**Socapel PAM**
A Programmable Axes Manager

# PAM System V2.1

## User's Manual

**Ordering Number: 006.8017.A**
Rev. October 1993

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. 006.8017.A/PB

# TABLE OF CONTENTS

# 1. INTRODUCTION

## 1.1. WHAT'S PAM

PAM (Programmable Axes Manager) is one of the most advanced axes-coordination systems on the market. That's the fruit of more than fifteen man-years of development. It offers the best available performance levels and compactness and employs the most advanced hardware and software technology available in an industrial product of its type.

PAM is designed for two main roles: integrated into a master unit, or operating as a standalone controller. Its hardware configuration is specially tailored to allow high-level, distributed control of a large number of axes and inputs/outputs. Its software architecture is configured for priority handling of axes and optical field bus (PAM-Ring), since any delays in these essential functions could desynchronise the field bus and cause cumulative errors in axis control. Management of inputs and outputs, the internal programmable logic controller and sequencing of applications are regarded as secondary activities, hence processed with a slight delay if the system risks to be overloaded.

PAM is therefore an axes-control system first, and input/output controller second. Its purpose is not to replace traditional programmable logic controller, but to enhance their functions.

## 1.2. PAM SYSTEM

The PAM system is composed of the following parts :

- *PAM module (for SIMATIC S5, VME, etc.)*
- *PAM-Ring*
- *Peripherals (ST1, Smart-IO, etc.)*

For more information refer to chapter 6 "System Considerations".

## 1.3. PAM TOOLS

The PAM Tools software running on DOS PC 286 or more computer, provides the following functions :

- *Generating a PAM application from the source file written in PAM Application language.*
- *Downloading and testing an application.*
- *Monitoring error messages.*
- *Cam file conversion utility.*

Refer to chapter 2 "Getting Started" for Tools installation and use.

## 1.3. PAM APPLICATION LANGUAGE

The PAM application language is a literal language with a natural syntax oriented for automation people.

This language allows to cut the application in parts as small as you need to fit with any sub function of the system to control and to associate an event to each part. These parts are executed as soon as the associated event occurs and are executed simultaneously.

Refer to Chapter 4 "PAM Application Language".

## 1.4. WHAT CAN DO PAM

The PAM associated with some ST1 digital motion controllers allows the following functions :

- *Independent axis control (multi-axis positioning).*
- *Multi-axis contouring with sequencing of movements.*
- *Electric drive shafts with severe time constraint and phase correction.*
- *Multi-axis synchronising with phase-correction.*
- *Axis movements synchronised by on-the-fly indexing functions.*
- *Replacement of mechanical cam functions.*
- *General interpolation functions.*

PAM can execute many of the mentioned functions simultaneously and these functions can be modified, activated or deactivated during execution of movements.

All the parts of a PAM application (tasks or actions) are activated as soon as the associated event occurs.

## 1.5. ABOUT THE USER'S MANUAL

The aim of this manual is to cover globally all that needs the user to create and execute a PAM application. For the chapters on application language, the approach is more global an close to the behaviour than in the Reference Manual which contents all syntactic description of all language items.

**Chapter contents :**

- *Chapter 2 : "Getting Started " :*
  *This chapter covers the PAM Tools installation, the PC configuration needed and how to create an application.*

- *Chapter 3 : "PAM Basic Concepts " :*
  *This chapter describes some basic concepts used in PAM and defines the names of the different active parts of the application.*

- *Chapter 4 : "PAM Application Language " :*
  *This chapter describes globally the PAM application language.*

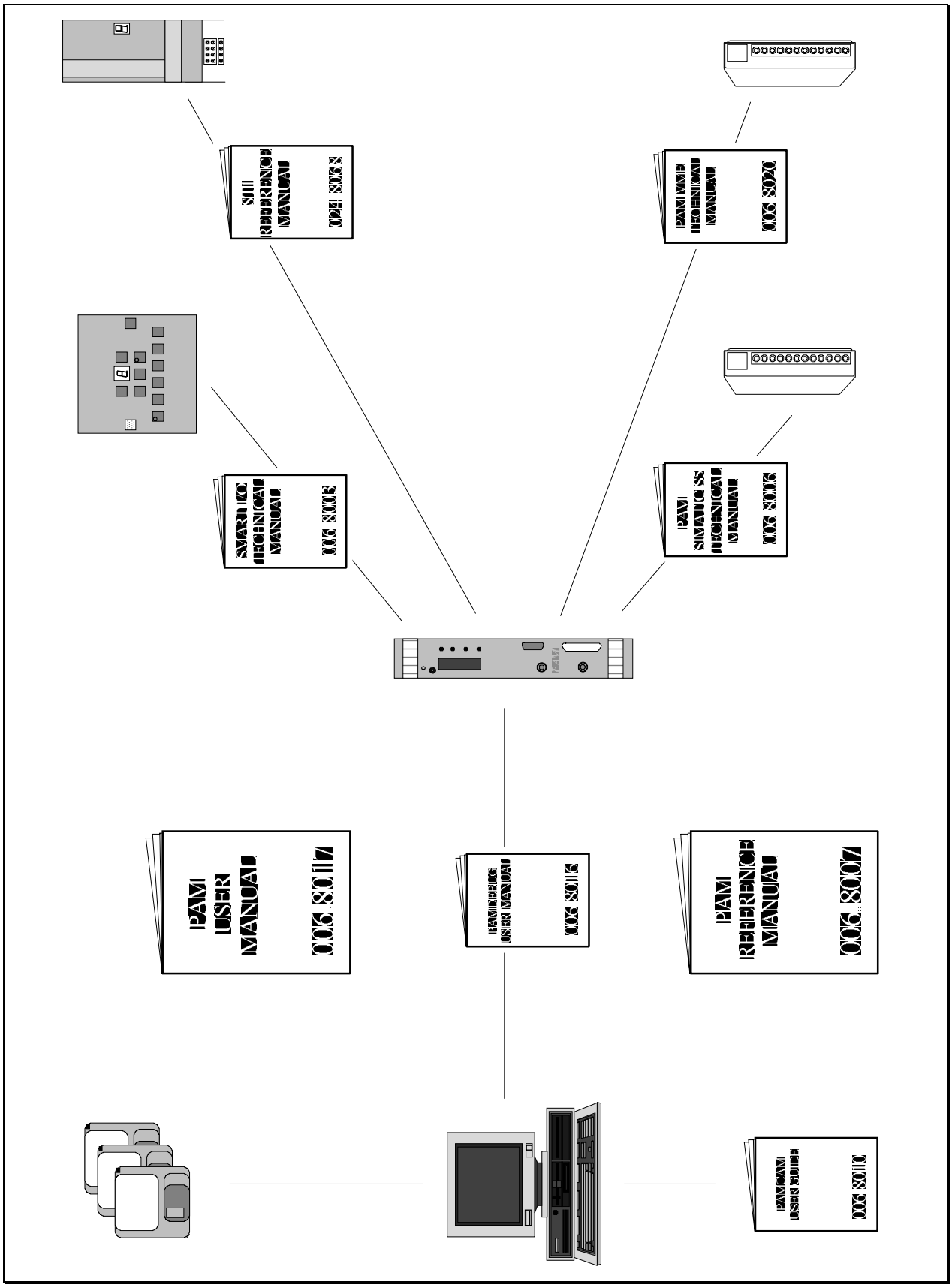
- *Chapter 5 : "PAM Application Behaviour " :*
  *This chapter gives more information's about internal mechanism and their effect on application behaviour, particularly the reaction time.*

- *Chapter 6: "System Consideration " :*
  *This chapter exposes different considerations about the physical parts of the SOCAPEL multi-axes products.*

- *Chapter 7: "PAM Display and Keys " :*
  *This chapter describes the use of the PAM display and keys for display and search of errors.*

- *Chapter 8 : "Advanced Concepts " :*
  *This chapter describes the concepts of multiple system and how multiple objects are handled.*

## 1.6. TECHNICAL PUBLICATION OVERVIEW



# 2. GETTING STARTED

## 2.1. SYSTEM REQUIREMENTS

The PAM Tools requires an IBM AT or compatible computer and MS-DOS version 3.3 or above. Your computer must be equipped with at least 640Kb of RAM, 1Mb of extended memory, and have a hard disk with 5 Mb free to store the various programs and files of the PAM Tools. At least one floppy disk and one serial port are also required.

Besides those stated above, a number of optional hardware components are supported. If you intend to use the PAM debugger with a mouse, it must be compatible with the Microsoft mouse. The PAM Tools supports almost any monitor, including EGA and VGA compatible boards.

## 2.2. PAM TOOLS INSTALLATION

Installing the PAM Tools on your hard disk is a simple procedure consisting normally of only two steps.

1. Run the INSTALL installation program.
2. Configure your system to accommodate the PAM Tools.


**INSTALL WILL OVERWRITE ANY FILES OF AN IDENTICAL FILENAME IN THE TARGET DIRECTORIES. IT IS RECOMMENDED THAT YOU REMOVE ALL FILES AND DIRECTORIES OF AN OLD PAM TOOLS VERSION TO AVOID POSSIBLE PROBLEMS WITH MIXED VERSIONS.**

### 2.2.1. RUNNING THE INSTALL PROGRAM

As mentioned above, the program INSTALL on disk 1 will install the PAM Tools onto your hard disk.

Before running INSTALL you should make backup copies of the distribution disks using the DISKCOPY program provided with your operating system. Use these working copies for the installation. You should then put the distribution disks in a safe place.

To run INSTALL place the working copy of the disk 1 in a floppy drive. At system prompt enter either:

```
INSTALL
```

If you are logged on to the floppy drive that contains the disk 1, or

```
A:INSTALL
```

where A: is the floppy drive that contains the disk 1.

The next screen shows the configuration of your system.

**THE PAM TOOLS VERSION NUMBER AND THE CONFIGURATION OF YOUR SYSTEM ARE NEEDED FOR TECHNICAL SUPPORT, YOU CAN PRINT THEM TYPING THE *Print Screen* KEY DURING THE INSTALLATION.**

The next screen informs you of what programs will be installed according to your CPU. Then the installation program will ask you the tools you wish to install.

INSTALL will ask for the drive on witch to carry out the installation.

After you have specified the drive, you will be prompted for the base directory under which you want the PAM Tools directory structure to be installed. The default is `\PAM` on that drive.

The program will prompt you to insert the copies you have made of your original disks.

Install will then ask you about automatic or manual modification of your `AUTOEXEC.BAT` and `CONFIG.SYS` files.

### 2.2.2. POST INSTALL PROCEDURES

If INSTALL did not modify your system files, you will need to make some changes to them with an editor.

These changes will not affect the operation of the other software you use, unless that software uses the same environment variables, in which case you must set up a batch file system to switch between the environment setting when required.

The installation of the PAM Tools is now complete.

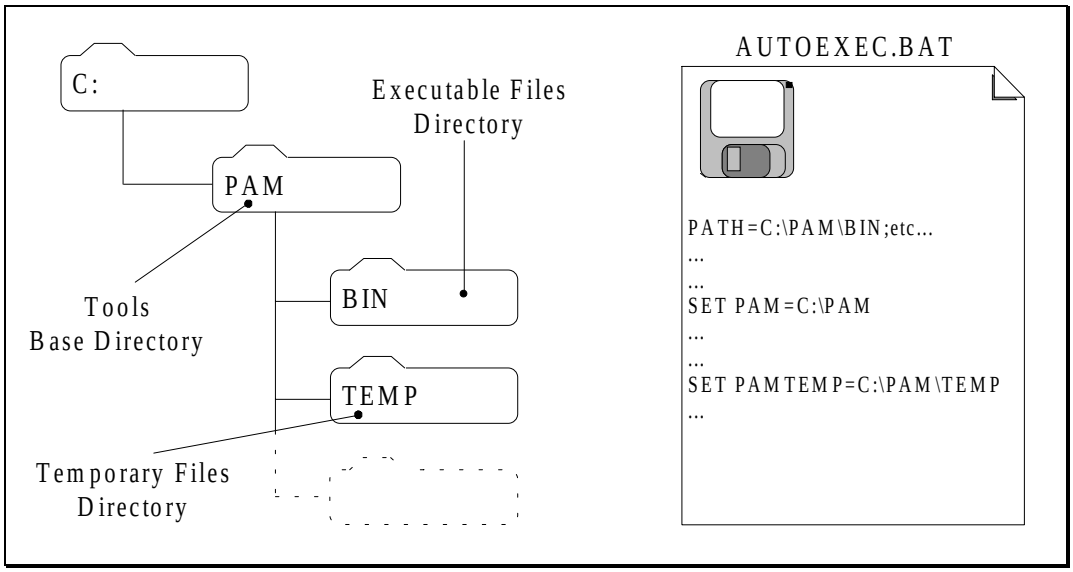
**REBOOT YOUR COMPUTER TO SET UP THE SYSTEM ENVIRONMENT.**

## 2.2.3. CONFIGURATION

If you wish to alter the default installation of the PAM Tools, you will need to understand the use of the directory paths and environment variables used by the PAM Tools.

### 2.2.3.1. PAM TOOLS ENVIRONMENT VARIABLES

After executing the INSTALL program, a directory structure like this below as been created:




PAM Tools directories structure

To use the PAM Tools programs from anywhere, the path of the `BIN` directory must be included in the directory paths ( the `PATH` environment variable in the `AUTOEXEC.BAT` file).

**Example**

```
SET PATH=<Disk>\<Base>\BIN
```

if the target disk is C: and the base directory is PAM, you will have

```
SET PATH=C:\PAM\BIN
```

#### PAM ENVIRONMENT VARIABLE

The PAM environment variable must indicate the PAM Tools base directory.
With a directory structure like above, you will have:

```
SET PAM=<Disk>\<Base>
```

if the target disk is `C:` and the base directory is `PAM`, you will have

```
SET PAM=C:\PAM
```

**PAMTEMP ENVIRONMENT VARIABLE**

The PAMTEMP environment variable must indicate the directory for temporary files.
If you use the default directory for temporary files you will have:

```
SET PAMTEMP=<Disk>\<Base>\TEMP
```

if the target disk is `C:` and the base directory is `PAM`, you will have:

```
SET PAMTEMP=C:\PAM\TEMP
```

If you use a RAM disk for temporary files you will have:

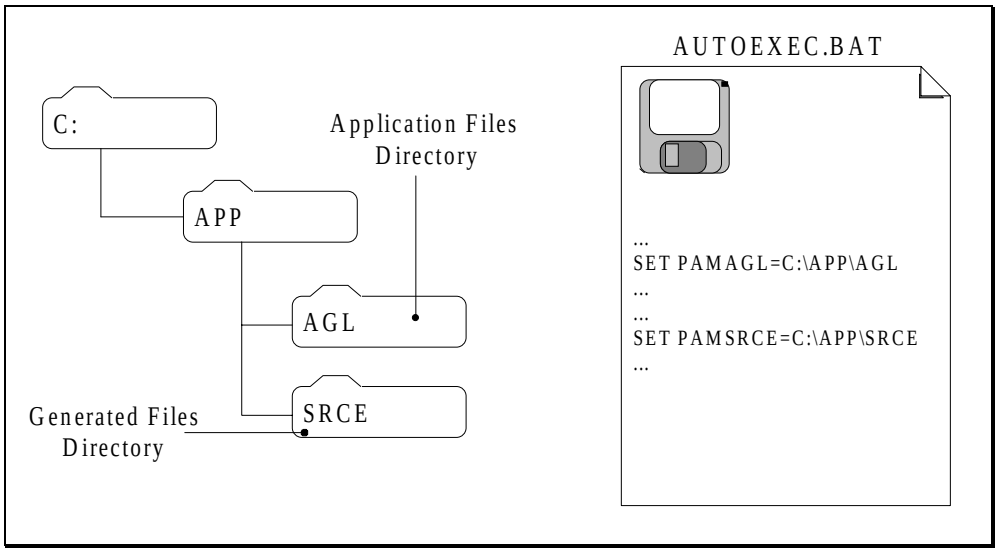
```
SET PAMTEMP=<RamDisk>
```

if the target disk is `E:`, you will have:

```
SET PAMTEMP=E:
```

### 2.2.3.2. APPLICATION ENVIRONMENT VARIABLES

The PAMAGL environment variable must indicate the directory where to find the application files. The PAMSRCE environment variable must indicate the directory for generated files.




Application directories structure

With a directory structure like above, if you are working on the APP application, you will have:

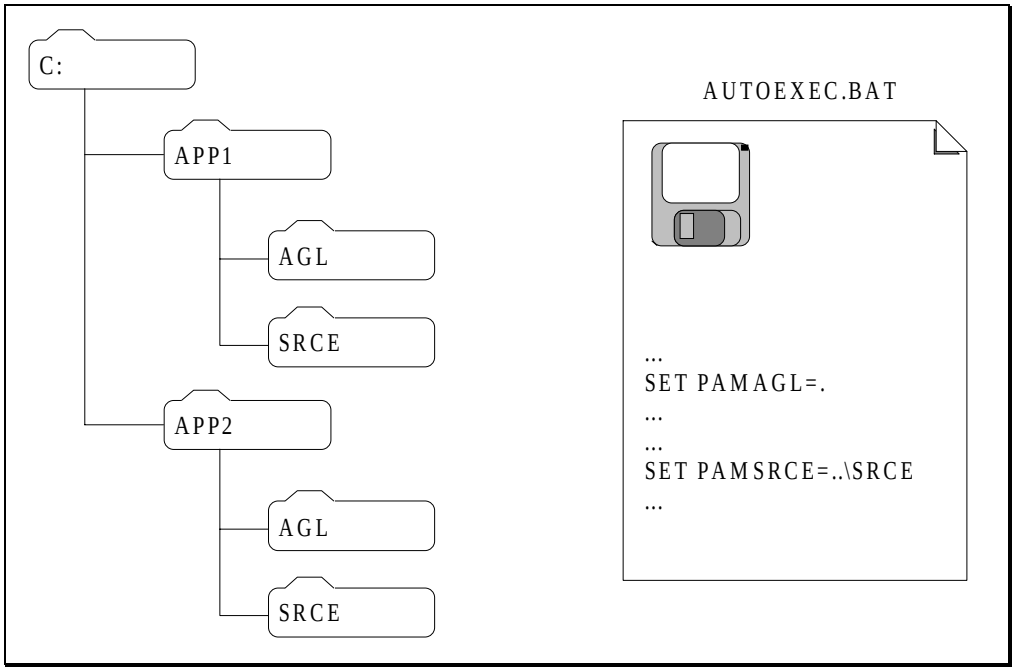
```
SET PAMAGL=<Disk>\APP\AGL
SET PAMSRCE=<Disk>\APP\SRCE
```

if the target disk is `C:`, you will have

```
SET PAMAGL=C:\APP\AGL
SET PAMSRCE=C:\APP\SRCE
```

If you are working on several applications, the default values can be used:

```
SET PAMAGL=.
SET PAMSRCE=..\SRCE
```




Multiple applications directories structure

If the default values are used, you must be in the `AGL` directory of the application before to start the compilation.

