIABLES

There are two types of DualPort variables: input variables and output variables. Within each category, a DualPort variable may be any of the standard variable classes (see paragraph 3.6). When referring to any DualPort variable, direction (input or output) is always with respect to PAM. Therefore, a DualPort input variable is an input to PAM and a DualPort output variable is a PAM output. DualPort variables must be declared in the PAM application program. The general concept for exchanging variable values via the DualPort is summarised below.

14.2.1 INPUT WORD VARIABLES

The PLC places an input word variable value into it's designated cell (as specified by the **ADDRESS** parameter in the variable declaration) in the Word Inputs area of DualPort, and places the relative address of the cell whose value has changed onto the Input FIFO. This address is the displacement in bytes from the DualPort top address to the designated cell. PAM interrogates the Input FIFO and updates it's internal processes at a frequency determined by the **PERIOD** parameter in the **SIMATIC DUALPORT HEADER** declaration.

14.2.2 INPUT FLAG VARIABLES

The PLC places an input flag variable, into it's designated cell (as specified by **ADDRESS** parameter in variable declaration) in the Scanned Inputs area of DualPort. PAM scans the Scanned Inputs area and updates it's internal processes at a frequency determined by the **PERIOD** parameter in the **SIMATIC DUALPORT HEADER** declaration. The bit position must be specified for flag variables.

14.2.3 OUTPUT VARIABLES

Whenever the value of a DualPort output variable changes, PAM places the new value into it's designated cell (as specified by the **ADDRESS** parameter in the variable declaration) in the Output Values area, and places the relative address of the cell whose value has changed onto the Output FIFO. This address is the displacement in bytes from the DualPort top address to the designated cell. The PLC must interrogate the Output FIFO for output variable changes and update it's status and processes accordingly.

14.3 DUALPORT DESCRIPTION

14.3.1 DUALPORT STRUCTURE

Figure 14-1 illustrates partitioning of dual port memory into the functional areas utilised by the DualPort. The function of each component of the DualPort is described in the following paragraphs. Dual port memory maximum size is 4096 bytes

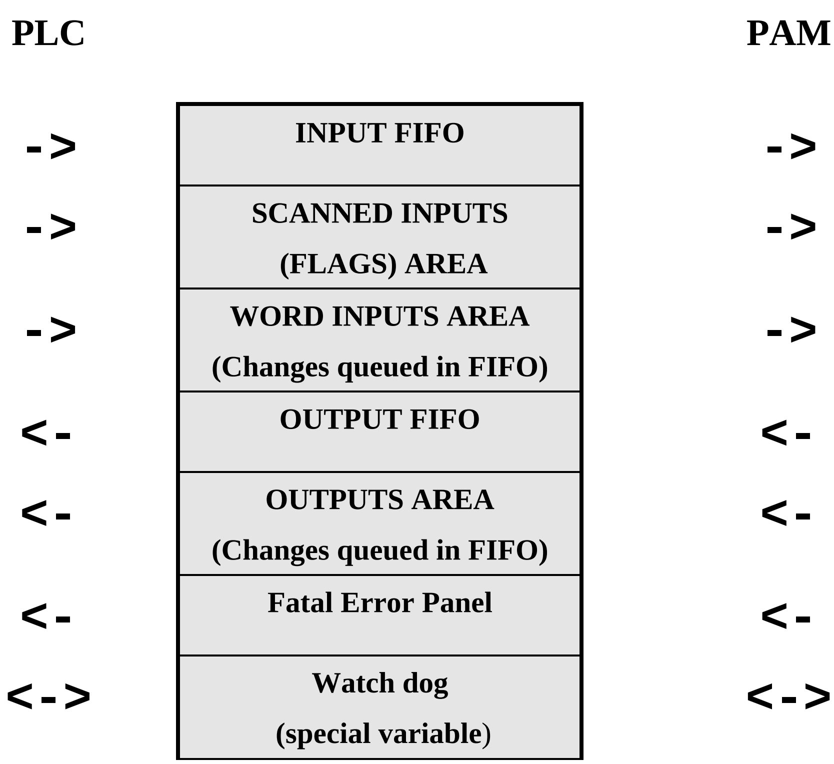


Figure 14-1 DualPort Memory Partitioning

14.3.1.1 DUALPORT CELL

The size of a DualPort cell is 16 bits (2 bytes). The PLC's data path may be 8 or 16 bits wide depending on the model PLC used. PAM should access the DualPort byte by byte if the PLC data path is 8 bits; otherwise, PAM normally accesses the DualPort as 16 bit words.

14.3.2 INPUT AND OUTPUT FIFOS

Figure 14-2 illustrates the structure of both the Input and Output FIFOs. The Input FIFO (including HEAD and TAIL pointer locations) is limited to 130 cells. It is the area of DualPort where the PLC places relative addresses of input word variables whose values it has changed. HEAD and TAIL pointers are manipulated by the PLC and PAM respectively when writing and reading from the Input FIFO. HEAD and TAIL are 8 bits wide.

The Output FIFO is the area of the DualPort where PAM places relative addresses of DualPort output variables whose values it has changed. It functions in a manner similar to the Input FIFO previously described.

0 254	128 Blocks (max.)	(128 Cells)	1 255
	HEAD	not used	write pointer
	TAIL	not used	read pointer

Figure 14-2 Input and Output FIFO Structure

14.3.3 SCANNED INPUTS AREA

The Scanned inputs area is the segment of the DualPort where DualPort Input Flag variables reside. The Scanned Inputs Area is defined in size and address by the Input Flag definitions listed in the PAM application. The user must give an DualPort relative address for each dualport variable. Flag variables may configured one variable per byte, or multiple flag variables may be assigned to individual bit positions in a byte.

The Flag Inputs area, including unused portions between variables, is entirely scanned during the given period (**PERIOD** in the Simatic DualPort Header Declaration). For example, if the area is 300 bytes and **PERIOD** = 10, at each PAM cycle 30 bytes (15 cells) are scanned. For most efficient operation, the user should avoid gaps between Flag Input variables.

14.3.4 WORD INPUTS AREA

The Word Inputs area is a segment of the DualPort where DualPort Input Word variables reside. The Words Input Area begins.

14.3.5 OUTPUTS AREA

The Outputs area is a segment of the DualPort where DualPort Output variables reside. The Outputs Area begins.

14.3.6 FIFO READING AND WRITING PROTOCOL

i The SIMATIC S5 DRIVER FOR PAM (refer to technical manual 006.8023.B) makes the interface easier on the PLC side and eliminates the requirement for detailed knowledge of the PAM /SIMATIC interface.

Head and Tail are called pointers, but they are block indexes (0,2,4, ...254). Block size is equal to cell size (two bytes).

14.3.6.1 WRITING INTO INPUT FIFO

The following principle is used by the PLC when writing into the Input FIFO:

MAX_INPUT_FIFO_SIZE = FIFO_IN_HEAD - FIFO_IN_START(refer to DualPort declaration)

but MAX_INPUT_FIFO_SIZE max. value is 256

```
1) compute new HEAD
   new_head = HEAD + 2
2) testing for end of FIFO
   if new_head >= MAX_INPUT_FIFO_SIZE
      THEN new_head = 0
3) testing for FIFO full
   if new_head = TAIL
      THEN FIFO is full !
4) compute write address into FIFO
   P_write = TOP_DUALPORT_ADDRESS + FIFO OFFSET +HEAD
5) fill the block at P_write address
   *P_write = dualport address of modified variable
6) update HEAD
   HEAD = new_head
```

14.3.7 READING FROM OUTPUT FIFO

The following principle is used by the PLC when reading from the Output FIFO:

MAX_OUTPUT_FIFO_SIZE = FIFO_OUT_HEAD - FIFO_OUT_START(refer to DualPort declaration)

but MAX_OUTPUT_FIFO_SIZE max. value is 256

```
1) testing for FIFO empty
   if TAIL = HEAD
      THEN FIFO is empty !
2) compute bloc read address
   P_read = TOP_DUALPORT_ADDRESS + FIFO OFFSET +TAIL
3) read the bloc contents at P_read address
   variable address = *P_read
4) update TAIL
   new_tail = TAIL + 2
   if new_tail >= MAX_OUTPUT_FIFO_SIZE
      THEN new_tail = 0
   TAIL = new_tail
```

14.3.8 FATAL ERROR PANEL

This segment of the DualPort is reserved for communicating to the host PLC the occurrence of a fatal error which prohibits PAM from continuing normal executing the application. Refer to paragraph 12.3 for details on the Fatal Error Panel. The Fatal Error Panel occupies seven cells located above the WATCH_DOG. For example, if WATCH_DOG is located at #0FFE (# is a prefix indicating Hexadecimal for the PAM compiler), the Fatal Error Panel is located at #0FFE - #E = #FF0.

14.3.9 WATCHDOG

This cell is reserved for a watchdog timer function implemented by PAM and an external host controller (see paragraph 14.5).

14.4 SYNCHRONISATION AND INITIALISATION UPON START-UP

This paragraph describes the recommended sequence of operations to be performed by PAM and a host controller (SIMATIC PLC) upon power-up to initialise the DualPort. The following three pre-declared DualPort Flag variables (see paragraph 14.6) are used during the start-up sequence:

DUALPORT_READY	Output flag which indicates the DualPort is ready for use in application execution.
HOST_READY	Input flag used by host to indicate it has completed initialisation of flag and word input variables in DualPort.
PAM_READY	An input/output flag used upon start-up by PAM and the host controller for synchronisation.

Synchronisation (see Table 14-1) is initiated by the host controller setting **PAM_READY** = 0, followed by a series of manipulations of the **PAM_READY** flag by PAM and the host controller. Upon completion of synchronisation, the host controller initialises its DualPort input variables, then PAM reads them. Two time-outs, **MASTER_READY_TIMEOUT** and **MASTER_CONFIGURATION_TIMEOUT** are associated with completion of this sequence (see paragraphs 14.4.1 and 14.4.2).

HOST CONTROLLER ACTION	PAM ACTION
	0 → **HOST_READY** initialise FIFOs reset flag and word outputs
0 → **PAM_READY**	wait for **PAM_READY** = 0 (with timeout = **MASTER_READY_TIMEOUT**) 1 → **PAM_READY**
wait for **PAM_READY** = 1 0 → **PAM_READY**	wait for **PAM_READY** = 0 (with timeout = **MASTER_READY_TIMEOUT**) 1 → **PAM_READY**

HOST CONTROLLER ACTION	PAM ACTION
wait for **PAM_READY** = 1 0 → **PAM_READY** initialise all flag and word inputs (direct write without use of Input FIFO) 1 → **HOST READY**	wait for **HOST_READY** = 1 (with timeout = **MASTER_CONFIGURATION_TIMEOUT**) read initial values of all flag and word inputs (direct read without use of Input FIFO). 1 → **DUALPORT_READY**. start updating DualPort outputs start handling watchdog evaluate equations with linked outputs execute power-up actions begin normal task execution
wait for **DUALPORT_READY** = 1 (with timeout = **MASTER_CONFIGURATION _TIMEOUT)** start reading Output FIFO start handling Watchdog	

Table 14-1 Synchronisation and Initialisation Sequence

14.4.1 MASTER_READY_TIMEOUT

ERROR CODE

[0780F020] Real Time Fatal Error HOST NOT READY

CAUSE

Synchronisation failed, PLC does not acknowledge PAM_READY during the interval specified by **MASTER_READY_TIMEOUT** parameter in the DualPort Header declaration (see paragraph 14.6).

PAM ACTION

- initialisation aborted, PAM stops and must then be restarted by performing a hardware reset.
- error code stored in fatal error panel.

14.4.2 MASTER_CONFIGURATION_TIMEOUT

ERROR CODE

[0780F020] Real Time Fatal Error HOST NOT READY

CAUSE

PLC does not set the **HOST_READY** flag within interval specified by **MASTER_CONFIGURATION_TIMEOUT.**

PAM Action

- Initialisation aborted, PAM stops and must then be restarted by performing a hardware reset.
- Error code stored in fatal error panel.

14.5 Watchdog

The DualPort register WATCH_DOG is utilised by PAM and the host PLC to verify that each is still active (alive). A **MASTER_INACTIVITY_TIMEOUT** (see paragraph 14.5.1.1) occurs when PAM determines that the host PLC is no longer alive.