**PAMINC ENVIRONMENT VARIABLE**

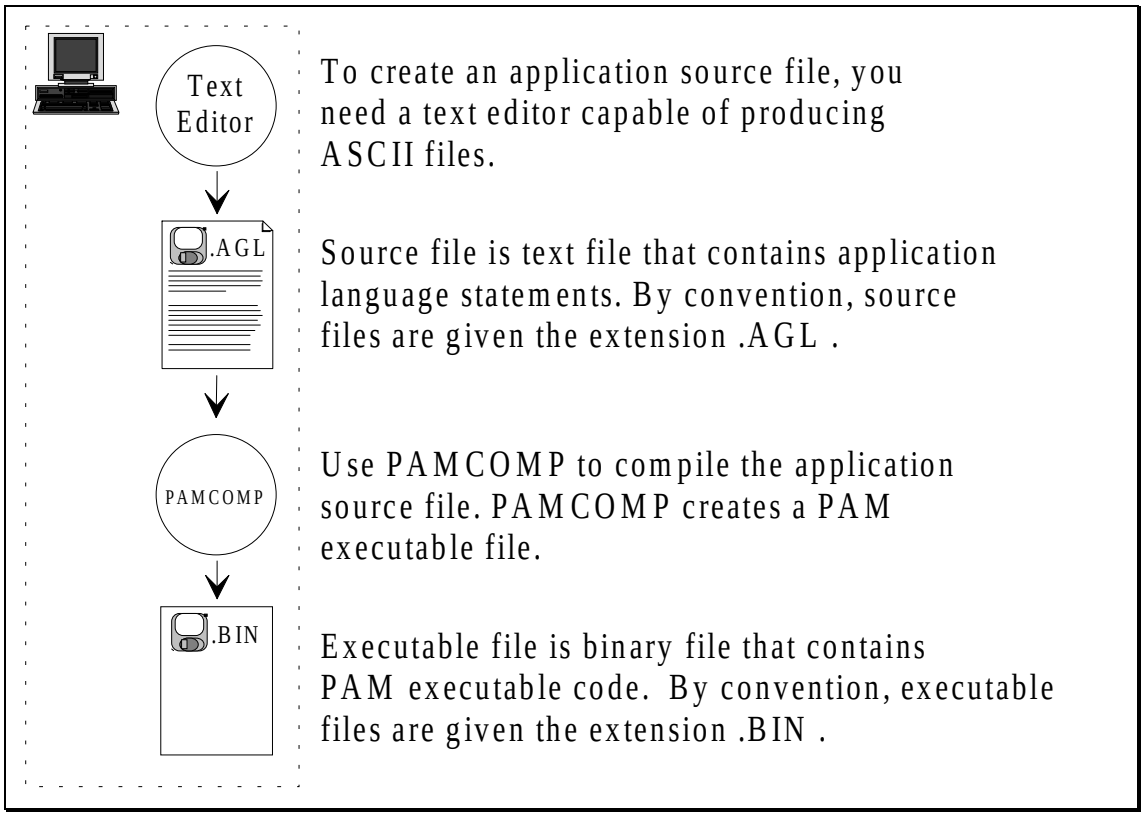
When a file is used (included) by several applications, it is a good practice to have only one copy of this file. The optional PAMINC environment variable indicates to the PAM Tools where to search included files. You can specify several directories separated with semicolon symbols.

**Example :**

```
SET PAMINC=C:\CAMFILE;D:\GLOBALS
```

## 2.3. CREATING AN APPLICATION

The two steps to create an application are the following:
- Edition of the application source file.
- Compilation of the application




Application creation steps

### 2.3.1. EDITING THE SOURCE FILE

To create application source files, you need a text editor capable of producing ASCII (American Standard Code for Information Interchange) files. Lines must be separated by a carriage-return/line-feed combination. If your text editor has a programming or non document mode for producing ASCII files, use that mode.

### 2.3.2. COMPILING AN APPLICATION

To compile an application, use the PAMCOMP compiler.

The PAMCOMP command line is the following:

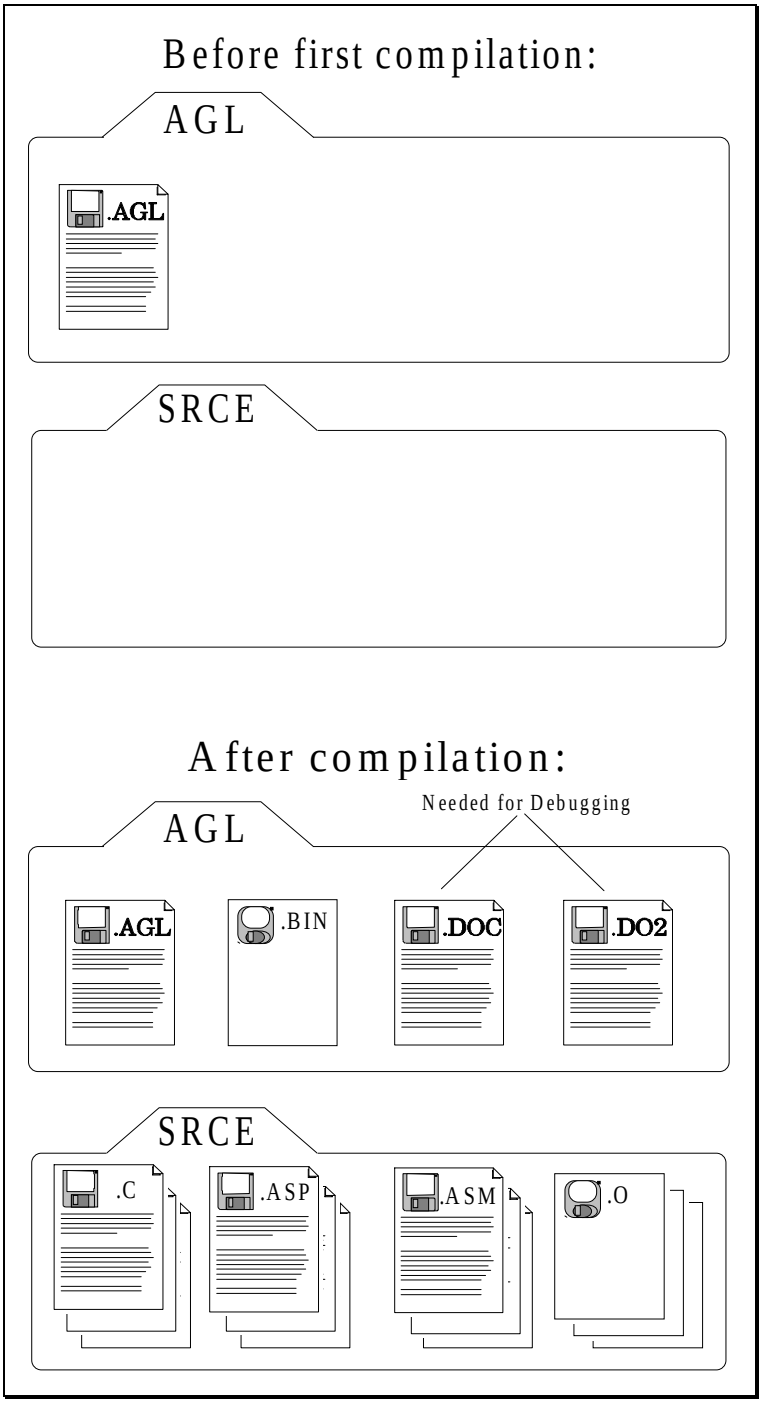
PAMCOMP [options] <srce_file> [.AGL]

**EXAMPLE :**

```
PAMCOMP example.agl
```

### 2.3.3. FILES LOCATION

The location of the files involved in the compilation process are as follows:




Application files location

# 2.4. COMMON PROBLEMS WHEN STARTING

## 2.4.1. COMPILATION PROBLEMS

If the compilation fail, generally, it is a memory problem. The steps to solve the problem are the following :

1. Simplify at maximum your `autoexec.bat` and `config.sys` files:
   - no memory manager.
   - all the memory "above" must be of extended type.
   - no resident utilities (TSR).
   - only the absolutely needed drivers must be loaded.

2. Reboot and compile again. If the compilation fail again go to the step 1.

3. Add one of the programs, drivers or memory managers you have suppressed at the step 1.

4. Delete all the files of your `SRCE` application directory.

5. Go to the step 2.

## 2.4.2. CONFIGURATION FOR WINDOWS AND MS-DOS 5.0

To run the compiler with a Windows$_{TM}$ and MS-DOS 5.0 compatible configuration, it is recommended to insert in the `config.sys` file the following lines:

```
DEVICE=C:\WINDOWS\HIMEM.SYS
DEVICE=C:\WINDOWS\EMM386.EXE 2048 RAM
```

For better performance, a RAM DISK can be created and the PAM environment variable of the `autoexec.bat` file modified as follows:

```
SET PAMTEMP=F:    if the ram disk is labeled "F"
```

# 3. PAM BASIC CONCEPTS

This chapter describes some basic concepts used in PAM and defines the names of the different active parts of the application.

## 3.1. TASK SCHEDULING

### 3.1.1. ABSTRACT

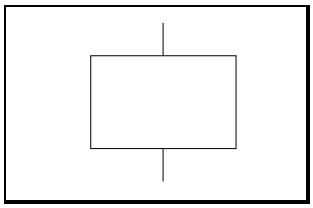
This unit will define the names of the different active parts of the application: statements, sequences and tasks. It will describe their properties and explain their relations in terms of time.

### 3.1.2. STATEMENT, DEFINITION

A statement is the smallest part of active code. We can distinguish two kinds of statements depending on their behaviour and effect on the system (machine).

For best readability, is it a good practice to write only one statement on each line of application program.

### 3.1.3. ACTIVE STATEMENT, DEFINITION



An active statement allows the description of **HOW** some parts of the system have to work. It is either an assignment or a control-flow statement.

An assignment statement consists in one left member which indicates the destination object and one right member which specifies the action that is to be performed on this objet. It is a combination of one or more basic instructions provided by the PAM language and some parameters.

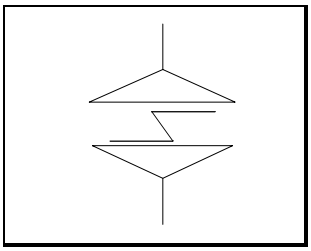
Both members are separated by an affectation operator. A semicolon indicates the end of the statement.

A control-flow statement allows to modify the flow of the sequence execution under some circumstances. It is build up with one of the IF, LOOP or XEQ_SEQUENCE keywords.

The following lines are examples of active statements:

```
Led1 <- set;
MainAxis <- run(1000.0);
LOOP
   ...
END_LOOP !Bool1;
```

### 3.1.4. SERVICE STATEMENT , DEFINITION



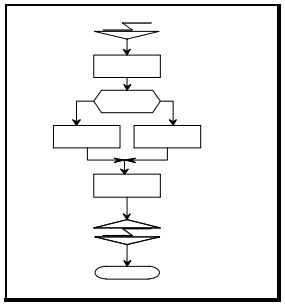
Service statement allow to specify **WHEN** some parts of the system have to work.

It is build up with the keywords CASE, CONDITION, WAIT_TIME or EXCEPTION and some object names or parameters to describe particular conditions. They are used to specify the activity conditions of a sequence.

The following lines are examples of service statements:

```
CONDITION MainAxis?ready;
EXCEPTION ErrorOccurs ENTRY stop;
```

### 3.1.5. SEQUENCE, DEFINITION



A sequence is a set of statements. These statements are processed sequentially. It means that each statement is completely executed before the execution of the next one begins.

### 3.1.6. SEQUENCE STATE

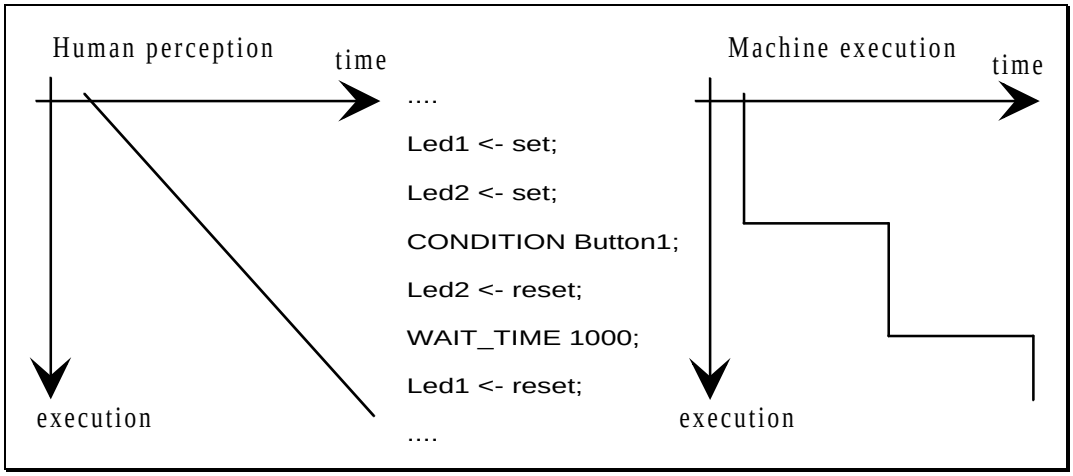
A sequence is called active while executing active statements. The execution of a service statement may imply to wait for something. In this case the sequence is suspended. When the end of the sequence is reached or if the sequence is aborted, the state becomes dead. Before having ever been active it is also called dead because all properties are the same with the preceding situation.

Both in active or suspended state, the sequence is also called alive because its history is in process even if it is suspended, waiting for a phenomenon.

### 3.1.7. SEQUENCE EXECUTION

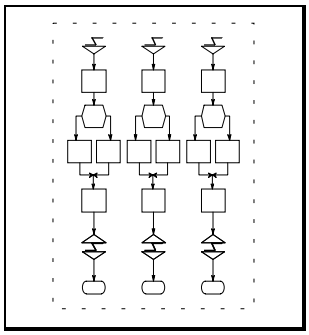
The human perception of a sequence execution, when reading application code, is a continuous and regular execution. On the contrary, the PAM system executes a sequence in a irregular way.

The execution time of any active statement is nearly zero. On the other hand, when a service statement is encountered it often implies the sequences has to wait for something. In this last case, the sequence is suspended until the corresponding event occurs.




Human perception of execution time versus machine execution time

### 3.1.8. TASK, DEFINITION



A task is a set of sequences that are related. A task is the part of code that describe the behaviour of a component of the system in an application program.

### 3.1.9. TASK PROPERTIES

Only one sequence of the task can be alive at the same time. It means that the sequences exclude them mutually. It means too that the task describes a part of the machine in with each elements are related and mutually excluded. It is important to keep in mind these rules when designing the application.

*Imagine for example a spaghetti manufacturing machine. One of many components is the pastry feeder. We can easily imagine a "feeder" task.*

*The "jog forwards feeder", "jog backwards feeder", "initialise feeder" and "feed in" could be sequences of this task. Indeed they are related because they all contribute to specify the behaviour of the pastry feeder. They also exclude them mutually because the machine physics imposes that it is not possible to jog forwards while jogging backwards and so on.*

### 3.1.10. TASK STATE

The state of a task depends on the state of its sequences. If there is one sequence of the task active, the task is active. If all sequences are dead, the task is dead. If one sequence is suspended, the task is suspended.

### 3.1.11. APPLICATION PROGRAM

The PAM application program is principally composed of all tasks necessary to describe the behaviour of all system components.

### 3.1.12. SCHEDULING

It is now easy to understand that only one sequence may be active at one time. The other are suspended, waiting for an external phenomenon. In fact, as soon as the phenomenon occurs the sequence is executed until the next waiting condition (or until the end). This operation takes quite no time and the system is immediately ready for an other phenomenon.

## 3.2. EVENT DRIVEN ARCHITECTURE

### 3.2.1. ABSTRACT

This unit describes the way in which different sequences of different tasks seem to be executed at the same time. Understanding this unit is not necessary for a basic knowledge of PAM but is useful for more advanced users.

### 3.2.2. EVENT, DEFINITION

An event is a change of some value **THAT HAS ANY IMPORTANCE FOR THE SYSTEM**. The importance attached to a change is due to the fact that any object is waiting for it.

Suppose the following code section:

```
...;
...;  // Code section A
EXCEPTION Input1 ENTRY EergencyStop;
...;
...;  // Code section B
CONDITION !Bool1;
```

During execution of code section "A", no event can occurs because nothing is awaited. During code section "B", only the state transition from false to true of the `Input1` Boolean object may cause an event. Indeed this transition has now any importance for the system because it may cause a modification of the program flow. While executing CONDITION statement, a transition from true to false of Boolean object `Bool1` will also cause an event to occur because the program is waiting for this condition.

### 3.2.3. EVENTS, EXAMPLES

```
CONDITION MainAxis?ready;
```

The event will occur when the specified axis will complete the motion in process.

```
CONDITION MainAxis?position > 12.0;
```

The event will occur as soon as the position of axis will exceed 12.0.

```
EXCEPTION MyVar<> 0 ENTRY stop;
```

If any non zero value is written in the variable, an event will be generated.

### 3.2.4. PARALLELISM

There are two basic ways to execute parallel jobs with a single processor machine (like PAM) bringing two different internal architectures. The first one is called time shared (sliced) architecture. The processor divides its time between several jobs. The switching between jobs is made without the knowledge of the job under execution. This configuration is particularly useful in systems where the activity of each job is constant and quite independent of time (business systems, general purpose computer).

The second approach is called event driven architecture. At a particular time, the job under execution itself "tells" what it is waiting for and what to do if it occurs. The corresponding task can be suspended and the processor is free for other jobs. This architecture is reserved for high performance real time systems. PAM is build following this model.

The main advantages of this way of managing the information are the following:

*The programmer knows exactly what is happening and when because he introduces service statements that allow task switching (explicitly or implicitly).*

*The other main difference is related to the statistics of a sequence activity. As presented before, a sequence (and thus a task) is quite always waiting. It saves then a lot of processor activity time by avoiding a sequence to be busy just to wait for a phenomenon.*

### 3.2.5. FAIRNESS

One of the basic hypothesis for fair application behaviour is that any active part of sequence has a short duration. As the execution time of any active statement is quite zero, the hypothesis is true. Nevertheless, if the number of actives statements sequentially executed is important, the overall time may be significant.

For this reason the END_LOOP statement introduces a small WAIT_TIME. This allows other sequences to perform their job without additional delay.

## 3.3. BASIC CYCLE

The basic cycle is the shortest time used in PAM's application. This basic cycle is given in third of millisecond (1/3 ms) and the smallest one is 3 third of millisecond (3/3 ms = 1 ms). This value is introduce in APPLICATION part (BASIC_PAM_CYCLE).

In each TASK and ACTIONS (in SPEC part), you introduce a cycle that is a factor of basic cycle (value can be 1, 5, 10, 20, 50). This cycle gives the maximum time to start the TASK or ACTION when an event occurs. This cycle gives also, in case of task switching, the time between the TASK's switch and next time that the task will run.

In one basic cycle, PAM does some SYSTEM jobs and some APPLICATION (user) jobs. The APPLICATION jobs are TASK and ACTION that are active and must be run.

All the TASKS and ACTIONS with a cycle = 1 are run every basic cycle, 1/5 of the TASKS and ACTIONS at cycle = 5 are run in one basic cycle, 1/10 of the TASKS and ACTIONS at cycle = 10 are run in one basic cycle, etc.

If all tasks are with a cycle = 1, PAM will be more loaded then if all of them are with a cycle = 50. The choice of the cycle for the TASKS and ACTIONS give the possibility to distribute the load of the application.

## 3.4. SUMMARY

Active statements describe **HOW** a job is performed, service statements describe **WHEN** it is performed.

An alive sequence is always waiting for something. The execution time of active parts is negligible.

No more than one sequence of the same task is active at one time.

An event is a change that has any importance for the system.

Events induce the activity of sequences. Service statements allow to manage the activity of a sequence.

# 4. APPLICATION LANGUAGE

## 4.1. GENERALITIES

The PAM application language is a **literal language** with a natural syntax oriented for automation people.

This language use the IEC normalised concepts (IEC 848 (1988), preparation of function charts for control systems).

The PAM application language is designed to be **compiled** by an IBM AT or compatible computer and to be executed by a PAM board. (The PAM board contents also all the firmware to run the compiled application).

### 4.1.1 APPLICATION LANGUAGE OBJECTS

The application language is build-up using objects like variables, Boolean equations, nodes, axes, pipe blocs, tasks, sequences and actions.

All these objects must be named. The name of an object is called **identifier**

**Variables :**

PAM variables are used to handle information and to access the surrounding world of PAM (Ring peripherals, dualport memory interface).

**Boolean Equation :**

A Boolean equation is the way to define a Boolean expression and to use the result of this expression.

**NODE :**

The declaration of nodes are used to define the PAM ring configuration (type of components, number and addresses).
A node declaration may associate a name to a single or a multiple component. (Refer to chapter 8 "advanced concepts" for more information's on multiple components).

**AXIS :**

Nodes with ST1 digital motion controllers are composed of an axis part and an IO part.
The axis declaration defines some axis properties ( units, motion ...).

**PIPE :**

A pipe is a set of chained pipe blocks. Pipes are used in case of axes interacting in a synchronous way.

**TASK :**

A task describes totally or partly the behaviour of a component of the system. This behaviour can be split up in several **sequences** of operations. Only one sequence of a task can be alive (under execution) at the same time.

**SEQUENCE :**

A sequence is started when an event happens, a condition is true, an exception occurs or at power on. In a sequence, besides actions on physical objects (IO, axes, nodes, variables, etc.) it is possible to start another sequence of the same task, execute an other task, abort the task, install and remove an exception, wait a condition or a "time out", activate and disactivate pipes, etc. Some particularities of the system behaviour do not need to be described in a sequence because the possibilities above-mentioned are useless. In this case **actions** are used.

**ACTION :**

An action is a succession of statements (except statements as sequence execution, task execution, exception installation, etc). The action is executed when an event happens or at power on.

### 4.1.2. ASSIGNMENT

The syntax of the assignment statement is the following:

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

**Types :**

The valid type combinations are the following:

<boolean object> <- <boolean expression> ;

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

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

**Functions :**

There are two kinds of functions: 'normal' an 'inquire'.

- Normal function: ( apply the function to the object)
  <object> **<-** <function name>[(parameters)]**;**

- Inquire function: (ask the object to return a value )
  <object> **?** <function name>[(parameters)]**;**

### 4.1.3. FLOW-CONTROL STATEMENTS

Flow-control statement are active statements. They represent a small part of the PAM application language because service statements are normally used for the switching.

Classical flow-control statements are :
- IF THEN ELSE ENDIF
- LOOP END_LOOP

The other ones are closer to a service statement :
- XEQ_TASK
- XEQ_SEQUENCE

### 4.1.4. APPLICATION COMMENTS

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

1. You may use the symbols **/*** and ***/** to start and end a comment, these cannot be nested.

```
/* This is a comment */
/* This is a comment  /* but this is illegal */ */
```

2. You may use an alternate form of commenting denoted by the characters **//.**
   These indicate the start of a comment, that terminates at the end of the line.

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

## 4.2. VARIABLES AND EQUATIONS

Variables are used to manipulate informations.

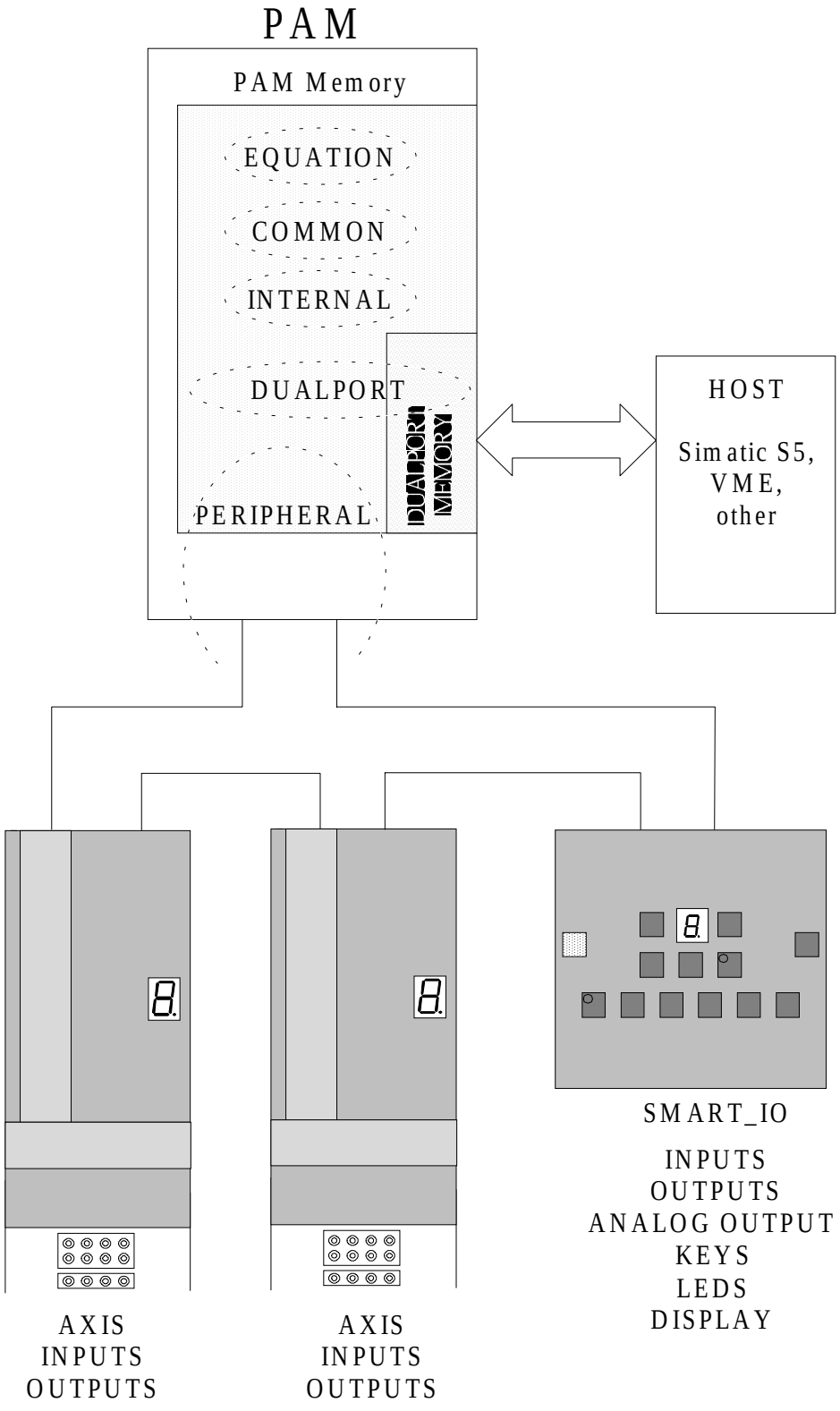
PAM variables are divided in four main classes
- COMMON variables
- INTERNAL variables
- DUALPORT variables
- PERIPHERAL variables

EQUATIONS may be considered as a fifth class of variables, but they look more like a function call returning a value.

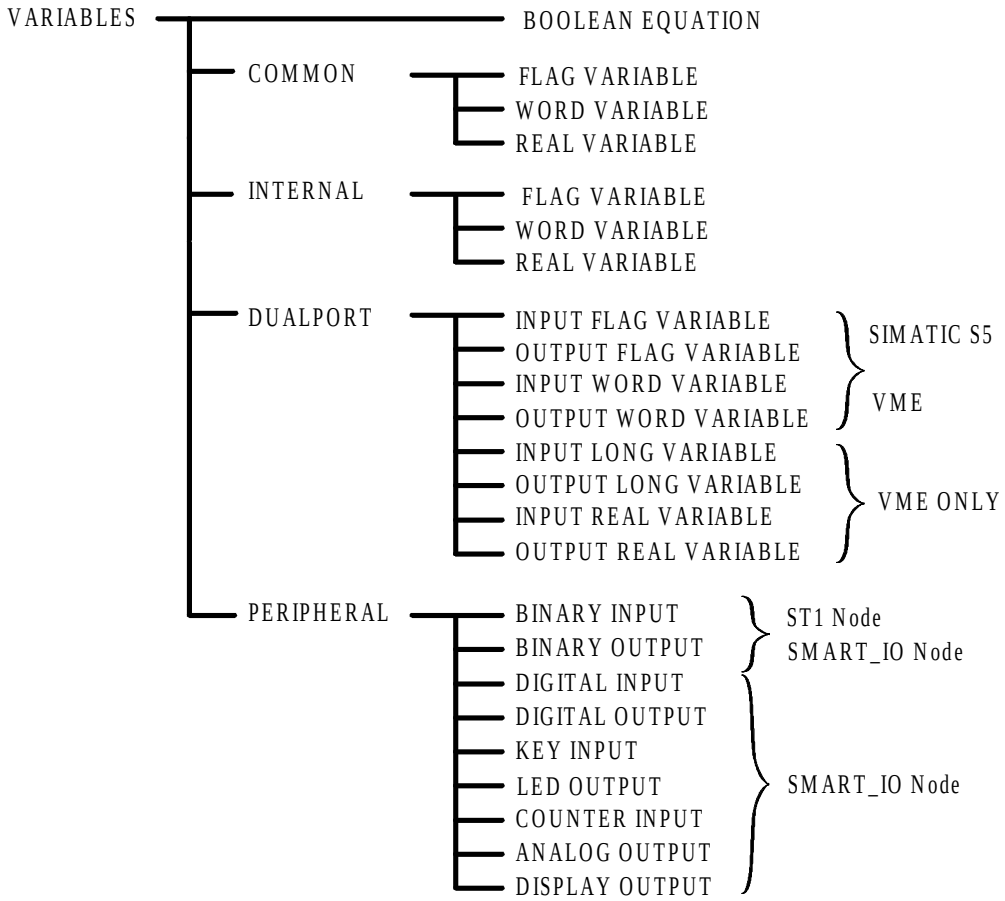
### 4.2.1. COMMON PROPERTIES OF VARIABLES

- Any variable or equation can be read by using the name given in its declaration part.
- Any variable has an internal representation in the PAM memory.

### 4.2.2. LOCATION OF VARIABLES



### 4.2.3. OVERVIEW OF VARIABLES



### 4.2.4. COMMON VARIABLES

Common variables are located in the PAM memory. These variables are similar to variables of programming language like C.
Common variables do not carry more information then their own value.

Common variables are provided to be used as loop counter and other local computation.

**COMMON Variables Types :**

| TYPE | VALUE | ACCESS |
|---|---|---|
| **FLAG_VAR** | **Boolean** | **read / write** |
| **WORD_VAR** | **integer 32 bits** | **read / write** |
| **REAL_VAR** | **float 64 bits** | **read / write** |

**Example :**

```
/* Declaration */
COMMON FLAG_VAR MyCommonFlag ;
COMMON WORD_VAR MyCommonWord ;
COMMON REAL_VAR MyCommonReal ;
/* Use in a sequence */
...
IF MyCommonFlag THEN
    MyCommanReal <- MyCommanReal * 1.05 ;
ELSE
    MyCommonReal <- MyCommonReal * MyCommonWord / 100 ;
END_IF
....
/* Use in the poweron part of a task */
POWERON;
    MyCommonFlag <- reset;
    MyCommonReal <- 1.03456912;
    MyCommonWord <- 100 ;
END_POWERON
```

### 4.2.5. INTERNAL VARIABLES

Internal variables are located in the PAM memory. In addition to their own value, internal variable carry the list of Boolean equations using this variable.

When a write access is made to an internal variable, the change is propagated to all equations in which this variable is used.

A write access to an internal variable is an Event !

**USE :**

An internal variable is to be used if this variable is not only accessed from sequences or actions, but is also used in equations or Boolean expressions.

**Types of INTERNAL Variables :**

| TYPE | VALUE | ACCESS |
|---|---|---|
| **FLAG_VAR** | **Boolean** | **read / write** |
| **WORD_VAR** | **integer 32 bits** | **read / write** |
| **REAL_VAR** | **float 64 bits** | **read / write** |

**Example :**

```
/* Declaration */
INTERNAL FLAG_VAR MyInternalFlag1 ;
INTERNAL FLAG_VAR MyInternalFlag2 ;
INTERNAL WORD_VAR MyInternalWord  ;
INTERNAL REAL_VAR MyInternalReal  ;


/* Use in a Boolean equation */
BOOLEAN MyFirstEquation ;
    EQUATION..MyInternalFlag2 * MyInternalWord > 100;
END_BOOLEAN
/* Use in a task */
TASK TaskExample1;

    SPECS
        CYCLES = 10;
    END_SPECS

    EVENTS
        ON_EVENT MyInternalFlag1 XEQ_SEQUENCE MySequence ;
    END_EVENTS
```

```
/* This Sequence starts after poweron because MyInternalFlag1 is set
during the poweron part of the task
*/

    SEQUENCE MySequence ;
        MyInternalFlag2 <- set;
        MyInternalReal  <- 1.07;
        WAIT_TIME 2000;
        MyInternalFlag2 <- reset;
        MyInternalReal  <- 0.87;
    END_SEQUENCE

    POWERON;
        /* To start the sequence after poweron */
        MyInternalFlag1 <- set ;
        MyInternalReal  <- 1.0;
    END_POWERON

END_TASK
```

## 4.2.6. DUALPORT VARIABLES

DualPort variables are located in the PAM memory and for the Simatic S5 version in the dualport memory. For VME version the dualport is used as communication channel. (For Simatic version, values of variables are written into the dualport memory and the change is queued into a part of the dualport used as communications FIFOs).

The dualport variables in PAM memory carry not only the value of the variable but also the list of Boolean equations using this variable and information to handle this variable in the dualport memory.

DualPort variable may be an INPUT variable (Host write it, PAM read it) or an OUTPUT variable (PAM write it, Host read it).

When a change to a DualPort INPUT variable is detected by PAM, the new value is then copied into the PAM part of the dualport variable and its changing is propagated to all equations in which this variable is used.

When PAM writes a new value into a DualPort OUTPUT variable, the PAM part of the variable is updated, the new value is written or announced into the dualport and its changing is also propagated to all equations in which this variable is used.

Changes to an INPUT variable detected by PAM and writing by PAM to an OUTPUT variable are Events !