14.5.1 Implementing a Watchdog Timer Function

PAM and the host PLC implement a watchdog timer utilising the flag variable **WATCH_DOG** (see Figure 14-3) as illustrated in the following example:

WATCHDOG (even byte)	not used

Figure 14-3 WATCH_DOG Flag Variable

```
PLC action :   ( at each PLC cycle )
IF WATCH_DOG = 0
   THEN (* OK *) write 0xFF into WATCH_DOG.
   ELSE if WATCH_DOG stay at 0xFF for more than 200 ms :
                               PAM is not running.
PAM action : ( each 10 PAM basic cycles  )
IF WATCH_DOG = 0xFF
      THEN (* OK *) write 0 into WATCH_DOG.
      ELSE if WATCH_DOG stay at 0 for more than
master_inactivity_timeout  ms :
                                PLC is not running.
```

14.5.1.1 MASTER_INACTIVITY_TIMEOUT

Error

[0780E01E] Real Time Error DUALPORT WATCH DOG ERROR

Cause

The host PLC has not written into **WATCH_DOG** for an interval exceeding **MASTER_INACTIVITY_TIMEOUT** parameter of DualPort declaration.

PAM Action

- The DualPort ? error becomes true (**DualPort ? error** = **true**).

- PAM stops acknowledging the **watch_dog** to the PLC host.
- Application continues to be executed, but PAM stops scanning the DualPort inputs.

14.6 DUALPORT DECLARATION

The ***DualPort*** object is pre-declared, so it need not to be declared in the application. The SIMATIC dual port header declaration is as follows:

DECLARATION SYNTAX

The DualPort header defines a part of the DualPort mapping (FIFOs and system variables).

```
DUAL_PORT
        START_ADDRESS            = <start address> ;
        LENGTH                   = <length>;
        FIFO_IN_START            = <cell address>;
        FIFO_IN_HEAD = <cell address> ;
        FIFO_IN_TAIL             = <cell address> ;
        FIFO_OUT_START           = <cell address>;
        FIFO_OUT_HEAD            = <cell address> ;
        FIFO_OUT_TAIL            = <cell address> ;
        WATCH_DOG                = <cell address> ;
        PAM_READY                = <byte address> BIT <bit address> ;
        DUALPORT_READY           = <byte address> BIT <bit address> ;
        HOST_READY               = <byte address> BIT <bit address> ;
        PERIOD                   = <scanning period> ;
        MASTER_READY_TIMEOUT             = <ready timeout [ms]>   ;
        MASTER_CONFIGURATION_TIMEOUT     = <config. timeout [ms]> ;
        MASTER_INACTIVITY_TIMEOUT        = <inactive timeout [ms]> ;
END
```

< start address> : (NA) Relative address of the first cell (0).

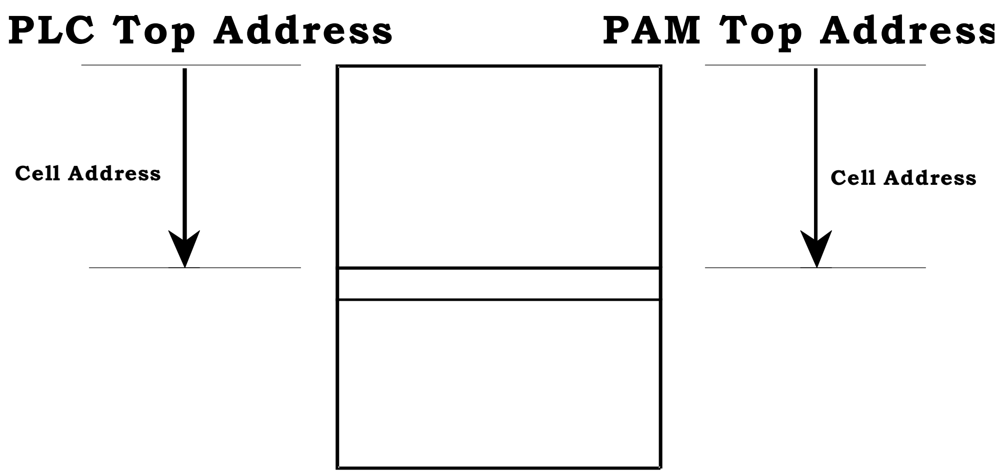
<length> : (NA) size in byte of dualport hardware (4096).

< start address> and <length> are given for user information only, values other than 0 and 4096 are not recommended !

< cell address> : (NA) DualPort cell address in bytes (must be an even address).

< byte address> : (NA) DualPort address in bytes.

DualPort byte and cell addresses are RELATIVE addresses (offset in bytes/cells between the physical address of the first byte/cell and the considered byte/cell).



Cell address must be relative because the TOP ADDRESS of PAM or PLC is not absolutely the same value (PAM Top Address is today #700000).

<bit address> : bit position in the addressed byte (0..7)

<scanning period> : defines number of PAM basic cycles (1, 5, 10, 20, 50) between sampling of all DualPort inputs.

<ready timeout> : (RO) time-out in msec. used when for waiting synchronisation with master. ***NONE*** *deactivates the timeout.*

<config. timeout> ; (RO) timeout in msec. used during initialisation phase. ***NONE*** *deactivates the timeout.*

<inactive timeout> : (RW) timeout value in msec used during execution phase. The PLC master is considered inactive when it has not responded via the Watchdog within the inactive timeout interval. ***NONE*** *deactivates the timeout.*

FUNCTIONS

The inquire functions available for the Simatic ***DualPort*** include:

– **error**

This Boolean inquire function is true if an error condition exists in the DualPort.

Syntax:

DualPort ? error

Examples:

```
IF DualPort ? error THEN ...
EXCEPTION DualPort ? error SEQUENCE SEQ_DualportErrorHandling ;
```

– **error_code (Boolean)**

This Boolean inquire function is true if the error code parameter matches the error state of the DualPort.

Syntax

DualPort ? error_code(<error code

<error code> : the specific error code to be tested for. The Include file "porterr.sys" contains DualPort error code definitions.

Examples:

```
IF DualPort ? error_code(DUALPORT_MASTER_STOPPED)THEN ...
EXCEPTION DualPort ? error_code(DUALPORT_MASTER_STOPPED) SEQUENCE
....
```

– **error_code (Numerical)**

>)This inquire function returns the current DualPort error code.

Syntax

DualPort ? error_code

Example

```
IF DualPort ? error_code = DUALPORT_MASTER_OK THEN ...
```

SAMPLE DECLARATION

```
DUAL_PORT
     START_ADDRESS       =    0 ;
     LENGTH              =  4096 ;
     FIFO_IN_START       =     0 ;
     FIFO_IN_HEAD        = #0100 ;
     FIFO_IN_TAIL        = #0102 ;
     FIFO_OUT_START      = #0800 ;
     FIFO_OUT_HEAD       = #0900 ;
     FIFO_OUT_TAIL       = #0902 ;
     WATCH_DOG           = #0FFE ;
/* Fatal error panel is located at watch_dog location - #0E   */
/* so last output can be located at watch_dog - #10  = #0FEE  */
     PAM_READY           = #0FEE BIT 3 ;
     DUALPORT_READY      = #0FEE BIT 4 ;
     HOST_READY          = #0104 BIT 4 ;
     PERIOD              = 10 ;
     MASTER_READY_TIMEOUT         = 20000;   /* [ms]      */
     MASTER_CONFIGURATION_TIMEOUT = NONE;    /* infinite */
   MASTER_INACTIVITY_TIMEOUT    = 500;     /* [ms]     */
END
```

The beginning and end of the Scanned Inputs, Word Inputs and Word Output areas are determined by the lowest and highest addresses used in the corresponding classes of DualPort variable declarations.

14.6.1 DUALPORT PARAMETER ACCESS

DualPort parameters may be read or modified, subject to each parameter's access level using the standard parameter access syntax (see paragraph 4.3.4).

EXAMPLES

```
/* wait till initial value is elapsed */
WAIT_TIME( DualPort:MASTER_INACTIVITY_TIMEOUT);
```

```
/* reduce the timeout value */
DualPort:MASTER_INACTIVITY_TIMEOUT <- 500;
```

14.7 DUALPORT VARIABLES

14.7.1 GENERAL DECLARATION SYNTAX

The general declaration syntax for DualPort variables is illustrated below. The specific declaration syntax and functions for each type and class of DualPort variable are listed in subsequent paragraphs of this section.

<direction > <class> <identifier> **;**
[{ **NUMBER** **=** <number> ¦
NODES_GROUP = <nodes group identifier> } ;]
[**RESERVED** **=** <reserved> ;]
ADDRESS **=** <address> **;**
END

<direction> : (NA) specifies the direction of exchange (input or output).

<class> : (NA) specifies the class of the variable (flag or word).

<identifier>: (NA) specifies the variable name.

*<reserved>: (NA) DualPort variable space reservation, must be greater than or equal to <number> or to the <number> given in the **NODES_GROUP** declaration.*

*If the **RESERVED** declaration is omitted, <number> is used for the reservation. If the **NUMBER** declaration is omitted or if <number> = 1, the variable is single, otherwise the variable is multiple.*

<number> : (NA) defines the number of item for a multiple variable.

<nodes group identifier> : (NA) the name of a node group. Used to define the number of duplicates in a multiple variable using the <number> parameter in the nodes group declaration.

<address>: (NA) the variable address. The nature of <address> depends of the variable type.

14.7.2 DUALPORT INPUT FLAG VARIABLE

SYNTAX

DUALPORT_IN FLAG_VAR <identifier> ;
  [{ **NUMBER** = <number> ¦
  NODES_GROUP = <nodes group identifier> } ;]
  [**RESERVED** = <reserved> ;]
  ADDRESS = <DualPort byte address> **[BIT** = <bit address>];
END

<DualPort byte address>: relative address of the DualPort byte where the variable is located.

<bit address>: bit rank (0..7) in the byte where the flag variable is located. If the **BIT** *declaration is omitted, <bit address> = 0.*

FUNCTIONS

FUNCTION NAME	SET/ INQUIRE	PARAMETERS	RETURN VALUE
<- (read value)	INQUIRE	-	BOOLEAN

A cell can hold single flags (contiguous or not) OR

multiple flags starting at bit 0 of the even byte of the cell !

Do not mix single flags with multiple flags in the same cell !

For best performance avoid unused cells between flag input variables because the entire area is scanned !

14.7.3 DUALPORT OUTPUT FLAG VARIABLE

SYNTAX

```
DUALPORT_OUT FLAG_VAR <identifier> ;
        [ { NUMBER              = <number> ¦
        NODES_GROUP = <nodes group identifier> } ; ]
        [RESERVED               = <reserved> ;]
        ADDRESS                 = <DualPort byte address> [BIT = <bit address>];
END
```

<DualPort byte address>: relative address of the DualPort byte where the variable is located.

<bit address>: bit rank (0..7) in the byte where the flag variable is located. If the ***BIT*** *declaration is omitted, <bit address> = 0.*

FUNCTIONS

FUNCTION NAME	SET/ INQUIRE	PARAMETERS	RETURN VALUE
<- (assign value)	SET	- boolean expression	-
invert	SET	-	-
reset	SET	-	-
set	SET	-	-
<- (read value)	INQUIRE	-	BOOLEAN

For output flag variables it is possible to mix single variables with multiple variables.

14.7.4 DUALPORT INPUT WORD VARIABLE

SYNTAX

DUALPORT_IN WORD_VAR <identifier> ;
[{ **NUMBER** = <number> ¦
NODES_GROUP = <nodes group identifier> } ;]
[**RESERVED** = <reserved> ;]
ADDRESS = <DualPort cell address>;
END

<DualPort cell address>: address of the DualPort cell where the variable is located (must be an even address).

FUNCTIONS

FUNCTION NAME	SET/ INQUIRE	PARAMETERS	RETURN VALUE
<- (read value)	INQUIRE	-	INTEGER

It is very important to put all the Input Word variables in the same memory area. The address of the first INPUT WORD defines the beginning of the Word Input area. The address of the last INPUT WORD define the end of the Word Input area. We recommend that the user define the variables in order of ascending address.