**DUALPORT Variables Types :**

| **Version: SIMATIC S5** | | |
|---|---|---|
| **TYPE** | **VALUE** | **ACCESS** |
| **DUALPORT_IN FLAG_VAR** | **Boolean** | **read** |
| **DUALPORT_OUT FLAG_VAR** | **Boolean** | **write** |
| **DUALPORT_IN WORD_VAR** | **integer 16 bits** | **read** |
| **DUALPORT_OUT WORD_VAR** | **integer 16 bits** | **write** |

| Version: VME | | |
|---|---|---|
| **TYPE** | **VALUE** | **ACCESS** |
| **INPUT FLAG_VAR** | **Boolean** | **read** |
| **OUTPUT FLAG_VAR** | **Boolean** | **write** |
| **INPUT WORD_VAR** | **integer 16 bits** | **read** |
| **OUTPUT WORD_VAR** | **integer 16 bits** | **write** |
| **INPUT LONG_VAR** | **integer 32 bits** | **read** |
| **OUTPUT LONG_VAR** | **integer 32 bits** | **write** |
| **INPUT REAL_VAR** | **float 64 bits** | **read** |
| **OUTPUT REAL_VAR** | **float 64 bits** | **write** |

**Example : (Simatic Version)**

Before declaring any dualport variable, the dualport header must be declared (see reference manual )

```
/* Declaration */
DUALPORT_IN FLAG_VAR MyDualportInputFlag ;
    ADDRESS = #104 BIT 3;
END

DUALPORT_OUT FLAG_VAR MyDualportOutputFlag ;
    ADDRESS = #905 BIT 0;// cell address 904 bit 8
END

DUALPORT_IN WORD_VAR MyDualportInputWord ;
    ADDRESS = #120 ;
END

DUALPORT_OUT WORD_VAR MyDualportOutputWord ;
    ADDRESS = #920 ;
END

/* Use in a Boolean equation */
BOOLEAN MySecondEquation ;
    LINKED_OUTPUT = MyDualportOutputFlag ;
    EQUATION (MyDualportInputWord > 90) * (MyDualportInputWord <= 99);
END_BOOLEAN

/* Use in an action */
ACTIONS ActionsExample1;

    SPECS
        CYCLES = 20;
    END_SPECS

    ON_EVENT MyDualportInputFlag  ACTION
        MyDualportOutputWord <- 3000;
    END_ACTION

    ON_EVENT !MyDualportInputFlag  ACTION
        MyDualportOutputWord <- 2000;
    END_ACTION

END_ACTIONS
```

**Example : (VME Version)**

All declarations of dualport variables must be enclosed between the dualport header declaration and the END_DUALPORT statement (see reference manual ).

Remark : with the VME version the addresses are not given in the declaration statements but at run time during a configuration phase between PAM and VME master ( PAM sends to the VME master the code or address used by PAM for Inputs and the VME master sends the code or address that PAM has to use for Outputs).

```
/* Declaration */

VME DUALPORT
    SPECS
        DEFAULT PERIOD = 20 ;
        MASTER TIMEOUT = 500 ;
    END_SPECS


/* Definitions of Variables */

    INPUT FLAG_VAR MyVmeDualportInputFlag ;
    OUTPUT FLAG_VAR MyVmeDualportOutputFlag ;
    INPUT LONG_VAR MyVmeDualportInputLong ;
    OUTPUT REAL_VAR MyVmeDualportOutputReal ;

END_DUALPORT

/* Use in a task */

TASK TaskExample1;

SPECS
    CYCLES = 10;
END_SPECS

EVENTS
    ON_EVENT MyVmeDualportInputFlag XEQ_SEQUENCE MySequence ;
END_EVENTS


SEQUENCE MySequence ;
    MyVmeDualportOutputFlag<- set;
    MyVmeDualportOutputReal <- 1.07;
    CONDITION  MyVmeDualportInputLong > 2000;
    MyVmeDualportOutputFlag<- reset;
    MyVmeDualportOutputReal <- 0.93;
END_SEQUENCE

END_TASK
```

### 4.2.7. PERIPHERAL VARIABLES

All peripheral variables have their logical part located in the PAM memory, and their physical part located on the peripheral with local memorisation and local CPU for handling and event detection.

A peripheral variable in PAM memory carries not only its value but also the list of Boolean equations using this variable and information to access this peripheral item through PAM ring.

Peripheral variables may be INPUT variables (PAM is waiting for changes sent through PAM-Ring by the peripheral device) or OUTPUT variable (PAM is sending the new value through PAM ring).

For INPUTS, the peripheral device send the value only if a change has been detected. As soon as PAM has received a change, the value is updated into the PAM part of the peripheral variable. Its changing is propagated to all equations in which this variable is used.

When PAM writes the new value into a peripheral OUTPUT variable, the PAM part of the variable is updated. Its changing is also propagated to all equations in which this variable is used, and the new value is send through PAM ring.

INPUT variable change announced by a peripheral device and OUTPUT variable change made by PAM are Events !

**PERIPHERAL Variables Types :**

| TYPE | VALUE | ACCESS | NODE |
|---|---|---|---|
| **BINARY_INPUT** | **Boolean** | **read** | **all** |
| **BINARY_OUTPUT** | **Boolean** | **write** | **all** |
| **DIGITAL_INPUT** | **integer n bits** | **read** | **smart_io** |
| **DIGITAL_OUTPUT** | **integer n bits** | **write** | **smart_io** |
| **COUNTER_INPUT** | **integer 32 bits** | **read** | **smart_io** |
| **ANALOG_OUTPUT** | **integer 16 bits** | **write** | **smart_io** |
| **KEY_INPUT** | **Boolean** | **read** | **smart_io** |
| **LED_OUTPUT** | **Boolean** | **write** | **smart_io** |
| **D7SEG_OUTPUT** | **1 byte** | **write** | **smart_io** |

**Functions applied to peripheral variable :**

Peripheral output variables with Boolean values may be **set, reset or inverted** The **blink** and **no_blink** functions may be applied to LED_OUTPUT.

Integer values may be written into non Boolean Peripheral output variables**. T**he **blink** and **no_blink** function may be applied to D7SEG_OUTPUT.

**Declaration of peripheral variables :**

The declaration of **Input-Output Peripherals** provides for definition of:
- the node on which the peripheral is located
- the logical characteristics of the peripheral
- the type of the physical part of the peripheral and its local location address

(see reference manual : Input-Output Peripheral)

**Example :**

```
NODE MyNode ;
    NUMBER  = 1;
    ADDRESS = 7 ;
    TYPE    = ST1 ;
END

BINARY_INPUT InputA ;
    NODE    = MyNode ;
    ADDRESS = 203 ;   // OIO board 3rd input
    ACTIVE  = HIGH ;
    PERIOD  = 10 ;
    DEBOUNCE = 1 ;
END

BINARY_OUTPUT OutputA ;
    NODE    = MyNode ;
    ADDRESS = 201 ;   // OIO board 1st output
    ACTIVE  = HIGH ;
END


/* use in a task */

TASK TaskExample2;

    SPECS
        CYCLES = 10;
    END_SPECS

    EVENTS
        ON_EVENT InputA XEQ_SEQUENCE MySequenceA ;
    END_EVENTS

    SEQUENCE MySequence ;
        LOOP
            OutputA <- set;
            WAIT_TIME 200;
            OutputA <- reset;
            WAIT_TIME 200-10
        END_LOOP !InputA
    END_SEQUENCE

END_TASK
```

```
/* other examples with led, keys and 7 segment display */
    ....
    MyLed <- blink;
    WAIT_TIME 1000;
    MyLed <- set;

    My7segDisplay <- display("A");  // display of char A
    MyInternalWord <- 3;
    My7segDisplay <- display(MyInternalWord,1);  // display of value
    My7segDisplay <- blink;
    WAIT_TIME 5000;
    My7segDisplay <- no_blink;
    ....
```

## 4.2.8. BOOLEAN EQUATIONS

A Boolean Equation is a variable that carries the evaluation result of the Boolean Expression specifying the equation.
A Boolean Equation is invoked like other variables in using it's name. Equations can be used in any Boolean Expression or assignment statement. It is evident that **Boolean Equation are read only variables.**

When a Boolean equation is read, the expression is evaluated only if a change has occured to at least one term of the equation.

**Use of Boolean Equations:**

• **A** Boolean Equation is the way to give a name to a Boolean Expression. This is useful if the Boolean Expression has to be used more than one time.

• The LINKED OUTPUT option of a Boolean Equation is a way to copy automatically into an output (Binary Output, Led Output or DualPort Output Flag) the value corresponding to the result of the equation, each time it changes. This avoid the use of TASK or ACTIONS.

**Boolean Equations generated by the PAM compiler**

The PAM compiler generates automatically a Boolean Equation made up of the Boolean Expression used in statement like ON_EVENT ..., CONDITION ..., EXCEPTION ... Thus is done to have a reference for event handling.

**About Boolean Expressions**

A Boolean Expression may be made up of Boolean variables, equations, integer variables with comparison operators or inquire functions.

Basic Boolean Operators are as follows:

| | |
|---|---|
| + | OR operator |
| * | AND operator |
| ! | NOT operator |

Comparison Boolean Operators are as follows:

| | |
|---|---|
| > | GREATER THAN operator |
| < | LESS THAN operator |
| >= | GREATER THAN or EQUAL operator |
| <= | LESS THAN or EQUAL operator |
| = | EQUAL operator |
| <> | NON EQUAL operator |

**Boolean Equations Examples :**

```
BOOLEAN EquationA ;
    EQUATION
        (Flag1 * !Flag2) + Wordvar1 <= 456 ;
END_BOOLEAN

/* Boolean Equation with Linked Output */

BOOLEAN EquationB ;
    LINKED_OUTPUT = BinaryOutFastSpeed ;
    EQUATION
        (AxisA ? speed > 234.25) * FlagAxisAenabled ;
END_BOOLEAN

/* using Boolean Equation  */

IF EquationA THEN ...
...

CONDITION EquationA * !Flag3;
```

Parenthesis are used to specify that some operation has to be performed before others. They may also be used for improving readability

## 4.3. TASKS AND ACTIONS

Tasks and actions represent the executive part of the PAM application language.
Tasks are intended to realise some operations, sequenced by condition, and started when a related event occurs.
Actions are intended to execute some simple operations. They are started when a related event occurs or repeated while a related state stay present.

### 4.3.1. TASKS AND SEQUENCES

A task content the following parts:

• The SPECS part is the specifications part where the global behaviour of the task (number of PAM basic cycles used and optional advanced specifications) is defined.

• The EVENTS part is the list of Boolean Expressions and their related sequence names. This sequences are executed when the corresponding Boolean Expression value changes from 0 to 1.

• The main part of the task are the statements building-up one or more SEQUENCE.

• The last part is the POWERON part that specifies some statements which will be executed once during power on phase. This part is optional.

**Task structure example :**

```
TASK MyTaskName ;

    SPECS
        CYCLES = 10 ; // 10 basic cycle
    END_SPECS

    EVENTS
        ON_EVENT BooleanExpression1 XEQ_SEQUENCE Sequence1 ;
        ON_EVENT BooleanExpression2 XEQ_SEQUENCE Sequence2 ;
        ON_EVENT BooleanExpression3 XEQ_SEQUENCE Sequence3 ;
    END_EVENTS

    SEQUENCE Sequence1 ;
        ....
        ....
    END_SEQUENCE

    SEQUENCE Sequence2 ;
        ....
        ....
    END_SEQUENCE

    SEQUENCE Sequence3 ;
        ....
    END_SEQUENCE

    POWERON ;
        ...
    END_POWERON
END_TASK
```

**Task Specifications**

**CYCLES** parameter is used to specify the typical activation cycle time (time interval between two activation of the task). Refer to chapter Application Behaviour for more information.

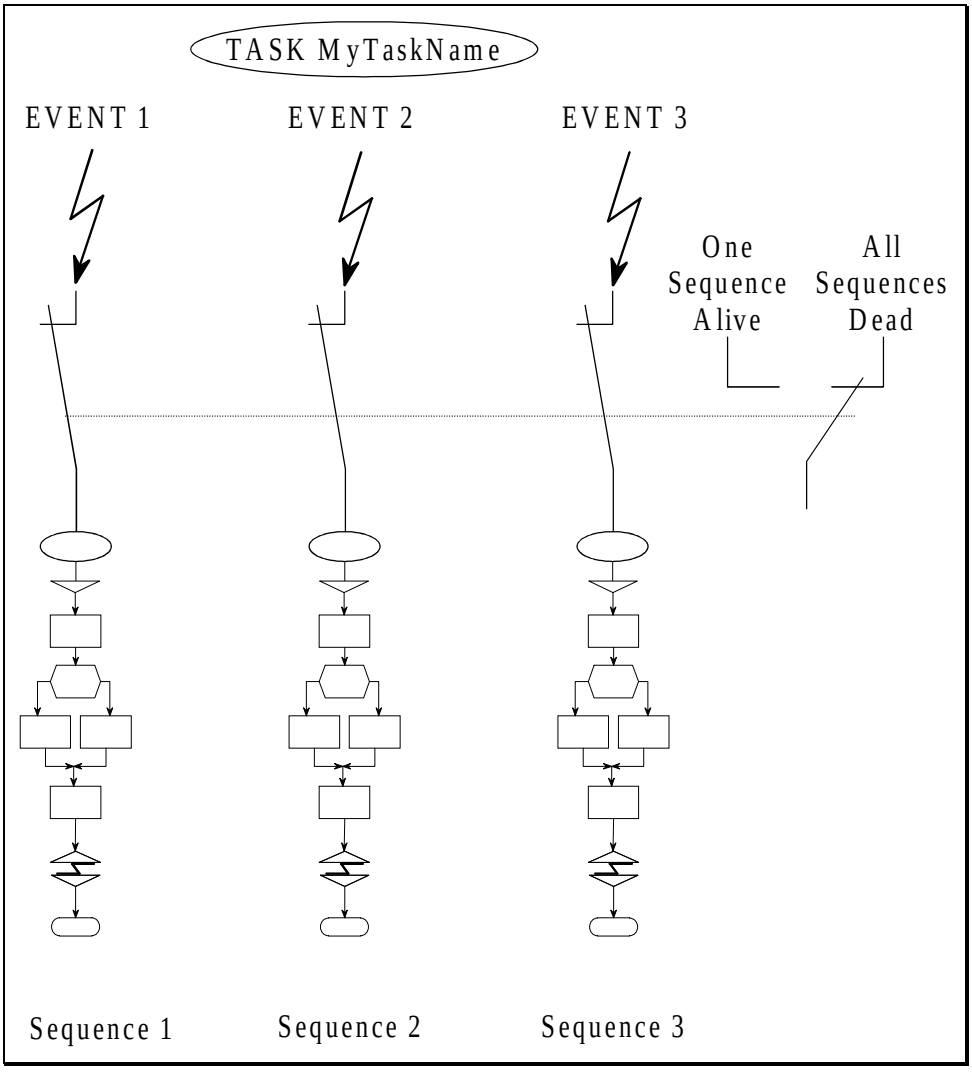
Some other task specifications are optional and will be discussed in the chapter Advanced Concepts.

**Task Execution principle**

- Only one sequence of a task can be alive at the same time.

- A sequence is executed as soon as the corresponding event occurs.

- While a sequence is alive, the other sequences of the task and the alive sequence itself cannot be activated if a start event occurs.



**Sequence made up**

Sequence are made up of two kinds of statement :

- Active statements :
  - assignment statements,
  - control flow statements.

- Services statements (event driven statements) :
  - WAIT_TIME statements,
  - CONDITION statements,
  - EXCEPTION statements,
  - etc.

**Example :**

```
SEQUENCE SequenceExample ;
    Flag1 <- set ;
    IF Flag2 THEN
        Stepvalue <- StepValue * 1.2 ;
    END_IF
    AxisA <- relative_move(StepValue) ;    // start for a position step
    /* wait until completion of the displacement */
    CONDITION AxisA ? ready TIMEOUT 10000;
    IF TIMEOUT THEN
        /* stop axis if execution time of the step exceeds 10000 ms */
        AxisA <- stop ;
    END_IF
END_SEQUENCE
```

Remarks about the example :

*The statement* `CONDITION AxisA ? ready TIMEOUT 10000 ;` *is a service statement. It means that the sequence is suspended until the expected event occurs (end of displacement) or until the 10000 ms maximum delay is elapsed.*

*The sequence is then reactivated when event occurs.*

*The sequence returns to the dead state when the* `END_SEQUENCE` *statement is reached.*

*(Refer to chapter Application Behaviour for more information's)*

### 4.3.2. ACTIONS

Actions are build up using active statements only.
This set of statements can be executed once as soon as an event occurs (ON_EVENT). It can also be cyclically executed as long the state of a Boolean expression is true (ON_STATE).

Each action has to be declared into an ACTIONS bloc. It is possible to create several actions bloc. Each action bloc gets a name.

**Action bloc structure**

• The SPECS part is the specifications part in which the user defines the global behaviour of the group of actions (number of PAM basic cycles used and optional advanced specifications).

• The list of actions. Each one beginning with an ON_EVENT or ON_STATE and ACTION, continuing with the statements to execute and ending with an END_ACTION.

• The last part is the POWERON part that specifies some statements which will be executed once during power on phase. This part is optional.

**Action bloc structure example :**

```
ACTIONS MyActionsGroupName ;

    SPECS
        CYCLES = 10 ; // 10 basic cycles
    END_SPECS

    ON_EVENT BooleanExpression1 ACTION
        ....
        ....
    END_ACTION


    ON_EVENT BooleanExpression2 ACTION
        ....
        ....
    END_ACTION

    ON_STATE BooleanExpression3 ACTION
        ....
        ....
    END_ACTION

    POWERON ;
        ...
        ...
    END_POWERON
END_ACTIONS
```

**Actions Specifications**

**CYCLES:**
- For ON_STATE action it defines the number of PAM basic cycles between two execution of the related action group.
- For ON_EVENT action, it defines the maximum number of cycles till the action can be installed in the special task that executes all actions.
(Refer to chapter Application Behaviour for more information).

Some other actions specifications are optional and will be discussed in the chapter Advanced Concepts.

**Action made up**

Statements that can be used in action are active statements :
- assignment statements.
- function calls.
- control flow statements (except LOOP and sequence execution)

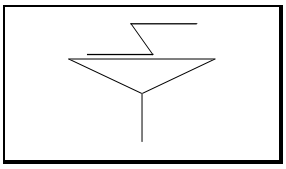
Service statements cannot be used in actions !