Word inputs are 16 bits signed integer !

14.7.5 DUALPORT OUTPUT WORD VARIABLE

SYNTAX

DUALPORT_OUT WORD_VAR <identifier> ;
[{ **NUMBER** = <number> ¦
NODES_GROUP = <nodes group identifier> } ;]
[**RESERVED** = <reserved> ;]
ADDRESS = <DualPort cell address> ;
END

<DualPort cell address>: address of the DualPort cell where the variable is located (must be an even address).

FUNCTIONS

FUNCTION NAME	SET/ INQUIRE	PARAMETERS	RETURN VALUE
<- (assign value)	SET	- numerical expression	-
<- (read value)	INQUIRE	-	INTEGER

Word outputs are 16 bit signed integers !

14.7.6 DUALPORT VARIABLES DECLARATION EXAMPLES

```
#define MaxHead 48
   DUALPORT_IN FLAG_VAR DFI_MachRunning ;
      ADDRESS = #104 BIT 0 ;
   END

   DUALPORT_IN FLAG_VAR DFI_MachEmergency ;
      ADDRESS = #105 BIT 3 ;    // cell #104 bit 11
   END

   DUALPORT_IN FLAG_VAR DFI_MachHeadReady ;
      NODES_GROUP = NGR_PrintHeads ;
      RESERVED = MaxHead ;
      ADDRESS = #106 ;    // size is 48 / 8 = 6 bytes
   END

   DUALPORT_IN FLAG_VAR DFI_MachHeadStopped ;
      NODES_GROUP = NGR_PrintHeads ;
      RESERVED = MaxHead ;
      ADDRESS = #10C ;
   END


   DUALPORT_IN WORD_VAR DWI_MachPressure ;
      ADDRESS = #120 ;
   END

   DUALPORT_IN WORD_VAR DWI_MachHeadSpeed ;
      NODES_GROUP = NGR_PrintHeads ;
      RESERVED = MaxHead ;
      ADDRESS = #122 ;    // size is 48 * 2 = 96 bytes
   END

   DUALPORT_IN FLAG_VAR DFI_MachHeadTorque ;
      NODES_GROUP = NGR_PrintHeads ;
      RESERVED = MaxHead ;
      ADDRESS = #182 ;
   END


   DUALPORT_OUT FLAG_VAR DFO_MachStopped ;
      ADDRESS = #904 BIT 5 ;
   END

   DUALPORT_OUT FLAG_VAR DFO_MachOk ;
      ADDRESS = #905 BIT 5;     // cell #904 bit 13
   END

   DUALPORT_OUT FLAG_VAR DFO_MachHeadBusy ;
      NODES_GROUP = NGR_PrintHeads ;
      RESERVED = MaxHead ;
      ADDRESS = #906 ;    // size is 48 / 8 = 6 bytes
   END

   DUALPORT_OUT FLAG_VAR DFO_MachHeadFastSpeed ;
      NODES_GROUP = NGR_PrintHeads ;
      RESERVED = MaxHead ;
      ADDRESS = #90C ;
   END


   DUALPORT_OUT WORD_VAR DWO_MachCurrentPressure ;
      ADDRESS = #920 ;
   END
```

```
DUALPORT_OUT WORD_VAR DWO_MachHeadPosition ;
   NODES_GROUP = NGR_PrintHeads ;
   RESERVED = MaxHead ;
   ADDRESS = #922 ;     // size is 48 * 2 = 96 bytes
END

DUALPORT_OUT FLAG_VAR DFO_MachHeadLevel ;
   NODES_GROUP = NGR_PrintHeads ;
   RESERVED = MaxHead ;
   ADDRESS = #982 ;
END
```

15 RS422 SERIAL COMMUNICATIONS CHANNEL

15.1 INTRODUCTION

The RS422 serial communication channel (here-after called the “RS422 Port”)allows the exchange of variable values between PAM and host equipment (called RS422 Master) connected to the RS422 port. The communication protocol is command oriented and works in full duplex mode, permitting simultaneous exchange of RS422 variable values in both direction.

This chapter deals only with the top level interface between the application and the Serial-Line Port within PAM. Technical Manual 006.8021 (PAM Serial Line Protocol) provides details on the operation and use of the Serial-Line Port from both the PAM and RS422 Master perspectives.

15.2 RS422 PORT SOFTWARE ROUTINES

A group of C language routines (called “PAM RS422 utilities”) which may be integrated into the user’s application provide all the functions needed for configuration and exchange of variable values during PAM application execution. The PAM RS422 utilities call low level functions in order to send and receive characters through the RS422 serial communication port and to update user variables. Protocol details at board level and application levels are handled by the PAM RS422 utilities.

i | Details on the PAM RS422 utilities are found in document number 006.8021.

15.3 SYSTEM START-UP WITH AN INACTIVE RS422 MASTER

PAM allows an application containing an RS422 DualPort to start application execution with the RS422 Master inactive or the RS422 line disconnected. In this situation, PAM continues functioning by considering the RS422 variables as internal variables. The PamDisplay shows error code [0780E039].

At the moment the RS422 line is connected and communications is started, PAM acknowledges the communication request, performs the configuration phase, then begins sending RS422 variables changes. An error cancelled message is generated to turn the PamDisplay back to scrolling application name and version.

15.4 RS422 PORT DECLARATION

The **RS422 PORT** object is pre-declared, so it need not be declared in the application. The declaration is divided into three sequential parts :

- RS422 Port Header
- List of individual RS422 variable declarations
- **END_PORT** key word

15.4.1 RS422 PORT HEADER

SYNTAX

```
RS422 PORT
    SPECS
        PERIOD          =  <scanning period> ;
        MASTER_READY_TIMEOUT            = < ready timeout> ;
        MASTER_CONFIGURATION_TIMEOUT    = < config timeout> ;
        MASTER_INACTIVITY_TIMEOUT       = < inactive timeout > ;
        PAM_WATCHDOG_MESSAGE_PERIOD     = <watchdog period> ;
    END_SPECS

            <individual RS422 variable declarations>

END_PORT
```

<scanning period> : scanning period (1, 5, 10, 20, 50 PAM basic cycles) used for all RS422 variables.

*<ready timeout > : (RO), timeout value in msec used by PAM when waiting for synchronisation with RS422 Master. **NONE** deactivates the timeout.*

*<config timeout > : (RO), timeout value in msec used during configuration phase, when waiting for master commands or acknowledgements. **NONE** deactivates the timeout.*

*<inactive timeout > : (RW), timeout value in msec used during execution phase. The RS422 master is considered inactive when it has not responded via the Watchdog within the inactive timeout interval. **NONE** deactivates the timeout.*

<watchdog period> : (RW) specifies the interval in msec at which PAM sends the watchdog message.

FUNCTIONS

The inquire functions available for the ***RS422 Port*** are :

– **error**

This boolean inquire function is true if the RS422 Port has an error condition active.

Syntax: **RS422Port ? error**

Examples

```
IF RS422Port ? error THEN ...
EXCEPTION RS422Port ? error SEQUENCE SEQ_RS422portErrorHandling ;
```

– **error_code (boolean)**

This Boolean inquire function is true if the parameter error code matches the error state of the RS422 Port.

Syntax: **RS422Port ? error_code(**<error code>**)**

<error code> : the specific error code to be tested for. The Include file "porterr.sys" contains RS422 Port error code definitions.

Examples

```
IF RS422Port ? error_code(RS422PORT_DICONNECTED )THEN ...
EXCEPTION RS422Port ? error_code(RS422PORT_DICONNECTED) SEQUENCE ....
```

– **Error_code (Numeric)**

This inquire function returns the current error code for the RS422 Port.

Syntax: **RS422Port ? error_code**

Example

```
IF (RS422Port ? error_code = RS422PORT_CONNECTED)THEN ...
```

DECLARATION EXAMPLE

```
RS422 PORT
   SPECS
      PERIOD = 20;
      MASTER_READY_TIMEOUT                  = 20000;
      MASTER_CONFIGURATION_TIMEOUT    =  5000;
      MASTER_INACTIVITY_TIMEOUT             =  2000;
      PAM_WATCHDOG_MESSAGE_PERIOD     = 800 ;
   END_SPECS
```

15.4.2 PARAMETER ACCESS

RS422 Port parameters may be read or modified, subject to each parameter's access level using the standard parameter access syntax (see paragraph 4.3.4).

EXAMPLES

```
/* wait till initial value is elapsed */
WAIT_TIME( RS422Port:MASTER_INACTIVITY_TIMEOUT);
/* reduce the timeout value */
RS422Port:MASTER_INACTIVITY_TIMEOUT <- 500;
```

15.4.3 MASTER_INACTIVITY_TIMEOUT

The **MASTER_INACTIVITY_TIMEOUT** parameter is a 32 bits unsigned integer. So the maximum usable value is 4,294,967•10^3 ms.

> i The maximum value in the PAM SYSTEM 2.2 release was 65535 ms.

15.4.4 PAM_WATCHDOG_MESSAGE_PERIOD

PAM sends the watch dog message when there is no variable change to communicate to the RS422 Master within an interval specified by the **PAM_WATCHDOG_MESSAGE_PERIOD** parameter. This parameter permits the user to select the interval at which PAM sends the watchdog message.

15.4.4.1 RELATIONSHIP TO RS422_WATCHDOG_TIMEOUT

The value of **PAM_WATCHDOG_MESSAGE_PERIOD** must be smaller than the value of the **RS422_WATCH_DOG_TIMEOUT** (defined in module USER_PAR.H of the RS422 utilities). We recommend using :

PAM_WATCHDOG_MESSAGE_PERIOD `<=  (RS422_WATCH_DOG_TIMEOUT / 2)`

15.5 RS422 PORT TIME-OUTS BEHAVIOUR

15.5.1 MASTER_READY_TIMEOUT

ERROR

[0780E039] Real Time Error RS422 : comm. disconnected

CAUSE

PAM did not receive communication request from RS422 master during time interval corresponding to **MASTER_READY_TIMEOUT**.

> i PAM raises this error only the first time it is waiting for a communications request.

PAM ACTION

- PAM continue functioning and waiting for master communication request.

15.5.2 MASTER_CONFIGURATION_TIMEOUT

ERROR

[0780E039] Real Time Error RS422 : comm. disconnected

CAUSE

The RS422 Master did not send next configuration command during time interval corresponding to **MASTER_CONFIGURATION_TIMEOUT**

PAM ACTION

- PAM abort configuration and waits for the to master request communication again

15.5.3 MASTER_INACTIVITY_TIMEOUT

ERROR

[0780E035] Real Time Error RS422 : no user activity

CAUSE

Master does not send any message for an interval exceeding **MASTER_INACTIVITY_TIMEOUT** parameter in RS422 Port declaration.

PAM ACTION

- The *RS422Port ? error* becomes true
- Application continue to be executed but with all RS422 variables handled as internal variables.
- PAM abort normal handling of RS422 Port and waits for the master to request communication again.

When a lack of user activity is due to a communication problem, a warning message to the PAM Display indicating the kind of communication problem is also generated.

i | The ***RS422Port ? error*** becomes true to signal the disconnection at application level

15.6 RS422 VARIABLES DECLARATION

The general declaration syntax for RS422 Port variables is listed below. The declaration syntax and functions for each type and class of RS422 Port variable are listed in subsequent paragraphs of this section. RS422 variables declarations must be enclosed between the keywords **RS422 PORT** and **END_PORT** in the RS422 Port declaration.

GENERAL SYNTAX

<direction> <class > <identifier> **;**
    [{ **NUMBER** **=** <number> ¦
    NODES_GROUP = <nodes group identifier> } ;]
[**END]**

<direction> : (NA) specifies the direction of exchange (input or output).

<class > : (NA) specifies the variable class (flag, word, long, real).

<identifier>: (NA) specifies the variable name.

<number> : (NA) defines the number of item in a multiple variable.