**Action example :**

```
ACTIONS MyActions ;
    SPECS
        CYCLES = 10 ; // 10 basic cycles
    END_SPECS
    ON_EVENT Input_16 + Input_17 ACTION
        Flag1 <- set ;
        Flag2 <- reset ;
        DualportWordOut2 <- 345;
    END_ACTION

    ON_STATE Scanforever ACTION
        IF AxisA ? error_code <> AxisAErrorCodeMemo THEN
            AxisAErrorCodeMemo <- AxisA ? error_code ;
            IF AxisA ? error_code(HARD_ERROR) THEN
                AxisAHardErrorFlag <- set ;
            END_IF
        END_IF
    END_ACTION

    POWERON ;
        AxisAErrorCodeMemo   <- 0 ;
        AxisAHardErrorFlag   <- reset ;
        Scanforever <- set ; // to enable error scanning
    END_POWERON
END_ACTIONS
```

## 4.5. EXCEPTIONS & ERRORS

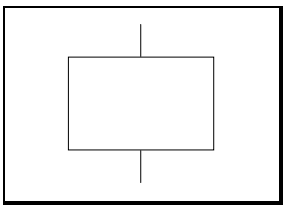
The PAM application language is provided with some statements that allow to manage exceptions treatment related to the normal treatment.



Exceptions statements are services statement that must only be used in tasks.

Exceptions statement are as follows:

- **EXCEPTION .... ENTRY**
- **EXCEPTION .... SEQUENCE**
- **EXCEPTION .... XEQ_TASK**
- **EXCEPTION .... ABORT_SEQUENCE**
- **REMOVE_EXCEPTION**



The PAM application language provide two inquire functions for error handling.

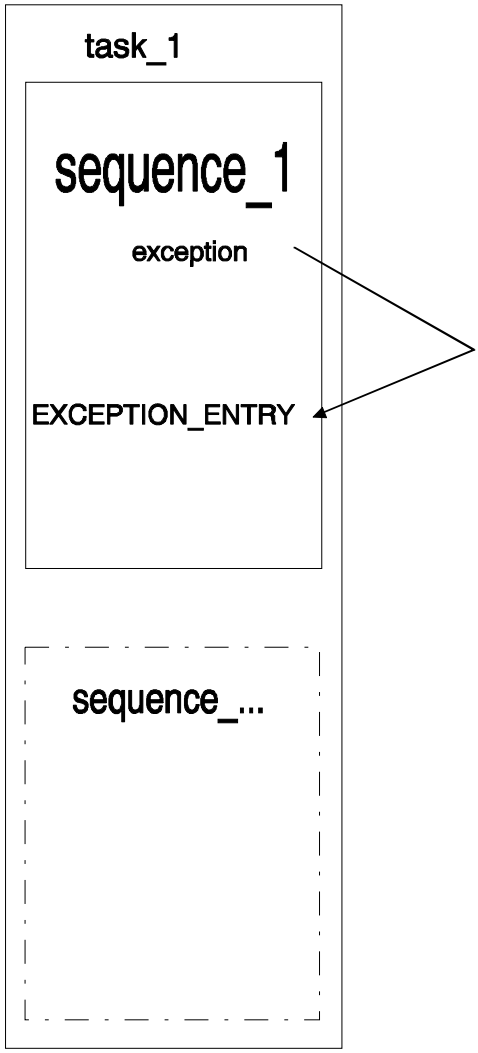
Errors inquire functions are as follows:

- **? error**
- **? error_code**

### 4.5.1. EXCEPTION ENTRY

The EXCEPTION.....ENTRY mechanism is provided to abort the working sequence and to restart it at the EXCEPTION_ENTRY label.

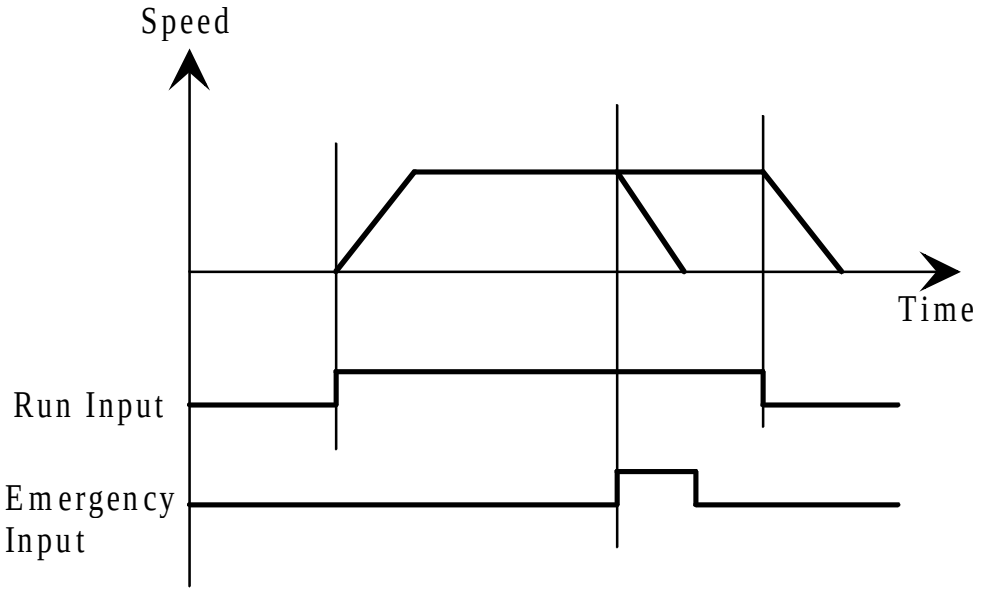
The statements between EXCEPTION_ENTRY and the end of the sequence are a common part. It is executed under normal situation or when exception occurs.




Syntax :

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

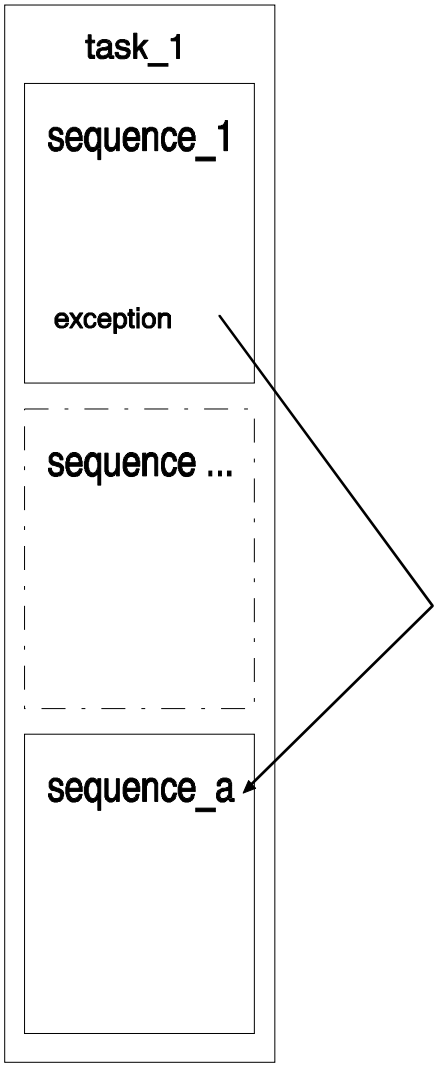
**Example : Manual rotation with emergency stop**




```
TASK ExceptionExample1 ;

   SPECS
        CYCLES      = 10;
   END_SPECS

   EVENTS
        ON_EVENT Axis1In19  XEQ_SEQUENCE Axis1Run ;
   END_EVENTS

   SEQUENCE Axis1Run ;
      EXCEPTION Axis1In20 TIMEOUT 5000 ENTRY Axis1Stop;
      Axis1  <- run(360) ;  // [ULU/s]     1t/s
      CONDITION !Axis1In19 ;
      EXCEPTION_ENTRY Axis1Stop;
      REMOVE_EXCEPTION;
      Axis1  <- run(0);
      CONDITION Axis1 ? ready ;
   END_SEQUENCE

   POWERON;
      Axis1  <- power_on ;
   END_POWERON
END_TASK   /* ExceptionExample1 */
```

### 4.5.2. EXCEPTION SEQUENCE

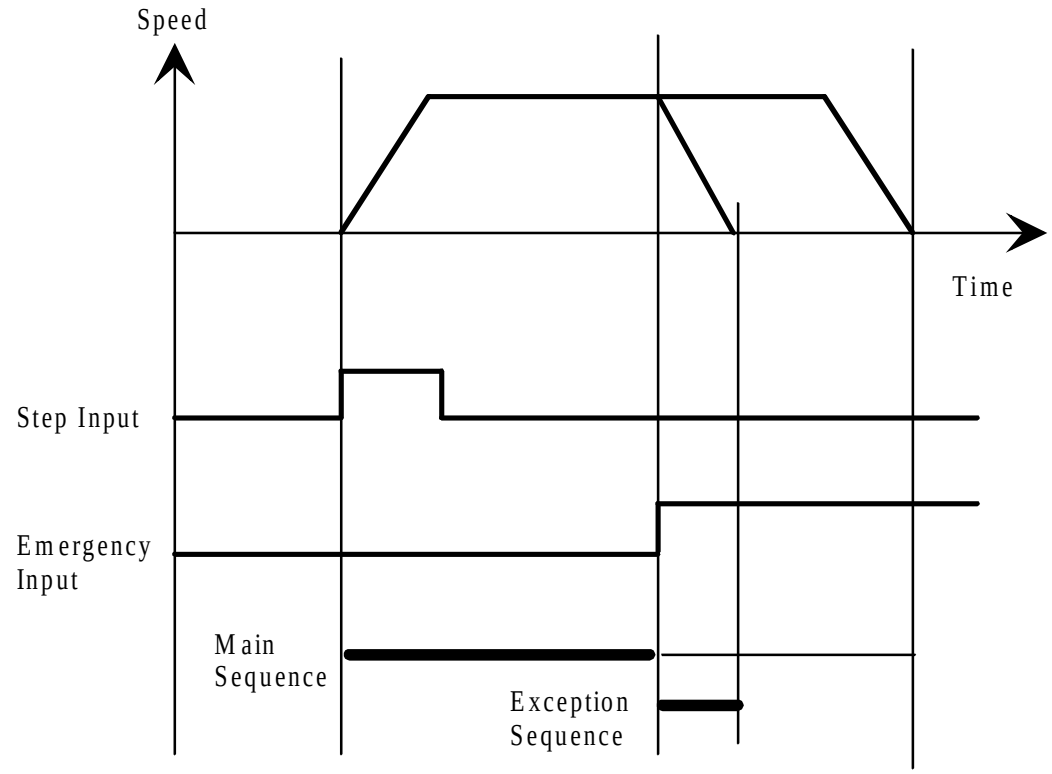
The EXCEPTION.....SEQUENCE mechanism is provided to abort the working sequence when exception occurs and to start the exception sequence.




Syntax:

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

**Example : Manual steps with emergency stop**

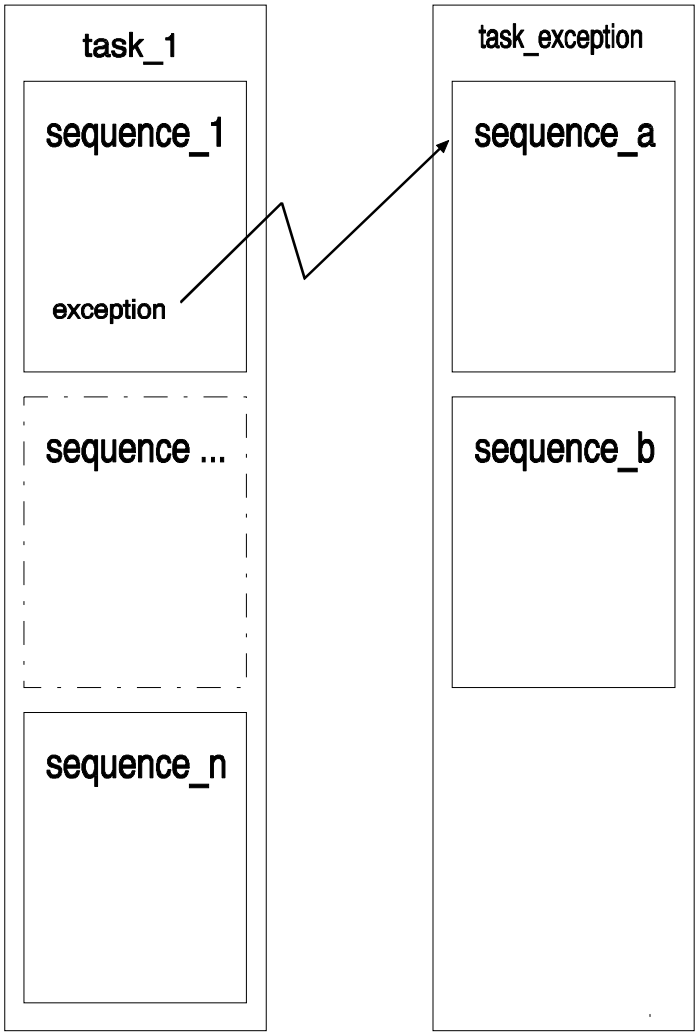



```
TASK ExceptionExample2 ;

   SPECS
        CYCLES      = 10;
   END_SPECS

   EVENTS
        ON_EVENT Axis1In21  XEQ_SEQUENCE Axis1Step ;
   END_EVENTS

   SEQUENCE Axis1Step ;
      EXCEPTION Axis1In22 SEQUENCE Axis1StopStep;

      Axis1  <- relative_move(3600) ;  // 10 turns
      CONDITION Axis1 ? ready ;
   END_SEQUENCE

   SEQUENCE Axis1StopStep ;
      Axis1  <- stop ;
      CONDITION Axis1 ? ready ;
   END_SEQUENCE

   POWERON;
      Axis1  <- power_on ;
   END_POWERON

END_TASK   /* ExceptionExample2 */
```

### 4.5.3. EXCEPTION XEQ_TASK

The EXCEPTION.....XEQ_TASK mechanism is provided to add an exception treatment to the normal treatment when exception occurs.

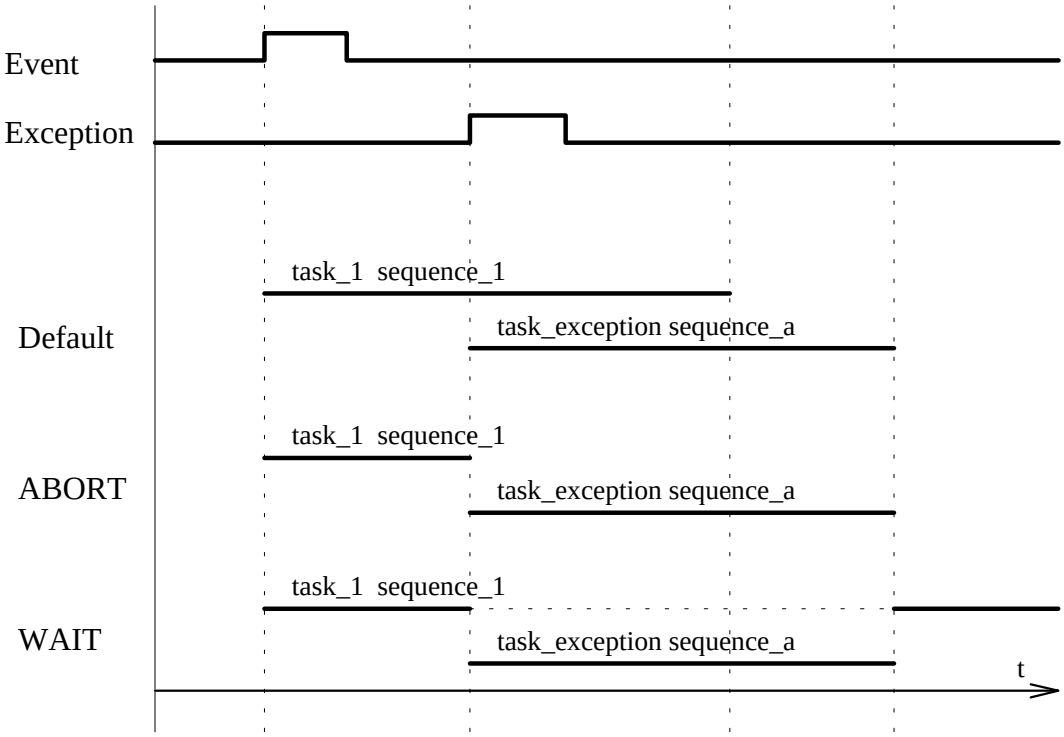
This mechanism need an exception **task** to activate the exception sequence and keep working the normal sequence.




Syntax :

**EXCEPTION** <Boolean expression> [**TIMEOUT** [<time>]]
**XEQ_TASK** <task identifier>
**SEQUENCE** <sequence identifier> [**WAIT** | **ABORT**];

**Sequences activity versus time (for default, Abort and Wait):**




## 4.5.4. EXCEPTION ABORT_SEQUENCE

The EXCEPTION.....ABORT_SEQUENCE mechanism is to be used when the active sequence must be aborted when the exception occurs.

Syntax :

**EXCEPTION** <Boolean expression>
[**TIMEOUT** [<time>]] **ABORT_SEQUENCE** :

## 4.5.5. EXCEPTION WITH TIMEOUT

The TIMEOUT can be added to any exception.
In this case there two causes to the exception:
- When the Boolean expression comes true;
- When the timeout is over.