<nodes group identifier>: (NA) the name of a nodes group used to define the number of duplicates in a multiple variable using the <number> parameter in the ***NODES_GROUP*** *declaration.*

15.6.1 RS422 INPUT FLAG VARIABLE

SYNTAX

INPUT FLAG_VAR <identifier> ;
[{ **NUMBER** = <number> ¦
NODES_GROUP = <nodes group identifier> } ;]
[**END**]

FUNCTIONS

FUNCTION NAME	SET/ INQUIRE	PARAMETERS	RETURN VALUE
<- (read value)	INQUIRE	-	BOOLEAN

15.6.2 RS422 OUTPUT FLAG VARIABLE

SYNTAX

OUTPUT FLAG_VAR <identifier> ;
[{ **NUMBER** = <number> ¦
NODES_GROUP = <nodes group identifier> } ;]
[**END**]

FUNCTIONS

FUNCTION NAME	SET/ INQUIRE	PARAMETERS	RETURN VALUE
<- (assign value)	SET	- boolean expression	-
invert	SET	-	-
reset	SET	-	-
set	SET	-	-
<- (read value)	INQUIRE	-	BOOLEAN

15.6.3 RS422 INPUT WORD VARIABLE

SYNTAX

INPUT WORD_VAR <identifier> ;
[{ **NUMBER** = <number> ¦
NODES_GROUP = <nodes group identifier> } ;]
[**END**]

FUNCTIONS

FUNCTION NAME	SET/ INQUIRE	PARAMETERS	RETURN VALUE
<- (read value)	INQUIRE	-	INTEGER

WORD inputs are 16 bits signed integers !

15.6.4 RS422 OUTPUT WORD VARIABLE

SYNTAX

OUTPUT WORD_VAR <identifier> ;
[{ **NUMBER** = <number> ¦
NODES_GROUP = <nodes group identifier> } ;]
[**END**]

FUNCTIONS

FUNCTION NAME	SET/ INQUIRE	PARAMETERS	RETURN VALUE
<- (assign value)	SET	- numerical expression	-
<- (read value)	INQUIRE	-	INTEGER

WORD outputs are 16 bits signed integers !

15.6.5 RS422 INPUT LONG VARIABLE

SYNTAX

INPUT LONG_VAR <identifier> ;
[{ **NUMBER** = <number> ¦
NODES_GROUP = <nodes group identifier> } ;]
[**END**]

FUNCTIONS

FUNCTION NAME	SET/ INQUIRE	PARAMETERS	RETURN VALUE
<- (read value)	INQUIRE	-	INTEGER

LONG inputs are 32 bits signed integers !

15.6.6 RS422 OUTPUT LONG VARIABLE

SYNTAX

OUTPUT LONG_VAR <identifier> ;
[{ **NUMBER** = <number> ¦
NODES_GROUP = <nodes group identifier> } ;]
[**END**]

FUNCTIONS

FUNCTION NAME	SET/ INQUIRE	PARAMETERS	RETURN VALUE
<- (assign value)	SET	- numerical expression	-
<- (read value)	INQUIRE	-	INTEGER

LONG outputs are 32 bits signed integers !

15.6.7 RS422 INPUT REAL VARIABLE

SYNTAX

INPUT REAL_VAR <identifier> ;
[{ **NUMBER** = <number> ¦
NODES_GROUP = <nodes group identifier> } ;]
[**END**]

FUNCTIONS

FUNCTION NAME	SET/ INQUIRE	PARAMETERS	RETURN VALUE
<- (read value)	INQUIRE	-	REAL

REAL inputs are 64 bits floating ! (IEEE 754-1985)

15.6.8 RS422 OUTPUT REAL VARIABLE

SYNTAX

OUTPUT REAL_VAR <identifier> ;
[{ **NUMBER** = <number> ¦
NODES_GROUP = <nodes group identifier> } ;]
[**END**]

FUNCTIONS

FUNCTION NAME	SET/ INQUIRE	PARAMETERS	RETURN VALUE
<- (assign value)	SET	- numerical expression	-
<- (read value)	INQUIRE	-	REAL

REAL outputs are 64 bits floating ! (IEEE 754-1985)

15.6.9 RS422 VARIABLES DECLARATION EXAMPLE

```
RS422 PORT
   SPECS
      PERIOD = 20 ;
      MASTER_READY_TIMEOUT   = 20000 ;
      MASTER_CONFIGURATION_TIMEOUT = 5000 ;
      MASTER_INACTIVITY_TIMEOUT =   2000 ;
      PAM_WATCHDOG_MESSAGE_PERIOD =  500 ;

   END_SPECS

   INPUT FLAG_VAR RFI_MachRunning ;   // single variable

   INPUT FLAG_VAR RFI_MachHeadReady ; // multiple variable with dynanic
      NODES_GROUP = NGR_PrintHeads ;  // configuration
   END


   INPUT FLAG_VAR RFI_MachHeadStopped ;     // multiple variable with
      NUMBER = 6 ;                          // static configuration
   END

   INPUT WORD_VAR RWI_MachPressure ;

   INPUT LONG_VAR RLI_MachHeadSpeed ;
      NUMBER  = 6 ;
   END

   INPUT REAL_VAR RRI_MachHeadFactor ;
      NUMBER  = 6 ;
   END

   OUTPUT FLAG_VAR RFO_MachStopped ;

   OUTPUT WORD_VAR RWO_MachCurrentPressure ;
   END

   OUTPUT LONG_VAR RLO_MachHeadPosition ;
      NODES_GROUP = NGR_PrintHeads ;
   END

   OUTPUT REAL_VAR RRO_MachHeadLevel ;
      NODES_GROUP = NGR_PrintHeads ;
   END

END_PORT
```

APPENDIX A

A RESERVED NAMES

ABORT
ABORT_SEQUENCE
ACCELERATION
ACTION
ACTIONS
ACTIVE
ADDRESS
AMPLIFIER
ANALOG_OUTPUT
APPLICATION
AUTOREPEAT
AXES_SET
AXIS
abs
absolute_move
acceleration
acos
anti_delay
asin
atan
BASIC_PAM_CYCLE
BINARY_INPUT
BINARY_OUTPUT
BIT
BITS
BLOC_SIZE
BOOLEAN
BREAKING
blink
CAM
CASE
COARSE_EDGE
COARSE_MOVE
COARSE_SPEED
COMMON
COMPARATOR
CONDITION
CONVERTER
CORRECTION_LEVEL
CORRECTION_SLOPE
CORRECTION_VALUE
CORRECTOR
COUNTER_INPUT
CURRENT
CYCLES
ceil
change_all_ratios
change_ratio
connect
connect_all
cos
cosh
D7SEG_OUTPUT
DC_MOTOR
DEBOUNCE
DEC
DEC
DECELERATION

DEFAULT_PERIOD
DEFAULT_SEQUENCE_WORKSPACE
DEFAULT_TASK_WORKSPACE
DELAY
DELAY_COMPENSATION
DERIVATOR
DESTINATION
DIGITAL_INPUT
DIGITAL_OUTPUT
DIGITS
DUALPORT
DUALPORT_IN
DUALPORT_OUT
DUALPORT_READY
DUAL_PORT
DUPL
DUPL_START
deceleration
delay_compensation
disable_stroke_limits
disactivate
disconnect
disconnect_all
display
EDGE_NONE
ELSE
ENCODER
END
END_ACTION
END_ACTIONS
END_BOOLEAN
END_CASE
END_DUALPORT
END_EVENTS
END_IF
END_LOOP
END_PORT
END_POWERON
END_ROUTINE
END_SEQUENCE
END_SPECS
END_TASK
ENTRY
EQUATION
ERROR
EVENTS
EXCEPTION
EXCEPTION_ENTRY
enable_stroke_limits
end_error
end_warning
error
error_code
exp
FALLING
FIFO_IN_HEAD
FIFO_IN_START
FIFO_IN_TAIL
FIFO_OUT_HEAD
FIFO_OUT_START
FIFO_OUT_TAIL
FILE
FINE_EDGE
FINE_MOVE
FINE_SPEED
FLAG_VAR
FONT

floor
GAIN
GAIN_SLOPE
generator_position
HEXA
HIGH
HOST_READY
IF
INC
INC
INITIAL_POSITION
INPUT
INPUT_AMPLITUDE
INPUT_OFFSET
INTERNAL
inquire_value
install_reference
invert
KEY_INPUT
LED_OUTPUT
LENGTH
LINKED_OUTPUT
LOCATION
LONG_VAR
LOOP
LOW
latched_dd_value
latched_d_value
latched_value
ln
log10
MASTER_CONFIGURATION_TIMEOUT
MASTER_INACTIVITY_TIME
MASTER_READY_TIMEOUT
MASTER_TIMEOUT
MODE
MULTI_COMPARATOR
message
modify_value
NAME
NEXT_PERIOD
NODE
NODES_GROUP
NONE
NUMBER
no_blink
OBJECT
OFFSET
OFFSET_SLOPE
ON
ONCE
ON_EVENT
ON_OFF
ON_REQUEST
ON_STATE
OPTIMIZE
OUTPUT
OUTPUT_AMPLITUDE
OUTPUT_OFFSET
output_amplitude
output_offset
PAM_ANALOG_OUTPUT
PAM_READY
PERIOD
PHASER
Pi
pipe_motionless

PMP_GENERATOR
PORT
POSITION
POSITION_PERIOD
POSITION_RANGE
POWERON
PULSES_PER_UNIT
position
power_off
power_on
RANGE
REAL_VAR
REFERENCE
relative_move
REMOVE_EXCEPTION
REPEAT
REPETITIVE
RESERVED
RESOLVER_OFFSET
REVERSE_ROUTINE
RISING
RS422
RS422PORT
RS422_WATCHDOG_TIMEOUT
ready
reference
relative_move
reset
run
SAMPLER
SENSOR
SEQUENCE
SINK
SMART_IO
SOURCE
SPECS
SPEED
ST1
ST1_CMD
START_ADDRESS
START_ADDRESS
start_correction
set
sin
sinh
speed
sqrt
STANDBY_VALUE
start
status
stop
stroke_limits
SUBSET
TASK
THEN
THROUGH_ZERO_REFERENCE
TIMEOUT
TIME_ORIGIN_COMPARE_MODE
TIME_ORIGIN_REFERENCE
TIME_ORIGIN_ROUTINE
TIME_ORIGIN_SLOPE
TMP_GENERATOR
TOGGLE
TORQUE
TRAVEL_SPEED
TRIGGER_INPUT
TRIGGER_MODE
TYPE

tan
tanh
through_zero_reference
travel_speed
trigger
triggered
trigger_off
UNITS_PER_VOLT
update_status
VALUE
VALUE_PERIOD
VALUE_RANGE
VME
WAIT
WAIT_TIME
WATCH_DOG
WORD_VAR
WORKSPACE
warning
XEQ_ACTION
XEQ_SEQUENCE
XEQ_TASK
ZERO
ZERO_POSITIONER
zero_position

APPENDIX B

B USER ERROR CODES

This appendix provides a listing and description of all system error and message codes produced by PAM. Refer to the paragraph titled "PAM Display Codes Intepretation" for a breakdown of the error message coding scheme.

The paragraph titled "Isolating Ring Errors" provides supplemental information on isolating some types of ring errors.

B.1 PAM DISPLAY CODES INTEPRETATION

The 8 hexadecimal digits of PAM message code are intepreted as follow:

O	**O**	**R**	**X**	**T**	**N**	**N**	**N**

Where:

OO indicates the origin of the message (00 for application message)

R provies additional information, (0 means non real time message and 8 means real time message).

X not used (it's value is 0).

T indicates the type of the message with:
- **M** message,
- **W** warning message,
- **K** end of warning (stored in the error list),
- **F** fatal error (stored in the error list),
- **E** error message (stored in the error list),
- **S** end of error (stored in the error list).

NNN indicates the message code number within the corresponding type (in hexadecimal).

B.2 ISOLATING RING ERRORS

For certain types of ring errors including CRC errors [error code 0608F00E] and loss of carrier [error codes 0680F00C and 0680F010], indicators on the Smart IO and ST1 peripherals provide supplemental information useful for isolating the fault.

i

In order for an ST1 or Smart IO peripheral to provide the indications illustrated in this appendix, the following are required:

On the ST1, bit 12 of STATD must be “1” in CMASKU.
ST1 firmware version must be V0590 or later.
Hardware for controlling optical ring is equipped with an ASIC board. The ASIC board contains a red and a yellow LED. The red LED indicates CRC error and the yellow LED indicates Carrier Fail error.

CRC ERRORS

Figure 2-1 illustrates the situation when a CRC error is detected at a Smart IO peripheral. The first Smart IO Node downstream of the optical ring defect displays “F-7” and it’s yellow and red LEDs are on, while Smart IO’s upstream and downstream display different indications.

Erreur! Nom de fichier incorrect.

Figure B-1 CRC Error detected at a Smart IO Node

Figure 2-2 illustrates the situation when a ST1 peripheral is the first peripheral downstream of the point of origin of a CRC error.

Erreur! Nom de fichier incorrect.

Figure B-2 CRC Error at ST1 Node

B.2.1 LOSS OF CARRIER ERROR

Figure 2-3 illustrates the situation for a loss of carrier at a Smart IO Node. The first Smart IO Node downstream of the point of carrier loss displays “F-6” and it’s yellow and red LEDs are on, while Smart IO’s upstream and downstream display different indications.

Erreur! Nom de fichier incorrect.

Figure B-3 Loss of Carrier at Smart IO Node

Figure 2-4 illustrates the situation when the loss of carrier occurs at an ST1 Node. Note that the display is different for ST1 nodes upstream, downstream and at the point of carrier loss.

Erreur! Nom de fichier incorrect.

Figure B-4 Loss of Carrier at ST1 Node

B.2.2 OPTICAL RING BROKEN AT STARTUP

Figure 2-5 illustrates the situation when a break in the optical ring is detected upon system startup. Note that peripherals upstream and downstream of the point where the optical ring is broken display different patterns.

Erreur! Nom de fichier incorrect.

Figure B-5 Optical Ring broken upon system startup

B.3 BOOT-UP ERROR CODES

[01 ... ...]	Boot Error

[01 xx **E 022]**	Boot Error	Sys
	Spoiled application in RAM. You are attempting to save a spoiled application from the RAM into the EEPROM. Download the application in RAM and try to save it again	

[01 xx **E 023]**	Boot Error	Flt
	CRC error in RAM. PAM detected a CRC error on your application in ram. Hardware problem, the RAM is defective.	

[01 xx **E 024]**	Boot Error	Sys
	Download Error. Download of your application was not successful. Try again.	

[01 xx **E 025]**	Boot Error	Sys
	Socapel System Error. The address between application (link address) and address where it must be doesn't match.	

[01 xx **E 027]**	Boot Error	App / Cfg
	Wrong Pamcomp Version. You can not download this application with this PAMEPROM version. Put an old version of PAMEPROM or compile your application with the new version of PAMCOMP.	

[01 xx **E 028]**	Boot Error	App / Cfg
	Wrong Pameprom Version. You can not download this application with this PAMEPROM version. Put a new version of PAMEPROM or compile your application with the old version of PAMCOMP.	

B.4 SYSTEM ERRORS

[02 . . . . . .]	Kernel System Error

[02 xx **E 001]**	Kernel System Error	App
	Socapel System Error. Not enough heap (mailbox allocation) Solution : -Reduce the number of task working simultaneously -or reduce the memory size allocated to workspaces.	

[02 xx **E 00F]**	Kernel System Error	App
	Socapel System Error. Not enough heap (task allocation) Solution : -Reduce the number of task working simultaneously -or reduce the memory size allocated to workspaces.	

[02 xx **E 011]**	Kernel System Error	App
	Socapel System Error. Not enough heap (process allocation) Solution : -Reduce the number of task working simultaneously -or reduce the memory size allocated to workspaces.	

[02 xx **E 013]**	Kernel System Error	App
	Socapel System Error. Not enough heap (signal allocation) Solution : -Reduce the number of task working simultaneously -or reduce the memory size allocated to workspaces.	

[02 xx **E 018]**	Kernel System Error	App
	Socapel System Error. Not enough heap (library allocation) Solution : -Reduce the number of task working simultaneously -or reduce the memory size allocated to workspaces.	

[02 xx **F 060]**	Kernel Error	App
	PAM basic cycle overran. Read part FATAL SYSTEM ERROR in your user's manual.	

[02 xx **F FF1]**	Kernel System Error	Soc
	Socapel System Error. Operation fault handler Read part FATAL SYSTEM ERROR in your user's manual.	

[02 xx **F FF2]**	Kernel System Error	App
	Arithmetic fault handler Arithmetic fault in application program -Integer Overlow. -Aritmetic Zero-Divide	

[02 xx **F FF3]**	Kernel System Error	App
	Floating point fault handler Floating point fault in application program -Floating Owerflow -Floating Invalid-Operation -Floating Zero-Divide	

B.5 REAL TIME SYSTEM ERRORS

[03 . . . . . .]	Real-time Kernel System Error

[03 xx **E 003]**	Real-time Kernel System Error	App
	Socapel System Error. Not enough heap (process allocation) Solution : -Reduce the number of task working simultaneously -or reduce the memory size allocated to workspaces.	

[03 xx **E 02C]**	Real-time Kernel Error	App
	Workspace Error This error appear only is the Stack control is enabled. (ref to QNA 104) Message : Snnn WORKSPACE Allocated : xxx Used: zzz with : Snnn is the Sequence number nnn (ID).You can find the sequence name in the file <applicname>.doc under PAM SEQUENCES. xxx is the stack size (decimal) you gave to this sequence. zzz is the stack size (decimal) maximum used by this sequence. Solution: -Reduce the number of overlapped process or -Try to enlarge the workspace memory size allocation	

[03 xx **F 02D]**	Real-time Kernel Error	Sys
	Socapel System Error. With this error the green LED on the face blink. This error comes always after another one, and to correct this error you have to correct the prior one and restart PAM. (see users manual §8.6)	

[03 xx **E 031]**	Real-time Kernel Error	App
	EXCEPTION Error.	
	The value of the time-out for this exception is < 0	

[03 xx **E 032]**	Real-time Kernel Error	App
	CONDITION Error.	
	The value of the time-out for this condition is < 0	

[03 xx **E 033]**	Real-time Kernel Error	App
	XEQ_TASK Error.	
	XEQ_TASK try to start a SEQUENCE in a TASK that is already running. Read part on XEQ_TASK and EXCEPTION ... XEQ_TASK in your reference manual.	

[03 xx **E 050]**	Real-time Kernel System Error	App
	Socapel System Error.	
	Not enough heap (init rt_mul_process)	
	Solution : -Reduce the number of task working simultaneously -or reduce the memory size allocated to workspaces.	

B.6 SMART IO ERRORS

[04 . . . . . .]	Smart_io Error

```
[04 xx W 00C]  Smart_io Warning                                  App / Flt
[04 xx K 00C]  End of Smart_io Warning

                         Smart_io default while working

               Refer to field D1 to know which smart_io is faulty
               and   to field D2 to know which fault.

               D1: smart_io (keyboard) address
               D2:  error code :
                    0x08   general over temperature     D3 : 0
                    0x01   heap memory overflow         D3 : 0
                    0x02   one command fifo is full     D3 : fifo
                                                             number
                    0x03   crc error on received frame  D3 : 0
                    0x04   bad frame type received      D3 : 0
                    0x09   unknown command received     D3 : 0
                    0x0A   not allowed command received D3 : 0
                    0x0B   request on undeclared item   D3 : 0
                    0x0C   request on unknown i/o type  D3 : 0
```

```
[04 xx W 00E]  Smart_io Warning                                  App / Flt
[04 xx K 00E]  End of Smart_io Warning

                      Smart_io output default while working

               D1:  keyboard address
               D2:  [ module number * 0x10000 +  group number ]
                      module 1,2,                group 1,2,3
                    if overtemperature :         group = 0
               D3:  error code :
                    5 = short circuit on output
                    6 = output heat think over temperature
```

[04 xx **W 010]**	Smart_io Warning	Flt
[04 xx **K 010]**	End of Smart_io Warning	

Warning on DC_motor

```
D1:  keyboard address
D2:  DC_motor ID (hexadecimal)
D3:  error code :
     0x07   pulse counting time-out when movement
            ordered
     0x0D   lower limit reached
     0x0E   upper limit reached
     0x0F   position lost
     0x10   unexpected movement
     0x11   stop do not act
     0x12   inquire position answer time-out
```

[04 xx **E 01A]**	Smart_io Error	App

Initialisation frame refused by a Smart_io

For each io item declared and for each keyboard (smart_io) an initialisation message (frame) is send. This error appear, if the message content a wrong value.

D1: keyboard address
D2: io variable ID (hexadecimal)
D3: 0

cause of error :
bad configuration value for an io item
(check the io declaration for the given variable D)

Example:
-declaration of more analog_output then vailable on the hardware,
-definition of a wrong Led or Key

B.7 RING ERRORS

[06 . . . . . .]	Ring Error

[06 xx **F 001]**	Ring Error	App / Cfg Flt
	Unsuccessful Hardware initialisation. This error is always after an other 06XXXXXX error that explain what is wrong. This one specifies just that something is wrong at initialisation time. D1: always 0 D2: always 0 D3: always 0	

[06 xx **F 002]**	Ring Error	App / Cfg
	Extra peripheral detected on optical ring. The following peripheral was detected on the ring, but not declared in your application. D1: ring mode (Socapel usage) D2: peripheral address (selected on board) D3: 0	

[06 xx **F 003]**	Ring Error	App / Cfg
	One peripheral is missing on optical ring. The following peripheral was declared in your application, but not found on the ring. D1: ring mode (Socapel usage) D2: peripheral address (selected on board) D3: 0	

[06 xx **F 004]**	Ring Error	App / Cfg
	Peripheral type mismatch. Type mismatch between the peripheral declared in your application, and peripheral found on the ring. (Peripheral type can be ST1 or SMART_IO) D1: ring mode (Socapel usage) D2: peripheral address (selected on board) D3: 0	

[06 xx **F 005]**	Ring Error	Flt
	Peripheral hardware problem. The following peripheral did not send the "ready" token during the initialisation part. D1: ring mode (Socapel usage) D2: peripheral address (selected on board) D3: 0 Cause of Error : peripheral hardware error, but optical connections are all right Example: ST1's microprocessor halted by a hardware problem.	

[06 xx **F 006]**	Ring Error	App / Cfg Flt
	Not correct number of peripherals. The count of detected peripherals on optical ring do not match with the number of peripherals declared in your application. (only in I indexed mode) D1: ring mode (Socapel usage) D2: peripherals count on the ring D3: 0 Cause of Error : Two or more peripherals have the same address.	

[06 xx **F 009]**	Ring Error	App / Flt
	Pam overloaded. Software reception fifo is full. D1: fifo address (Socapel usage) D2: peripheral address whose message can not be copied into fifo D3: 0 Cause of Error : PAM overloaded. Some peripherals send messages to Pam at very high rate Example : -Input oscillating at high rate -The I/O board is not present on a ST1 on which Inputs are declared.	

[06 xx **F 00C]**	Ring Error	Flt
	Optical link interrupted. Hardware communication fifo was empty when pam read it. D1: always 0 D2: always 0 D3: always 0 Cause of Error : at initialisation time :(ASIC or FPGA interfaces) (one of the following errors is 06000001) - the optical connection is not well made. - supply problem on one peripheral. at run time : (only with elder FPGA interface) - the optical connection is not well made. - supply problem on one peripheral.	

[06 xx **F 00E]**	Ring Error	Flt
	Detection of a frame with a CRC Error. D1: peripheral address received D2: always 0 D3: always 0 Cause of Error : At initialisation time : (one of the following errors is 06000001) - the optical connection is not well made. At run time : - hardware problem. - one peripheral was switched off.	

[06 xx **F 010]**	Ring Error	Flt
	Optical link interrupted. (ASIC interface) D1: peripheral address received D2: always 0 D3: always 0 Cause of Error : Optical link broken or power fail on one or several peripheral.	

[06 xx **F 016]**	Ring Error	App
	Too many peripherals on the optical ring. Too many peripherals on the ring to run at the cycle time (specified in your application). D1: nb maximum of peripheral possible D2: nb of peripheral detected D3: always 0 Solution to the problem : reduce peripheral number or increase cycle time.	