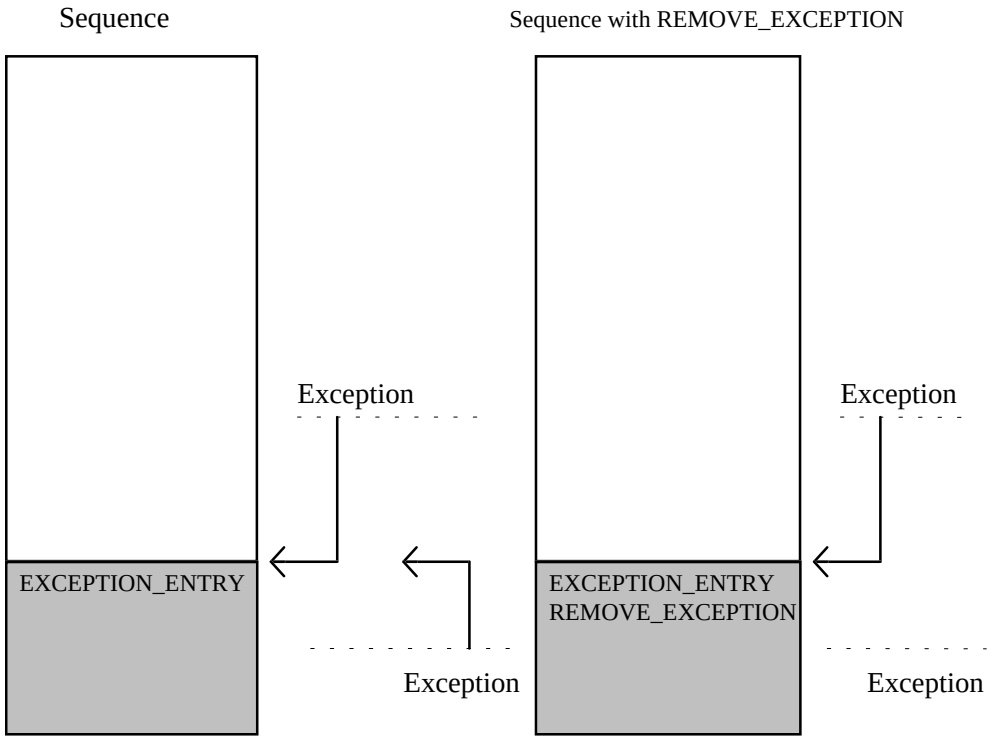
When the <time> is omitted, (normally after a statement defining timeout with a value), the current value of the timeout continue to be used without to be reinitialised.

To test if timeout as occurred is made by using:
IF **TIMEOUT** THEN ...

## 4.5.6. REMOVE EXCEPTION

syntax: REMOVE_EXCEPTION ;

It allows to cancel EXCEPTION from the REMOVE_EXCEPTION statement to the end of sequence or to another exception statement




Behaviour of the sequence without the REMOVE_EXCEPTION:

If the exception occurs during execution of the part located after the EXCEPTION_ENTRY label, this part will be executed once again from the entry label.

Behaviour of the sequence with the REMOVE_EXCEPTION:

The exception occurring during execution of the part located after the EXCEPTION_ENTRY label is discarded.

### 4.5.7. ERROR AND ERROR_CODE FUNCTIONS

Error function gives the possibility to know if an error has occured.

**ERROR function:**

Syntax :

**<object identifier> ? error**

This function returns a Boolean value:
- TRUE means one or more errors are present.
- FALSE means no error is occurred.

Object identifiers with available error function are:
- <axis identifier>, <smart_io node identifier> and <DC motor identifier>.

Example :

```
BOOLEAN AxesInError ;
    EQUATION Axis1 ? error + Axis2 ? error ;
END_BOOLEAN
```

Nevertheless, in many situations it is important to know what kind of error as occurred. The "error_code" function is provided for that purpose.
Two ways of error identification are possible:
- To read the error code value;
- To compare the error code value with a suspected error type.

**ERROR_CODE function**

Syntax:

**<object identifier> ? error_code** returns an integer (the error code)

**<object identifier> ? error_code (<error code mask>)** returns a Boolean

Object identifier with available error function are:
- <axis identifier>, <smart_io node identifier> and <DC Motor identifier>.

**Examples:**

1. assignment:
```
Axis1ErrorCode <- Axis1 ? error_code;
```

2. test of a particular code:
```
IF Axis1 ? error_code = 25 THEN ...
```

3. using error code mask:
```
ON_EVENT Axis1 ? error_code(SERVO_LAG_ERROR)
    SEQUENCE ...
```

The error_code function allows to know witch kind of error as occurred.

The value returned by `<object identifier> ? error_code` is dependant of the <object identifier>.

### 4.5.8. AXIS "ERROR CODE " DESCRIPTION

The value returned by the **? error_code,** simply called error code, is a collection of weighted bits.
When an error occurs, the corresponding bit to this error is set into the error code.

The error code is a 32 bits value for an axis located on a ST1 node. It is a collection of error bits corresponding to ABCD status of the ST1 servo amplifier, but filtered a first time by the CMASKS ST1 parameter and a second time by PAM (see "st1error.sys" file in the PAMTOOLS environment).

The syntax is the following to test a particular error:

**<axis identifier> ? error_code (<error code mask>)**
this expression returns a Boolean value.

*<error code mask> : mask value corresponding to a particular bit or bits pattern of the error code. These masks are defined into the file* ***ST1ERROR.SYS.***
(See PAM Reference manual, "System function references")

**Axis "error code mask" use:**

The include file ST1ERROR.SYS content individual mask definition for each error bit and also a mask to test if any hardware error is present :

```
#set HARD_ERROR   RESOLVER_FAILURE | OVER_VOLTAGE |\
                  POWER_STAGE_OVERLOAD | INT_SUPPLY_FAILURE |\
                  AUX_SUPPLY_FAILURE | COMMUNICATION_FAILURE
```

It is possible to create other masks in the application for special use and all mask can be redefined.

**Mask redefinition example:**

```
#set HARD_ERROR   RESOLVER_FAILURE | OVER_VOLTAGE |\
                  POWER_STAGE_OVERLOAD | INT_SUPPLY_FAILURE |\
                  AUX_SUPPLY_FAILURE
```

**Example of use** (creation of a SystemError Equation)**:**

```
BOOLEAN SystemError ;
    EQUATION
        FollowerAxis ? error_code(HARD_ERROR) +
        FollowerAxis ? error_code(SERVO_RESET) ;
END_BOOLEAN
```

Other recommended solution:

#set SYSTEM_ERROR  HARD_ERROR | SERVO_RESET

```
BOOLEAN SystemError ;
    EQUATION FollowerAxis ? error_code(SYSTEM_ERROR);
END_BOOLEAN
```

**FORBIDDEN** solution:

```
BOOLEAN SystemError ;
    EQUATION FollowerAxis ? error_code(HARD_ERROR + SERVO_RESET );
END_BOOLEAN
```

In this case the + operator performs the integer addition instead of the Boolean OR because HARD_ERROR & SERVO_RESET are integers.

## 4.5.9. ST1 PARAMETER CMASKS

This parameter selects those bits which, in case of any change in status C&D bits, sets bit 6 of status A to "1".

The axis handling by PAM needs that some bits have to be selected. For that purpose, PAM performs automatically a bitwise OR to CMASKS with the 0x130B value.

Default bits added by PAM in CMASKS are as follows:

| part C | | | | | | | | part D | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | | | | 3 | | | | 0 | | | | b | | | |
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| | | | **1** | | | **1** | **1** | | | | | **1** | | **1** | **1** |

## 4.5.10. COMMUNICATION FAILURE

The bit 4 of status C of the ST1 indicates a communication failure. This bit is set when a failure appears on the PAM-Ring.

**PAM-Ring failure:**

The following failure causes are detected:
- ring broken;
- frame received with a CRC error;
- no frames received within 50 ms.

In the case of a PAM-Ring failure, it is generally not possible for PAM to receive the new ST1 status to update its error code.

In case of ring failure the ST1 must be programmed to act independently.

**Use of CMASKU:**

If the mask corresponding to the communication failure bit is set into CMASKU, then the ST1 power stage will be switched OFF when communication failure occurs.

**Use of CMASKA:**

If the mask corresponding to the communication failure bit is set into CMASKA, then a STOP action will be performed by the ST1 when communication failure occurs.

### 4.5.11. SMART_IO ERROR CODE

The value returned by the **? error_code** function, called error code, is a collection of weighted bits.
When an error occurs, the bit corresponding to this error is set into the error code and if possible the bit is reset when the error disappears.

For a SMART_IO node, the error code is a 32 bits value that is a collection of error bits. Each error bit is corresponding to a particular value of error information sent by the node to PAM.

Kind of errors:
- system errors;
- ambient temperature too high;
- report of heat sink over temperature on I/O modules;
- report of errors on DC motors.

The syntax is the following to test a particular error:

**<Smart_io node identifier> ? error_code(<error code mask>)**
this expression returns a Boolean value.

*<error code mask> : mask value corresponding to a particular bit or bits pattern of the error code. These masks are defined into the file* ***SMARTERR.SYS*****.**
(See Reference manual, "System function references")

### 4.5.12. DC MOTOR ERROR CODE

The value returned by the **? error_code** function, called error code, is a collection of weighted bits.
When an error occurs, the bit corresponding to this error is set into the error code and if possible the bit is reset when the error disappears.

For a DC Motor peripheral on a smart_io node, the error code is a 32 bits value that is a collection of error bits. Each error bit is corresponding to a particular value of error information on dc motors sent to PAM.

Kind of errors :
- counting timeout ( no rotation after start);
- position limit reached;
- loss of position validity;
- detection of unexpected movement.

The syntax is the following to test a particular error:

**<DC Motor identifier> ? error_code(<error code mask>)**
this expression return a Boolean value.

*<error code mask> : mask value corresponding to a particular bit or bits pattern of the error code. These masks are defined into the file* ***SMARTERR.SYS.***
(See Reference manual, "System function references")

# 4.6. ERRORS MANAGEMENT TECHNIQUES

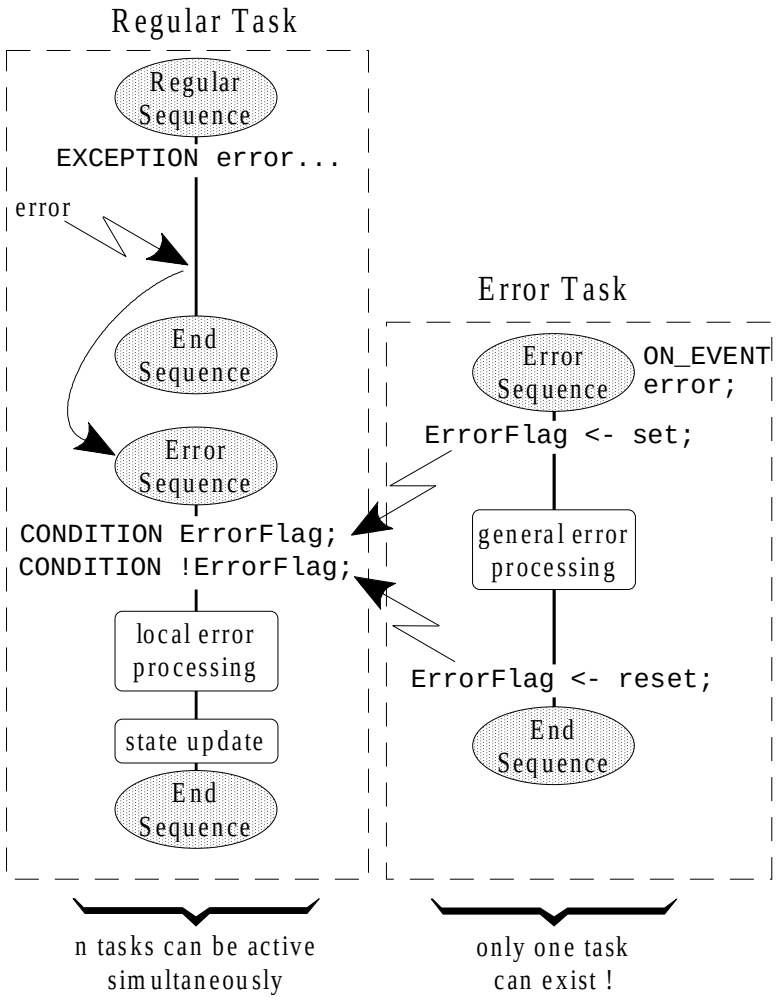
The PAM language is provided with some statements allowing different ways to manage errors into the application. Those statements have been presented in the previous paragraph.
The objective of this paragraph is to present the different techniques to implement the error management into the application.

The three main possibilities are as follows:

- centralisation of all general errors processing into one error task;
- local implementation of particular errors processing into each tasks;
- mixed implementation of the localised and centralised techniques.

## 4.6.1. CENTRALISED ERROR MANAGEMENT

The goal of this technique is to centralise all the general errors processing into one error management task ("Error Task" in the figure below). This task can be composed of several sequences. Each sequence will be automatically activated for its own by a "ON_EVENT" statement when the corresponding error occurs.




If any regular task ("Regular Task" in the previous figure) needs to react to several types of errors, this can be implemented into the tasks themselves by using exception statements.
In the previous figure, the following processing is executed when "error" occurs:

The regular sequence of the regular task is aborted and the error sequence of this task is activated in relation with the exception treatment.
At the same time, the error sequence of the error task is activated in relation with its starting rules ("ON_EVENT...").
The regular task is waiting for the termination of the error sequence of the error task execution by using a flag ("ErrorFlag").
Then the local error processing, if any, can be executed.
The last statement of the error sequence of the regular task is to update the machine state if necessary (due to the error).

The advantages of this techniques are:

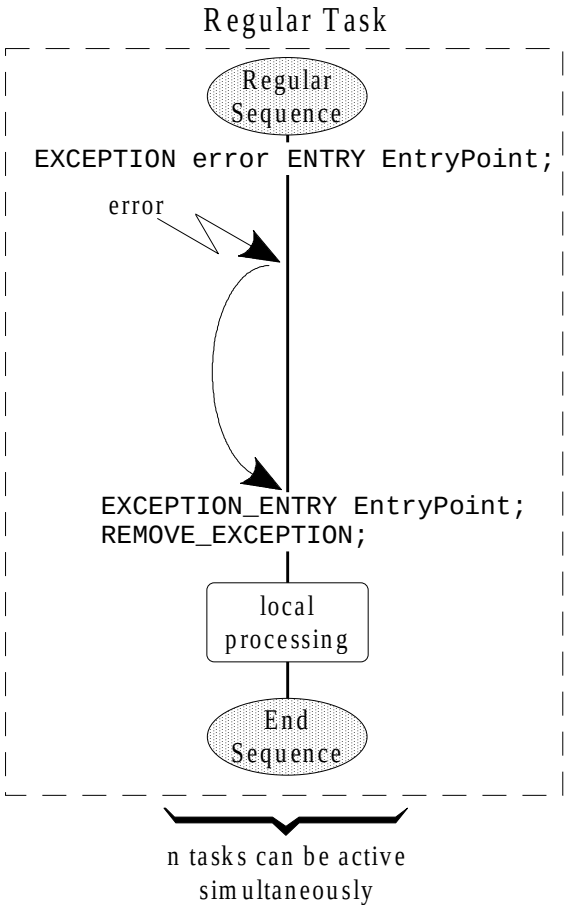
- local view of all general error processing of the whole machine;
- no code duplication;
- avoid the use of a transitory machine state because the task remains active (protected by the double CONDITION) and another sequence cannot be activated until all the error treatment is finished;
- the double synchronisation CONDITION can be placed everywhere in the regular sequence of the regular task;
- any number of regular tasks can react to the error event at the same time, but the general error processing is executed once.

Only one error task using the same error flag can exist.

### 4.6.2. LOCALISED ERROR MANAGEMENT

The goal of this technique is to localise all errors processing into each tasks ("Regular Task" in the figure below). Each sequence of the regular tasks have their own ways to react locally to any errors. The error processing is implemented into the active sequence or into a special error sequence.

The solution illustrated in the figure below can be used if no particular treatment are necessary.




In this case, the following processing is executed when "error" occurs:

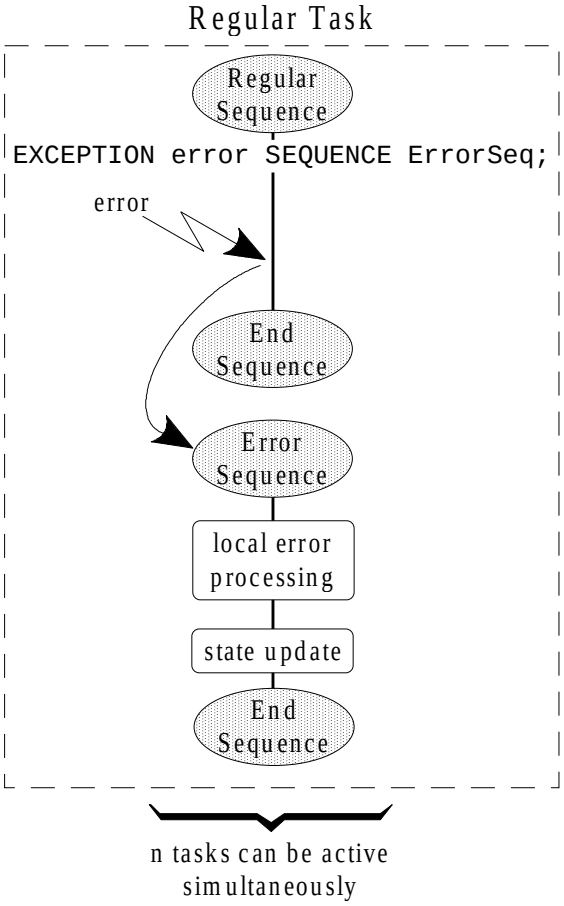
The regular sequence of the regular task is aborted and immediately restarted at the exception entry point.

Then the local regular processing, if any, can be executed.

The advantages of this techniques are:

- simplest wait to manage exception;
- local view of specific error reaction of a sub-part of the machine;
- any number of regular tasks can react to the error event at the same time.

The solution illustrated in the figure below can be used if some particular treatment are necessary.




In this case, the following processing is executed when "error" occurs:

The regular sequence of the regular task is aborted and the error sequence of this task is activated in relation with the exception treatment.

Then the local error processing, if any, can be executed.

The last statement of the error sequence of the regular task is to update the machine state if necessary (due to the error).