B.8 AUTOMATE ERRORS AND MESSAGES

[07 .. ....]	"Automate" Errors and Messages

[07 xx **E 001]**	"Automate" Error	Sys
	Memory allocation failure	
	Cause of error : Attempt to allocate more memory than available or allocation of 0 byte of memory (too many variables and equations definitions or System error)	

[07 xx **M 001]**	Message	Msg
	Attatch completed	

[07 xx **M 002]**	Message	Msg
	Waiting debugger command	

[07 xx **M 003]**	Message	Msg
	Ring empty configuration checked	

[07 xx **M 004]**	Message	Msg
	Ring configuration checked	

[07 xx **M 005]**	Message	Msg
	no dualport declared	

[07 xx **E 006]**	"Automate" Error	App

Software Fifo PAM to ring for io is FULL

Cause of error :
Too many changes on outputs generate by equations with linked output outputs, during many successive cycles.

[07 xx **M 006]**	Message	Msg

PLC_SI5 dualport initialised

[07 xx **M 007]**	Message	Msg

VME dualport initialised

[07 xx **E 008]**	"Automate" Error	App

Dual port overflow Buffer Full

Cause of error :
Too many changes on outputs generate by equations with linked output outputs, during many successive cycles.

[07 xx **M 008]**	Message	Msg

zero axis declared

[07 xx **M 009]**	Message	Msg

no interpolation group declared

[07 xx **M 00A]**	Message	Msg

Gesaxes is initialized

[07 xx **E 00B]**	"Automate" System Error	Sys
	Bad command acknowledge received from a Smart_io Cause of error : message corrupted or smart_io faulty	

[07 xx **M 00B]**	Message	Msg
	Ring peripherics ready	

[07 xx **M 00C]**	Message	Msg
	Ring ready without peripherics	

[07 xx **M 00D]**	Message	Msg
	All smart_io are configured	

[07 xx **M 00E]**	Message	Msg
	All smart inputs value received	

[07 xx **M 00F]**	Message	Msg
	Gesaxes is running	

[07 xx **M 010]**	Message	Msg
	0 axe : Gesaxes not started	

[07 xx **E 011]**	"Automate" Error	Cfg / Flt
	Bad logical address read from the WORD_INPUT Fifo of DUALPORT Simatic S5 cause of error : -Mismatch between PAM definitions of inputs and definitions made in the PLC or dualport interface -communication hardware problem.	

[07 xx **M 011]**	Message	Msg
	Dualport INPUTS are initialized	

[07 xx **M 012]**	Message	Msg
	VME Dualport variables configured	

[07 xx **E 013]**	"Automate" Error	Cfg / Flt
	Ring in Error This error is send after detection of one or more Ring errors, to confirm that the ring has stop working	

[07 xx **M 013]**	Message	Msg
	All smart_io are running	

[07 xx **F 014]**	"Automate" Error	Sys
	Smart_io configuration error This error is send after detection of one or more errors during the configuration of all the Smart_io	

[07 xx **M 014]**	Message	Msg
	All St1_io are configured	

[07 xx **F 015]**	"Automate" Error	Flt
	Smart_io initialisation refused One or more Smart_io have refused the initialisation order. cause of error : Some smart_io do not work properly.	

[07 xx **M 015]**	Message	Msg
	Equations with linked out installed	

[07 xx **F 016]**	"Automate" Error	Flt
	Smart_io answer missing One or more Smart_io did not acknowledge a configuration frame cause of error : Some smart_io do not work properly or do not receive correctly under broadcast mode !	

[07 xx **M 016]**	Message	Msg
	Starting power ON actions	

[07 xx **F 017]**	"Automate" Error	Flt
	Smart_io run acknowledge missing One or more Smart_io did not acknowledge the Run order. cause of error : Some smart_io do not work properly or do not receive correctly under broadcast mode !	

[07 xx **M 017]**	Message	Msg
	Working in main loop	

[07 xx M 018]	Message	Msg
	Power ON actions completed	

[07 xx M 019]	Message	Msg
	Evaluator process started	

[07 xx M 01A]	Message	Msg
	Watch process started	

[07 xx M 01B]	Message	Msg
	Waiting synchronisation with PLC	

[07 xx M 01C]	Message	Msg
	Waiting synchronisation with VME master	

[07 xx E 01D]	"Automate" System Error	Sys
	Variable ID out of range	
	The value of the ID of the variable is out of the range of the declared variable or Boolean equation.	
	cause of error : memory corrupted or bad value in a message from a smart_io.	

[07 xx M 01D]	Message	Msg
	Waiting PLC ready	

[07 xx **E 01E]**	"Automate" Error	App / Flt
	Dualport watch dog DUALPORT Master Inactivity Timeout elapsed. cause of error : -MASTER_INACTIVITY_TIMEOUT parameter badly defined -communication problem with dualport master.	

[07 xx **M 01E]**	Message	Msg
	Working without PLC	

[07 xx **F 01F]**	"Automate" Error	Sys
	Command Initialise or Run refused from the axis handler (gesaxes) when the "automate" send the command Cause of error : bad axis declaration.	

[07 xx **M 01F]**	Message	Msg
	Working without VME master	

[07 xx **F 020]**	"Automate" Error	App / Flt
	Host not ready Before synchronisation phase : DUALPORT Master Ready Timeout elapsed After synchronisation phase : DUALPORT Master Configuration Timeout elapsed Cause of Error : -MASTER_READY or MASTER_CONFIGURATION_TIMEOUT parameter badly defined -Handshaking problem between Pam and Master (Simatic or VME)	

[07 xx **M 020]**	Ges RS422	Msg
	RS422 Port Declared	

[07 xx **E 021]**	"Automate" Error	Flt
	Read io time-out Time-out when waiting result of the reading of an io on the ring. cause of error : -the accessed peripheral is not working -the read order is lost due to smart_io reception fifo full.	

[07 xx **M 021]**	Ges RS422	Msg
	RS422 Port On line	

[07 xx **M 022]**	Ges RS422	Msg
	RS422 Port Configured	

[07 xx **E 023]**	"Automate" Error	App
	Event fifo Full The fifo used to transmit the events (variables or equation transition) is detected full after n wait cycles. This error can only appear if the application generate at each basic cycle more events then the fifo size.	

[07 xx **M 023]**	Ges RS422	Msg
	RS422 Port Started	

[07 xx **M 024]**	Ges RS422	Msg
	RS422 Waiting connection	

[07 xx **F 025]**	"Automate" Error	App / Flt

Node initialisation Error

D1: ID of the declared NODE (ref file ***.doc) (hex)
D2: number of declared item of this node (hex)
D3: address of the first declared item (hex)

This error appear when none of the peripheral defined in the NODE declaration are detected on the ring.

Remark :
for each NODE declaration you get
error 07000025 with fields D1, D2, D3
followed by an last
error 07000025 with message NODE INIT. ERROR

Special case :
if there is no peripheral found on the ring:
for each NODE declaration you get
error 07000025 with message ALL NODES WITHOUT ACTIVE ITEM.

Check:
-ring configuration
-if the declared peripheral address are correctly coded on each peripheral.
-if some ring errors 06000xxx was displayed before.

[07 xx **F 028]**	"Automate" Error	App / Cfg

Bad CRC value for a VME_dualport output variable

During the initialisation of the VME_dualport output, the VME master refer to the output with the CRC value of the output declaration.
This error appear if the VME master give a CRC value that

PAM can not find in the output list.

D1: given CRC value
D2: given KEY value
D3: 0

Check if master and PAM are using the same outputs declaration files !

[07 xx **E 029]**	"Automate" Error	App / Cfg
	Bad KEY value for a VME_dualport input variable	
	The key value is used to refer to an particular input, this key value is given to the master by PAM during the declaration phase. This error appear if the VME master give a KEY value that PAM can not find in the input list.	
	D1: given KEY value D2: corresponding PAM KEY or FFFFFFFF if given key not found D3: 0	
	Check if master and PAM are using the same input declaration files and check if master handle correctly VME inputs !	

[07 xx **E 02A]**	"Automate" Error	App / Cfg
	Bad VME variable item index	
	The value of the index to access a particular item of an VME dualport variable is out of the range given by number or the node_groups in the variable declaration.	
	D1: value of PAM_key (hexadecimal) D2: Pam compiler ID value of the variable (hexadecimal) D3: index value (hexadecimal)	
	Cause of error : Vme master use a wrong value of item index for this variable.	

[07 xx E 02B] "Automate" Error App / Cfg

One VME event buffer is FULL

The change on VME dualport inputs are queued into buffers according to the period of the variable. This error occurs when the buffer corresponding to the period of this variable is full.

D1: period in number of cycles (hexadecimal)
D2: value of the count of buffer overflow (hexadecimal)
D3: size in bloc number of the buffer (hexadecimal)

Cause of error :
The Vme master write change at too high rate for too many variables of the same period.

[07 xx F 02D] "Automate" Error App

Bad ST1 IO item definition

D1: ID of the variable (hexadecimal)
D2: 0
D3: 0

Cause of error :
The declared io item for ST1 do not match with physical capability of the io board.
LPO board I101,I102
OIO board I201 to I216, O201 to O208

[07 xx E 035] "Automate" Error App / Flt

[07 xx S 035] "Automate" Error canceled

RS422 No user activity

PAM didn't receive any messages from the RS422 for a time longer then " Master Inactivity Timeout".

cause of error :
- -MASTER_INACTIVITY_TIMEOUT parameter badly defined
- -RS422 serial link problem (hardware, configuration)
- -Handshaking problem between Pam and RS422 Master application.

[07 xx **E 036]** "Automate" Error — App / Flt

[07 xx **S 036]** "Automate" Error canceled

RS422 Command refused

PAM has received a valid RS422 command but not executable within the phase in which PAM works. Examples : configuration command during execution phase or execution command during configuration phase.

Cause of error :
- -Handshaking problem between Pam and Master (RS422)
- -Bad retry of synchronisation after a PAM reset.

[07 xx **E 037]** "Automate" Error — App

RS422 bad command code

PAM has received an non valid RS422 command.

[07 xx **E 038]** "Automate" Error — App

RS422 time-out waiting to send

Appear only with application which use RS422 variables linked to equations, when time is over waiting to insert a message in the send queue.

Cause of error :
too many equations with linked output to RS422 variables with too many changes at the same time.

Solution :
Check linked equations to RS422 outputs and try to reduce number of changes at the same time.

[07 xx **E 039]**	"Automate" Error	App Cfg Flt
[07 xx **S 039]**	"Automate" Error canceled	

RS422 communication disconnected

Before synchronisation phase :
-RS422 Master Ready Timeout elapsed
After synchronisation phase :
-RS422 Master Configuration Timeout elapsed or
-RS422 serial link problem
In this case error 0700003A is previously displayed to give the detail of the communication problem

Cause of Error :
-MASTER_READY or MASTER_CONFIGURATION_TIMEOUT parameter badly defined
-RS422 serial link problem (hardware, configuration)
-Handshaking problem between Pam and RS422 Master application

[07 xx **E 03A]**	"Automate" Error	Cfg / Flt
[07 xx **W 03A]**	"Automate" Warning	Cfg / Flt

RS422 communication error

This message give the description of a RS422 communication error :

D1: communication status :
1 memory allocation error
2 port error
3 bad message crc
4 bad message END char.
5 bad message START char.
6 receive chars buffer full
D2: 0
D3: 0

[07 xx **E 03B]**	"Automate" Error	App / Cfg

Not valid RS422 Output CRC

This error may appear during configuration phase when the received CRC of the definition of an RS422 Output variable is not known from PAM.

D1: given CRC value
D2: given KEY value
D3: 0

[07 xx **E 03C]**	"Automate" Error	App / Cfg
	Bad RS422 Input KEY value	
	This error may appear during configuration phase, when master send initial value for inputs, if the received KEY of an RS422 Input variable is not known from PAM.	
	D1:	given KEY value
	D2:	PAM KEY value if unmatched, FFFFFFFF if not valid key
	D3:	0

[07 xx **E 03D]**	"Automate" Error	App / Cfg
	Bad RS422 Configuration	
	This error may appear after configuration phase, when some configuration errors did appear during the configuration.	