The advantages of this techniques are:

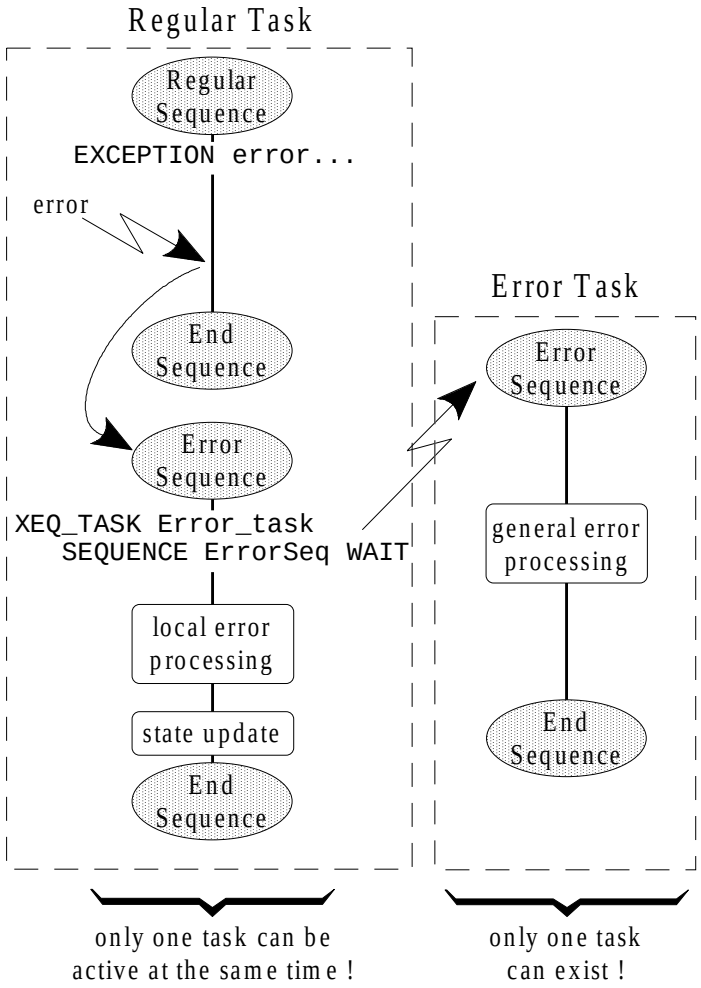
- local view of specific error processing of a sub-part of the machine;
- avoid the use of a transitory machine state because the task remains active and another sequence cannot be activated until all the error treatment is finished;
- any number of regular tasks can react to the error event at the same time.

### 4.6.3. MIXED ERROR MANAGEMENT

The goal of this technique is to localise all specific errors processing into each tasks ("Regular Task" in the figure below) and to centralise all the general errors processing into one error management task ("Error Task" in the figure below).

Each sequence of the regular tasks have their own ways to react locally to any errors. The error processing is implemented into a special error sequence.
The error management task can be composed of several sequences which are executed under control of the regular tasks.

This solution is illustrated in the figure below.



In this case, the following processing is executed when "error" occurs:

The regular sequence of the regular task is aborted and the error sequence of this task is activated in relation with the exception treatment.

Then the local error processing, if any, can be executed and the general error processing can be started from anywhere by using the "XEQ_TASK..." statement.

The local error processing is interrupted until the complete execution of the general error processing sequence.

The local error processing, if any, can finish its execution.

The last statement of the error sequence of the regular task is to update the machine state if necessary (due to the error).

The advantages of this techniques are:

- local view of specific error processing of a sub-part of the machine;
- local view of all general error processing of the whole machine;
- no code duplication;
- avoid the use of a transitory machine state because the task remains active and another sequence cannot be activated until all the error treatment is finished;
- any number of regular tasks can react to the error event at the same time.

Only one sequence can start the same error sequence. So only one regular task can be active at the same time. If another sequence try to start the same error sequence while it is already under execution, a PAM error will occurs.

# 5. MOTION CONTROL PRINCIPLES

## 5.1. ABSTRACT

This chapter oversees the motion control possibilities with PAM. First some features to move one axis independently will be presented.

Then the second part of this unit will introduce the pipe concept and pipe objects.

## 5.2. AXIS DECLARATION AND UNITS

In order to use an axis, the first step is to declare it. The basic properties of the axis are specified inside the declaration part.

The first step is the choice of a **suitable** unit. Some advises for a good choice are given below.

- *The unit does make sense for the machine.*
- *The unit must be related with the final mechanical part instead of any intermediate part (the motor shaft is an intermediate part).*
- *The unit must be as natural as possible for the user.*
- *The unit must be chosen as soon as possible and must not be changed during the program lifetime.*

When the unit is chosen, it is time to determine the relation between one chosen unit and the internal unit of the related ST1.
With ST1, the internal unit is defined as follows:

- *One electrical revolution of the motor resolver (one-speed type) corresponds to $2^{32}$ ST1 internal unit (RU).*

**EXAMPLE:**

Assume that a machine has an X axis that is moved by a ball screw with a pitch of 10 mm. The motor drives directly the screw and it is fit out with a one speed resolver. The programming unit must be 0.01 mm.
The unit factor is:

$$U_f = \frac{2^{32} * \frac{Ru}{turn}}{10 * 100 * \frac{0.01mm}{turn}} = 4294967.296 \frac{Ru}{0.01mm}$$

The other values specified at declaration time describe the standard dynamics of the axis.
In this example, the standard speed equal 2 m/s and the standard acceleration equal 6m/s$^2$. The declaration will be as follows:

```
AXIS Xaxis;
      NODE              = XNode;
      PULSES_PER_UNIT   = 429496729.6;
      TRAVEL_SPEED      = 2000.0;
      ACCELERATION      = 6000.0
      POSITION          = 0.0;
      POSITION_RANGE    = -1000.0  300000.0;
END
```

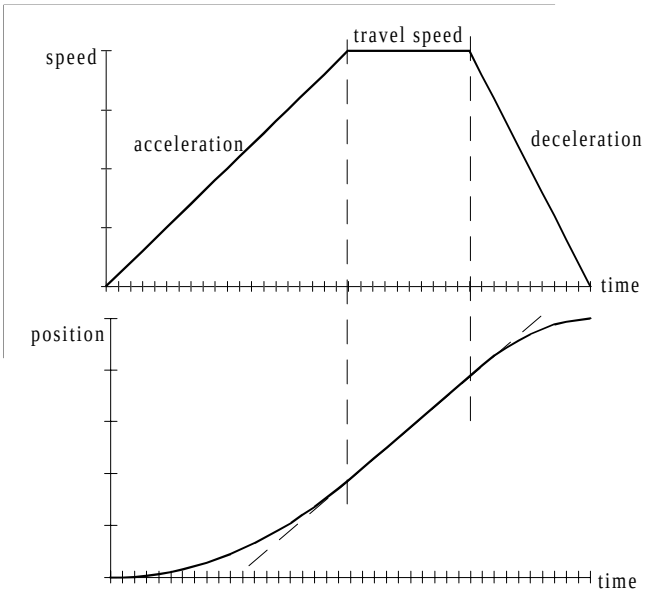
It is important to give as much as possible precision and resolution (up to 16 significant digits) for the pulses_per_unit parameter, mainly for periodic axes.

## 5.3. SINGLE AXIS MOTION

Some simple motion controls are available for single axes. They are mainly:

- run (modify the speed with the current acceleration or deceleration);
- absolute_move (go to the specified position);
- relative_move (perform the specified displacement).

All these motion controls are based on a trapezoidal profile model. It means with constant acceleration and limited speed. The maximum speed in either directions is given by the **travel speed** parameter. The **acceleration** value is used when the absolute value of the instantaneous speed is increasing (or the motor is increasing its potential energy). Conversely, the **deceleration** value is used when the absolute value of instantaneous speed in decreasing (or the motor is decreasing its potential energy).



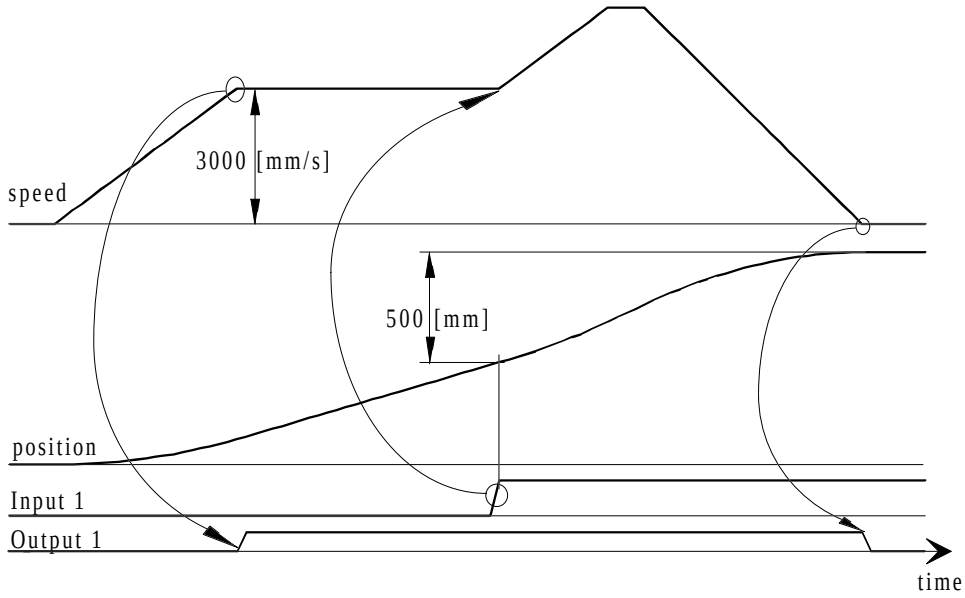

Typical trapezoidal speed profile motion

## 5.4. SYNCHRONISATION WITH A SINGLE MOTION

A typical job for a motion control or related sequence is to synchronise some controls with the evolution of a motion. In order to perform this job the PAM offers the `?ready` function for a large number of objects including axes.

**EXAMPLE:**

```
...
Xaxis <- run (3000.0);
CONDITION Xaxis ? ready;

Output1 <- set;
CONDITION Input1;

Xaxis <- relative_move(500.0);
CONDITION Xaxis ? ready;

Output1 <- reset;
...
```

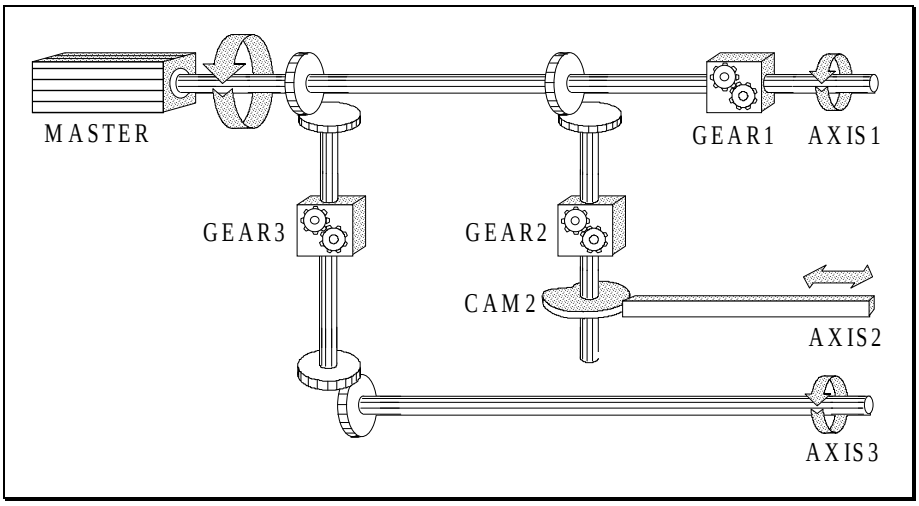



The system behaviour is as follows:
- Xaxis starts to run to reach a specified travel speed (3000);
- when Xaxis reaches its travel speed, Output1 is set;
- when Input1 becomes true, Xaxis starts a relative move with a specified amplitude (500);
- when the Xaxis has finished its relative move, Ouput1 is reset.

## 5.5. PIPES CONCEPT

Pipes is the innovative solution to solve axes synchronisation problems. This concept allows to define channels between source objects and destination objects. These channels are called pipes. The main application of the pipes concept is to describe motion profiles and relationships.

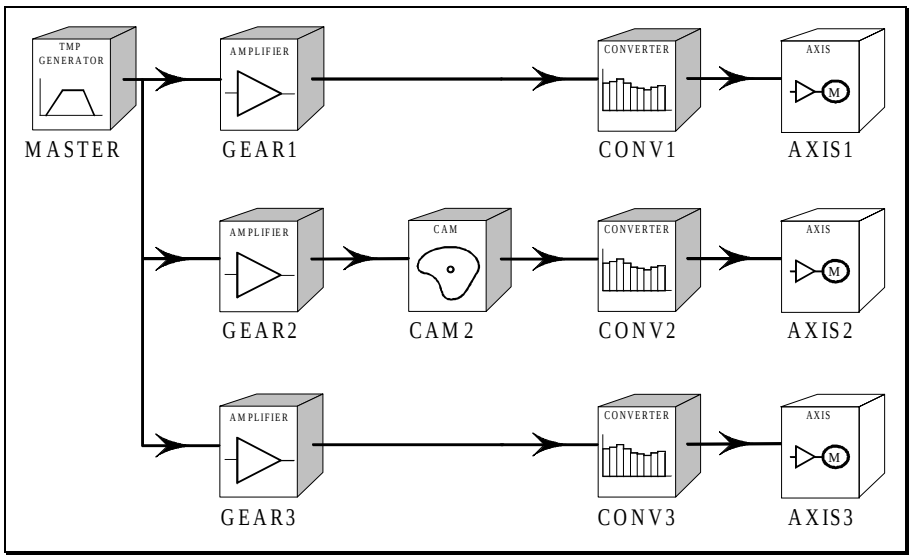
To introduce this concept, lets start with a mechanical analogy.




Pipes mechanical anlogy

In the figure above, the mechanical system is made up with three axes and drived with one main motor. All axes are connected to the motor through shafts, gears and cams. When the motor is in motion, all axes are moving synchronously.

The pipe system of the figure below corresponds to the mechanical system described above.




Pipes system

This pipe system is implemented with the PAM language in the following way:

```
/* Pipes installtion */

Conv1 <<      << Gear1 << Master ;
Conv2 << Cam2 << Gear2 << Master ;
Conv3 <<      << Gear3 << Master ;
CONDITION (Conv1 ? ready) *
          (Conv2 ? ready) *
          (Conv3 ? ready)

/* Run the system at 2 turn per second (for example) */

Master <- run (720) ;
```

After this short introduction, it is time to consider formally the pipes principles.

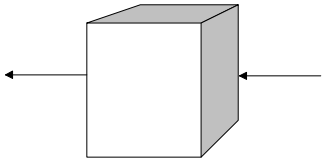
## 5.6. PRINCIPLES OF PIPES

Pipes are used to manage synchronisation between two or more axes or any number of axis and an external motions.
To be able to use correctly pipes it is necessary to consider first some definitions and rules.

### 5.6.1. PIPE BLOCK DEFINITION

For complex situation where multiple axes interact in synchronism, PAM offers the pipe concept. Pipes are made up a special kind of objects: the pipe block.

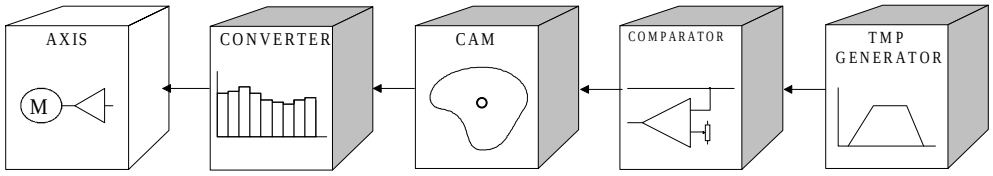
A pipe block is graphically represented by a box with grey shaded lateral sides.



A pipe block is a PAM object whose purpose is to modify a flow of values with strict time constraints. Pipe blocks have generally input and output flow of values. They can be connected together to build up pipes and networks of pipes.

### 5.6.2. PIPE DEFINITION

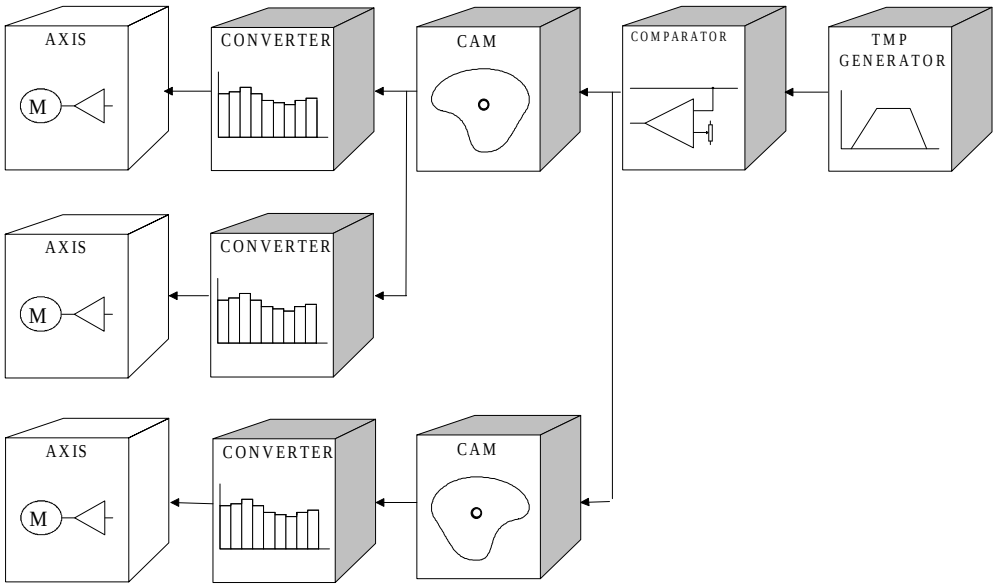
A pipe is set of chained pipe blocks. The first one is an input pipe block. It is followed by any number of transformer pipe blocks. The last block is an output pipe block. Particularities of the different kinds of pipe blocks are discussed later.




Pipe example

### 5.6.3. NETWORK OF PIPES DEFINITION

A network of pipes is a set of pipes that are dynamically interconnected. It means that all pipes are executed at the same time during application execution.




Pipe network example

### 5.6.4. GENERAL REMARKS

Note that the graphical representation does not show the time relations. It means that a dynamic set of pipes (running together at the same time) and a static set of the same pipes (not running at the same time) look out identical.

As indicated before, pipes are calculating periodically values. These values are then applied to axes in order to perform complex motions. As the flow of value is discrete, all the properties of discrete systems are applied to pipes. Nevertheless, most of design can be made by thinking about a continuous system.