[07 xx **E 03E]**	"Automate" Error	App / Cfg
	Bad RS422 Input class	
	This error may appear during configuration received class of an RS422 do not match with PAM class of the variable. The class is given by the XNG_xxx command.	
	D1:	given class
	D2:	PAM class
	D3:	0

APPENDIX C

C DISPLAY OUTPUT TABLE (7 SEGMENTS)

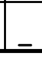
The HEXA font table for the display output is the following:

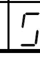
Dec	Hex	Chr	Disp
32	20		
33	21	!	1.
34	22	"	5
35	23	#	5
36	24	$	5
37	25	%	r.
38	26	&	e
39	27	'	5
40	28	(	5
41	29	)	5
42	2A	*	5
43	2B	+	5
44	2C	,	5
45	2D	-	-
46	2E	.	.
47	2F	/	r'
48	30	0	0
49	31	1	1
50	32	2	2
51	33	3	3
52	34	4	4
53	35	5	5
54	36	6	6
55	37	7	7
56	38	8	8
57	39	9	9
58	3A	:	_.

Dec	Hex	Chr	Disp
64	40	@	8.
65	41	A	A
66	42	B	b
67	43	C	C
68	44	D	d
69	45	E	E
70	46	F	F
71	47	G	G
72	48	H	H
73	49	I	I
74	4A	J	J
75	4B	K	K
76	4C	L	L
77	4D	M	n
78	4E	N	n
79	4F	O	0.
80	50	P	P
81	51	Q	q
82	52	R	r
83	53	S	5.
84	54	T	t
85	55	U	U
86	56	V	5
87	57	W	5
88	58	X	5
89	59	Y	5
90	5A	Z	5

Dec	Hex	Chr	Disp
96	60	`	5
97	61	a	A
98	62	b	b
99	63	c	C
100	64	d	d
101	65	e	E
102	66	f	F
103	67	g	G
104	68	h	H
105	69	i	-
106	6A	j	‾
107	6B	k	'
108	6C	l	ı
109	6D	m	_
110	6E	n	ı
111	6F	o	'
112	70	p	o
113	71	q	o
114	72	r	n
115	73	s	ɔ
116	74	t	ɔ
117	75	u	u
118	76	v	c
119	77	w	c
120	78	x	u
121	79	y	n
122	7A	z	=

Dec	Hex	Chr	Disp
59	3B	;	
60	3C	<	
61	3D	=	
62	3E	>	
63	3F	?	

Dec	Hex	Chr	Disp
91	5B	[	
92	5C	\	
93	5D	]	
94	5E	^	
95	5F	_	

Dec	Hex	Chr	Disp
123	7B	{	
124	7C			
125	7D	}	
126	7E	~	
127	7F		

APPENDIX D

D ST1 IO ADDRESSES AND CONNECTORS

D.1 LPO BOARD (NO ART. 024.7066)

PAM ADDRESS	ST1 IO
INPUT **101**	IN **0**
INPUT **102**	IN **1**

D.2 OIO BOARD (NO ART. 024.7047)

PAM ADDRESS	ST1 IO
INPUT **201**	IN **16 (with time stamps)**
INPUT **202**	IN **17**
INPUT **...**	IN ....
INPUT **216**	IN **31**
OUTPUT **201**	OUT **8**
OUTPUT **202**	OUT **9**
OUTPUT **...**	OUT ...
OUTPUT **208**	OUT **15**

D.3 ST1 IO PARAMETERS

These parameters are useful for fixing the states of the inputs and outputs after a reset or a power-on and before PAM has initialised the ST1 for the application.

D.3.1 OIO OUTPUTS PARAMETER

A ST1 parameter is provided for outputs active level initial value specification. Outputs states are physically initialised by the ST1 at reset or power-on. The initial state of an output is the inactive state.

This parameter is as follows:

CACOUT at address 180 (0xB4)

The parameter format is:

MSB LSB

not used	out 15	out 14	out 13	out 12	out 11	out 10	out 9	out 8

"1" = active high => 0V "0" = active low => 24V

D.3.2 OIO INPUTS PARAMETERS

These inputs parameter are useful only if some local actions in the ST1 are related to. If these inputs are used only by the PAM application, it is not necessary to initialise these parameters.

A ST1 parameter is provided for inputs active level initial value specification. These inputs states are logically initialised by the ST1 at reset or power-on.

This parameter is as follows:

CACTIN at address 178 (0xB2)

The parameter format is:

MSB LSB

in 31	in 30	in 29	in 28	in 27	in 26	in 25	in 24	in 23	in 22	in 21	in 20	in 19	in 18	in 17	in 16

"1" = active high "0" = active low

A ST1 parameter is provided for inputs validation masks initial value specification. These inputs states are logically initialised by the ST1 at reset or power-on.

This parameter is as follows:

CVALIN at address 179 (0xB3)

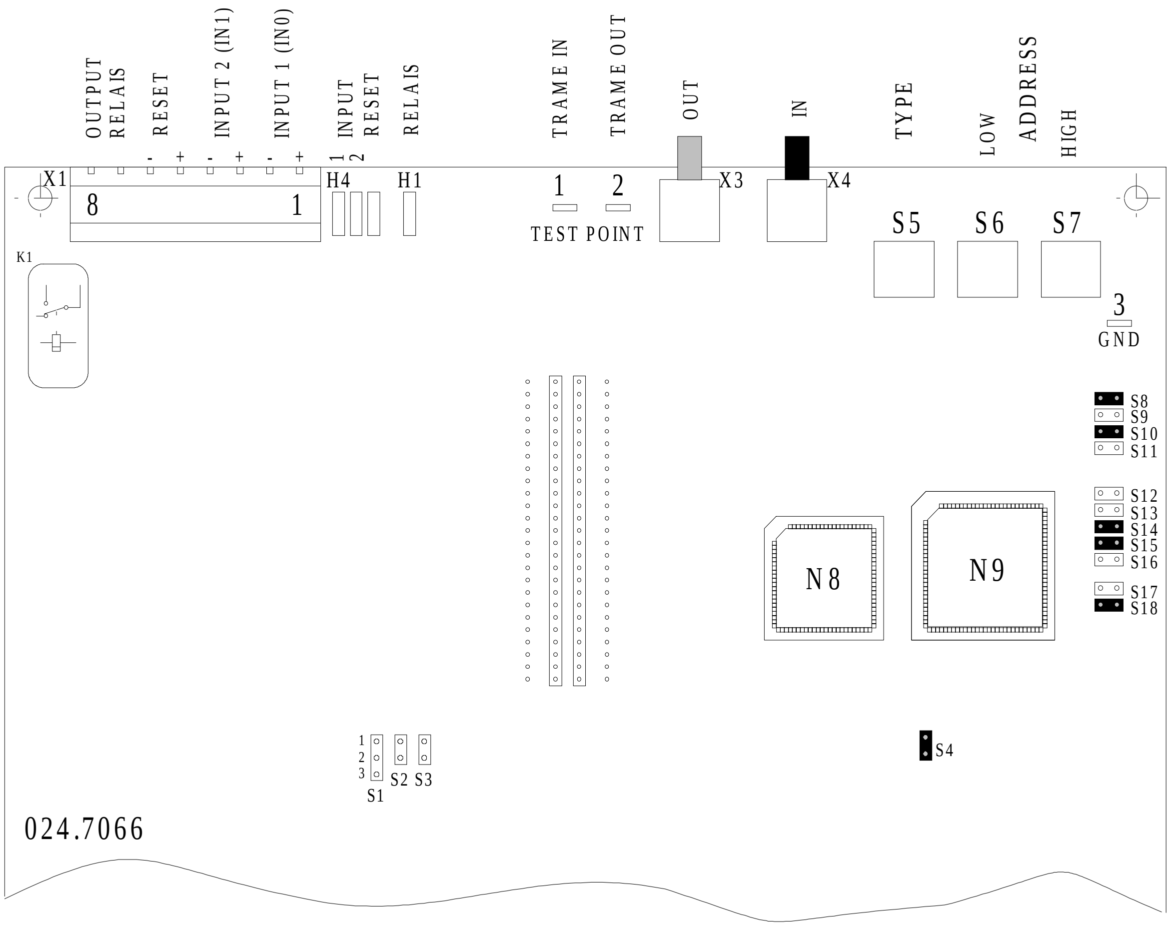
The parameter format is:

MSB LSB

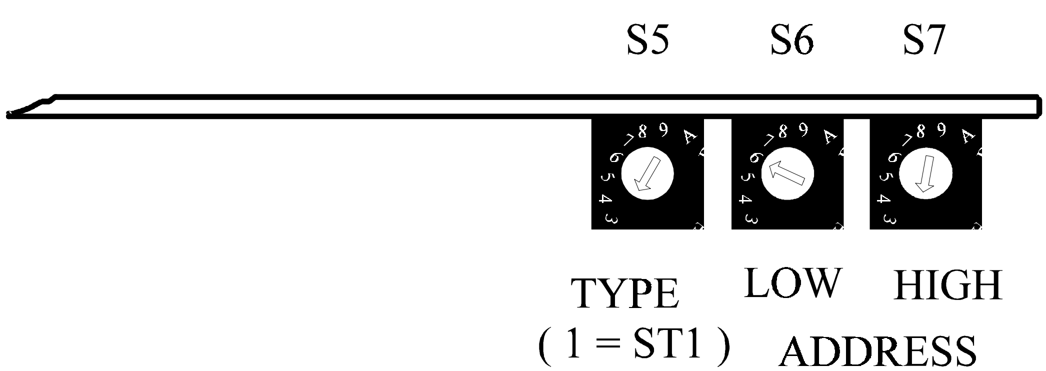
in 31	in 30	in 29	in 28	in 27	in 26	in 25	in 24	in 23	in 22	in 21	in 20	in 19	in 18	in 17	in 16

"1" = input activated "0" = input not used

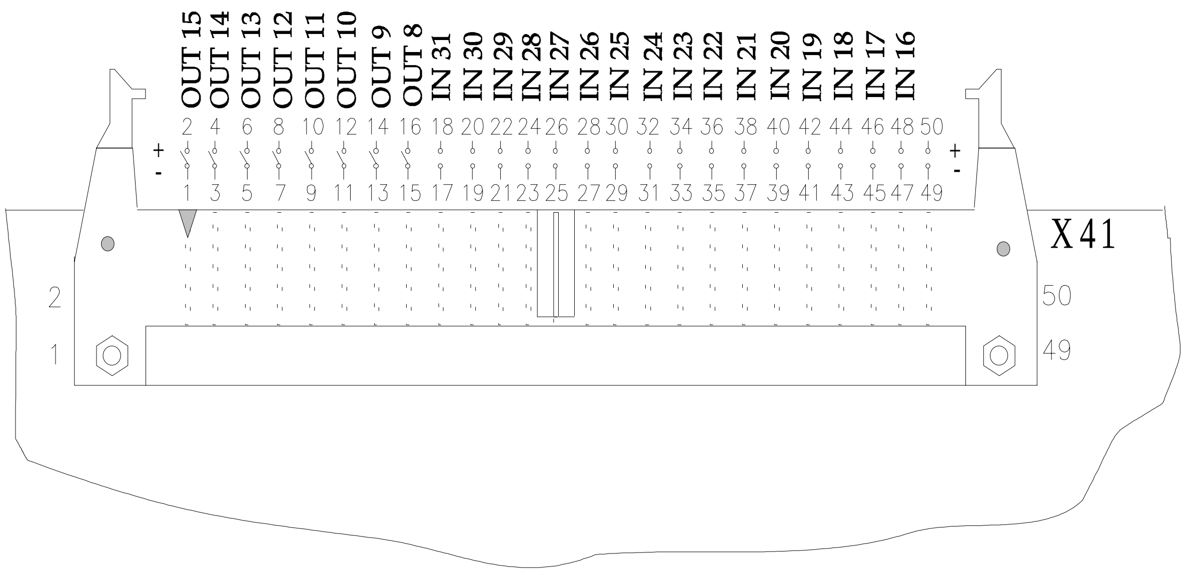
D.3.3 JUMPERS AND CONNECTORS CONFIGURATION ON THE LPO BOARD



D.4 RING ADRRESS CONFIGURATION ON THE LPO BOARD



D.5 CONNECTOR DESCRIPTION OF THE OIO BOARD (ART N° 024.7047)



APPENDIX E

E SMART-IO I/O ADDRESSES

Refer to PAM programmable axes manager SMART_IO technical manual No oo6.8003 for more information.

E.1 I/O CARD (NO ART. 006.7004)

Each Inputs and Outputs Card or Module have 12 inputs and 12 outputs

PAM address format **M P P**

<M> : Module number 1..16

<PP> : Input or Output position : 1..12

Module coding :

MODULE NUMBER	MODULE SWITCH CODING
1	`ON : X X X X` `OFF:`
2	`ON : X X X` `OFF:       X`
3	`ON : X X   X` `OFF:     X`
...	
16	`ON :` `OFF: X X X X`

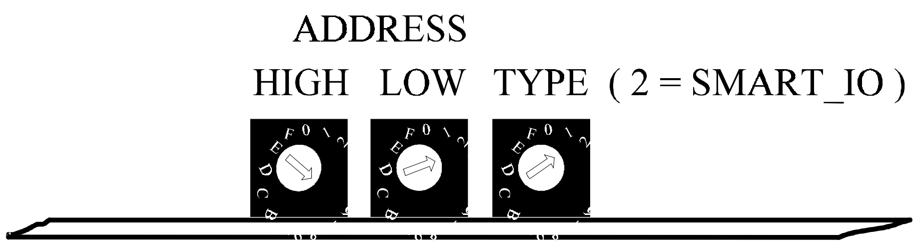
PAM I/O position and I/O number on board :

PAM ADDRESS	SMART_IO I/O CARD
INPUT **x01**	IN **0**
INPUT **x02**	IN **1**
INPUT **...**	IN **...**
INPUT **x12**	IN **11**
OUTPUT **x01**	OUT **0**
OUTPUT **x02**	OUT **1**
OUTPUT **...**	OUT **...**
OUTPUT **x12**	OUT **11**

E.2 CPU CARD (NO ART. 006.7018)

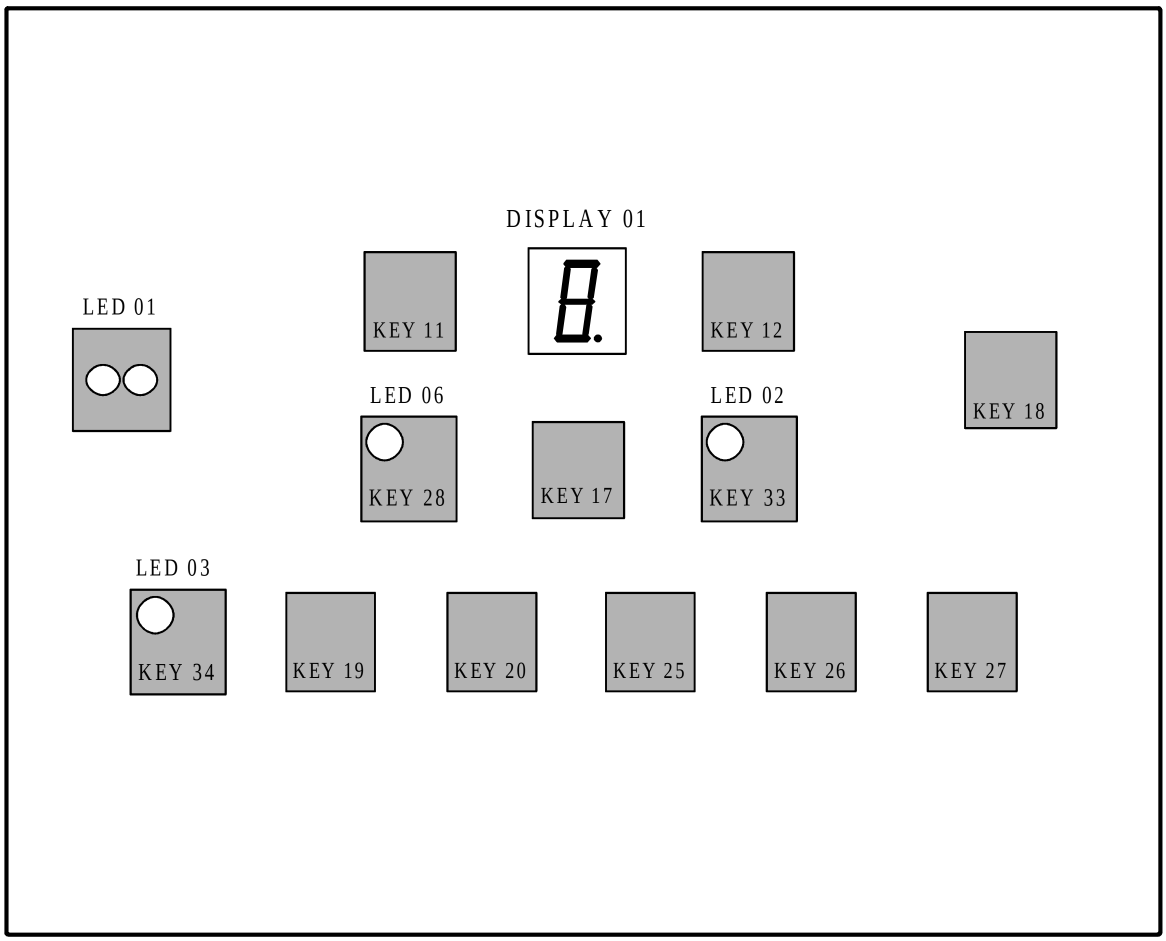
This card has 1 Analog Output whose PAM address is fixed at 1.

Ring address configuration on this board :



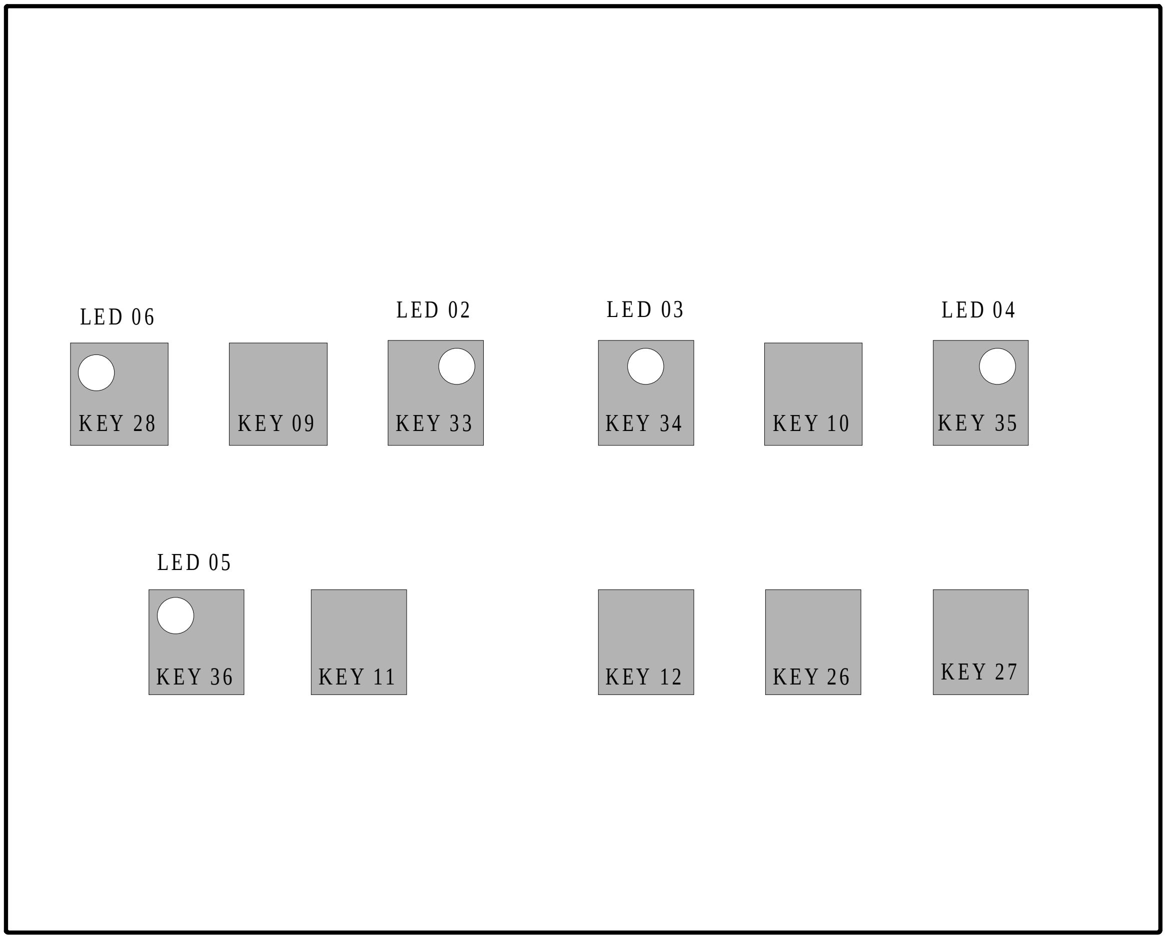
E.3 KEYBOARD CARD (NO ART. 006.7012)

This keyboard has key inputs, led outputs and one display output.



E.4 KEYBOARD CARD (NO ART. 006.7010)

This keyboard has key inputs and led outputs.



APPENDIX F

F SMART-IO DISPLAYED ERRORS

The error codes can only be displayed from SMART_IO with keyboard card **006.7012** (with one 7 segments display)

F.1 FATAL ERRORS :

1	on the 7 segment display.	**firmware CRC Error :** the contents of the eprom don't match with the computed crc value. ACTION : replace the eprom.
F 2	blinking on the display.	**Inactive ring :** detection of ring inactivity. CAUSES : - PAM system error. - optical link broken. - one or more peripheral not powered.
F 3	blinking on the display.	**Smart_io heap memory overflow** : Not enough memory space for the smart_io to work. CAUSE : the io configuration and number of io modules may be to important.
F 4	blinking on the display	**IO configuration error :** keybord not configured or loss of the configuration of all io. CAUSE : smart_io not yet configured or power switched off with battery low. ACTION : - check battery switch and voltage - reset PAM to reload configuration.

F.2 I/O CONFIGURATION IN PROGRESS :

8.	on the 7 segment display	**I/O configuration in progress :** PAM send the configuration of all i/o

F.2.1 OVERLOAD ERRORS :

These errors are non fatal errors, the smart_io display the code of error while it is trying to recover from this overload error.

E 1	appear on the display	**Fifo 1 full !** event fifo"slow" on interrupt 3.
E 2	appear on the display	**Fifo 2 full !** fifo for event to send to PAM.
E 3	appear on the display	**Fifo 3 full !** standard fifo.
E 4	appear on the display	**Fifo 4 full !** extended fifo.
E 5	appear on the display	**Fifo 5 full !** fifo for errors to send to PAM.
E 6	appear on the display	**Fifo 6 full !** reception fifo for PAM commands.
E 7	appear on the display	**Fifo 7 full !** hardware communication fifo with th ring.
E 8	appear on the display	**Fifo 8 full !** event fifo "fast" on interrupt 3.

The most frequent overload error is number 6. The cause is too large a number of command sent by PAM at each cycle. Check in the application if some sequence apply with a short period cycle, functions to outputs.

APPENDIX G

G WORKSPACE ERROR

When the size of the workspace is too small for a sequence, the system detects that the workspace limit is reached during sequence execution. This causes a PAM fatal error. PAM responds to this fatal error condition in the normal way which includes messages to the Fatal Error Panel, the PamDisplay and to the debugger.

The error code displayed on the PamDisplay is : **0380E02C** (with the green LED blinking)

The workspace error message displayed on the debugger screen includes the following:

date and time	0380E02C real time error
rt_kernel	Snnn WORKSPACE Allocated: xxx Used: zzz

where:

Snnn is the sequence number ID. Sequence names and ID's are listed in the file <application>.doc under PAM SEQUENCES.

xxx is the workspace size (decimal) specified for the sequence

zzz is the workspace size (decimal) maximum used by the sequence.

INDEX

—A—

—D—

—I—

—K—

—L—

—M—

—R—