Pipe blocks are declared like any PAM object. At declaration time the main parameters are specified.
Unlike the other PAM objects, pipes are dynamically constructed and activated when the corresponding statement located in a sequence is executed.

Graphical representation of a pipe network will be presented further with information going from the right to the left to be similar to the pipes implementation in the application.

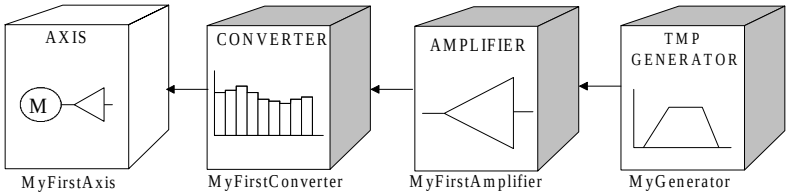
## 5.7. BUILDING THE FIRST PIPE

A pipe begins with an input block. This block produces the initial flow of values, and therefore gives to all the pipe its period of computation.
Let's take a TMP generator block called `MyGenerator`.

The second block of the pipe will be an amplifier called `MyFirstAmplifier`. Any value produced by the generator will then be amplified by this block. The ratio is specified in the declaration of the amplifier.

The output of the second block is now applied to the axis object called `MyFirstAxis`. The values produced by the two previous linked blocks have to be considered as a succession of position set values for the axis. To reach this goal, a converter block used in position mode is requested. It is called `MyFirstConverter`.

The graphical representation is as follows:

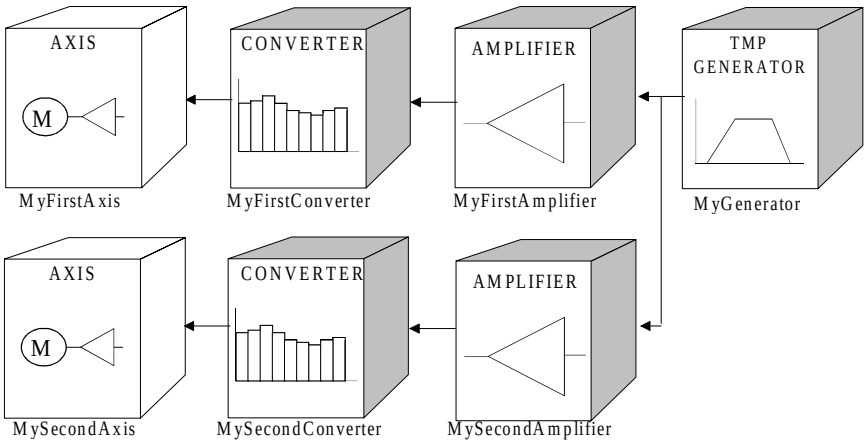



And the corresponding line of code in the application program:

```
MyFirstConverter << MyFirstAmplifier << MyGenerator;
```

This example of a pipe use is not very meaningful. Indeed the possibilities are the same as they are with single axis motion features. The only difference is that control (run, absolute_move, relative_move) are to be applied to the generator instead of the axis directly.

For the second example, a second axis is added. These two axes are moving always in the same way, with different amplitudes, keeping their synchronism. In this case it is necessary to use a pipe network.
The structure of this pipe network is as follows:




And the corresponding line of code in the application program:

```
MyFirstConverter  << MyFirstAmplifier  << MyGenerator;
MySecondConverter << MySecondAmplifier << MyGenerator;
```

**CONCLUSION:**

Each motion control command sent to the generator implies a synchronous motion on both axes, with respect to their own ratio.

Starting from this example, any modification can be made to the pipe network structure to reach any kind of synchronism required by an other application.

## 5.8. RULES TO BUILD UP PIPES

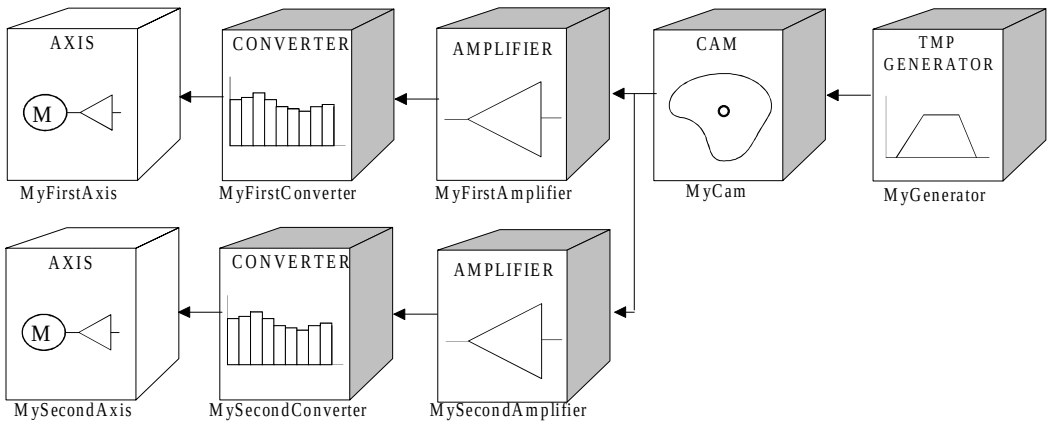
Building pipes to manage the computation of values' flows is a very powerful and flexible tool. Nevertheless, some rules have to be respected while building up pipes. These rules ensure a correct and reliable application execution. A second purpose of these rules is to allow PAM to do suitable optimisations.

**RULE 1**

Each block can only be active once at a time. It means that if two pipes are using the same block at the same time, all blocks preceding this one have to be common (the same).

In the previous example, the pipe block `MyGenerator` is shared by both pipes. Indeed the same generator is used for both axis to insure synchronism.

Another example could be the following:




```
MyFirstConverter  << MyFirstAmplifier  << MyCam << MyGenerator;
MySecondConverter << MySecondAmplifier << MyCam << MyGenerator;
```

**RULE 2**

Two output blocks with the same destination object cannot be active at the same time. This defines the mutual exclusion of output blocks having the same object destination.

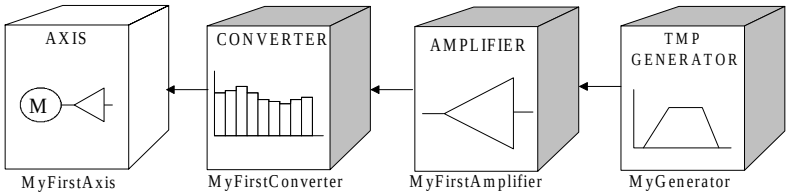
While the infringement of the first rule causes a compilation error because the pipes creation is impossible, the infringement of the second one is considered as an implicit command to disactivate the first pipe before activating the second one.

Let's consider the following example:

At a given time the statement

```
MyFirstConverter  << MyFirstAmplifier  << MyGenerator ;
```

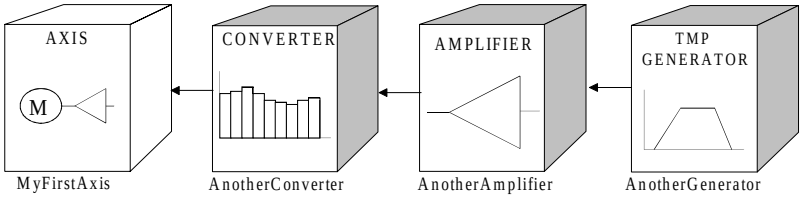
is encountered. The PAM builds the specified pipe.




Later the statement

```
AnotherConverter  << AnothertAmplifier  << AnotherGenerator ;
```

is encountered. As the destination axis of both converters are the same, the PAM disactivates the first pipe and builds the second one.




It is recommended to write explicitly in the application a "`disactivate`" of the current pipe before activating another one with the same destination.

## 5.9. LIFE OF BLOCKS

The life of a block is the time during witch the block has memory (data) assigned to it.

The life of a block begins (birth time) as soon as the creation-activation statement is executed. At this time the history of the block begins. It means that all properties are set to initial values whether or not the block as been used before. Technically, memory (data) are assigned to the block.

As long as the time increases, the history of the block runs. Each event related to the block can also modify its history. For some blocks the behaviour is closely related to its history (TMP generator for example) because the same command at different times may produce a different effect. For other blocks the history does not significantly modify their behaviour (amplifier for example). Indeed the same command at different time will produce similar effects independently of what has happened before.

The end of life (death time) of a block happens either when explicitly asked by a disactivate control or when implicitly asked by activating another pipe with the same destination object. At this time the history of the block is lost. Technically, memory (data) are released.

## 5.10. PERIOD AND PHASE OF EXECUTION

The period of execution of a pipe is the time spent between two successive computations of set values for the same pipe. The period of execution of a pipe is specified by the parameters of the input pipe block.

The absolute phase of execution of a pipe is the elapsed time between any fixed reference and the next computation for the specified pipe.

The relative phase of execution between two pipes is the elapsed time between the computation of the first pipe and the next computation for the second one.

The relative phase of execution between two pipes of the same network of pipes block is zero.
The phase of execution between two network of pipes cannot be specified by the user and depend of the pipe activation time of the application execution.

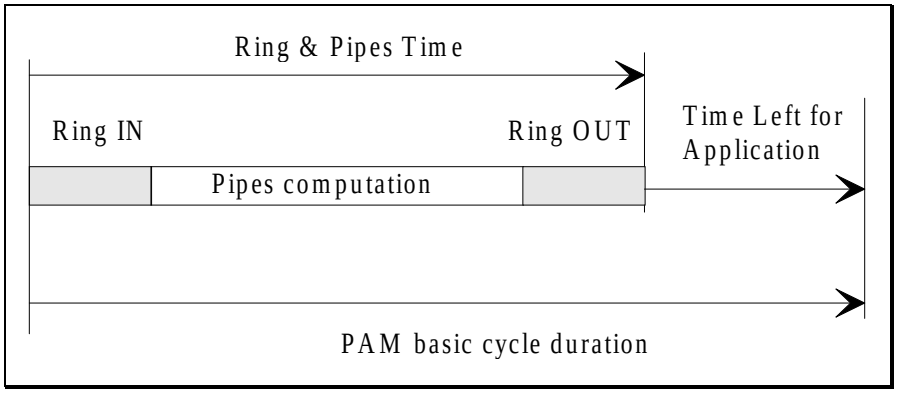
The period of a pipe is given in seconds and must be an integer multiple of the PAM_BASIC_CYCLE expressed in seconds.

**EXAMPLE:**

If the PAM_BASIC_CYCLE = 3 (0.001 s), the pipe period can be 0.007 s. For instance, a period value of 0.0075 is not allowed.

## 5.11. PIPES COMPUTATION

The computation of pipes takes place in PAM basic cycle after ring input phase and before ring output phase. This guarantees a minimum reaction time between input and output.




Pipes computation an PAM cycle

The Ring & Pipes Time (see figure above) must be shorter than the PAM basic cycle duration at each PAM cycle. This condition is checked at each cycle and if the cycle is overran, PAM fails in fatal error and the application execution is stopped.

## 5.12. TMP GENERATOR

The use of a TMP generator, as suggested in the preceding examples, is very similar to the use of a single axis at the level of the commands. The difference is located at the level of the structured approach of machine controls and the role of the TMP generator.
In opposition to the independent axes approach, synchronised axes must have something to put them in synchronism. That is the main goal of the TMP generator. It represent mainly a virtual master without physical representation. In other words, no axis should be directly connected to the TMP generator.

The TMP generator gives the cadence to the machine. It starts, stops and run the machine at the desired speed.

The recommendations to use TMP generator are as follows:

- ☞ *At least an AMPLIFIER or a CAM pipe block should be inserted between each CONVERTER and the TMP generator;*
- ☞ *The TMP generator units should have a global meaning for the whole machine and should not be related to one particular axis;*
- ☞ *All axis which have to be synchronised together should be connected to the same TMP generator.*

## 5.13. CAM

The purpose of the CAM pipe block is to perform the same job as does a cam in mechanical systems. Indeed for any value of the (generally periodic) input system corresponds a value for the output system.

Imagine the input system as a shaft driving a mechanical cam. In this situation, the position of the output system is driven by the instantaneous radius of the cam.
The same happens with the cam pipe block. The input value is converted into an output value via an extrapolation from points of the cam table.

The detailed process is the following.

The cam profile is determined with a set of coordinates. An easy way to generate it is to use a spreadsheet software (Excel for example). The file produced to describe the profile must meet the standard CSV (Comma Separated Values) format. The PAMCam Utility allows you to convert this file in PAM format. See corresponding documentation (006.8028) for more details.

When an application program declares a cam pipe block, it specifies the name of the file that holds data to describe the profile. When running a pipe including a cam, PAM looks at the preceding and following given coordinates around the present input value, to compute (with assumption of constant second derivative) the corresponding output value.

The minimum number of points to describe the profile is depending of the required precision and of the possibility of each particular profile to be approximated with the chosen cam model. The experience shows that a profile defined with hundred point is often precise enough. Note that the number of points to define a profile does influence the size of a program but does not influence the computation time of the block.

An particular characteristic of PAM cams is that the shape and the size of cams are specified separately. Indeed the PAMCam Utility normalises the profile. It means that the original values of each coordinates are transformed to be bounded between 0.0 and 1.0 (including bounds). At the time a cam is declared, the name of the file specifies the profile and the other parameters specify the default size for this use.

With cyclic machine, a mechanical part driven with an axis comes back to its original position after one complete cycle (position period) of the input system (the TMP generator for example). To express this into the profile, both first and last point of the profile table must be identical (with respect to the position period).

**EXAMPLE:**

Imagine a cyclic machine with several synchronised axes. The cycle of the machine is divided in 450 units. One of those axes must have the following behaviour:

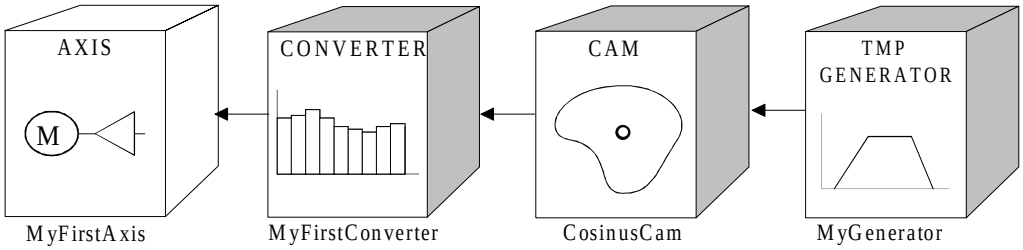
- *stay at axis position zero from the machine cycle unit 0 to the machine cycle unit 99;*
- *execute a sinus position profile from the machine cycle unit 100 to the machine cycle unit 350 and with an amplitude of one turn which must be equal to 360 axis units;*
- *stay at axis position zero from the machine cycle unit 351 to the machine cycle unit 449;*

All objects and their parameters requested for this axis are:

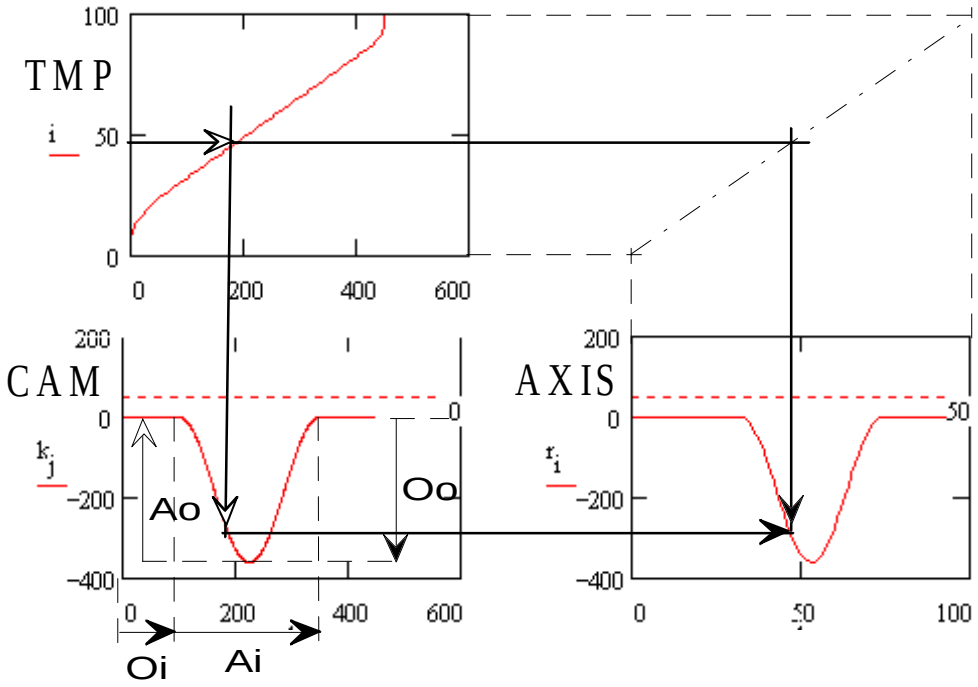
```
AXIS MyAxis ;
    NODE              = NodeOne ;
    PULSES_PER_UNIT   = 11930464.71111111  // 360 units per turn
    TRAVEL_SPEED      = 100.0 ;     // only for single axis motion
    ACCELERATION      = 500.0 ;     // only for single axis motion
    POSITION          = 0.0
    POSITION_RANGE    = - 400.0  10.0 ;
END

TMP_GENERATOR MyGenerator ;
    TRAVEL_SPEED      = 600.0 ;     // 1 cycle in 750 millisecond
    ACCELERATION      = 2500.0 ;
    DECELERATION      = 4500.0 ;
    POSITION          = 0.0 ;
    POSITON_PERIOD    = 450.0 ;     // 1 cycle = 450 units
    PERIOD            = 0.001 ;  // 1 ms
END

CAM CosinusCam;
    FILE              = cosin;
    INPUT_AMPLITUDE   = 250.0;   // duration of the sinus part
    OUTPUT_AMPLITUDE  = 360.0;   // motion amplitude = one axis turn
    INPUT_OFFSET      = 100.0;   // beginning of the sinus part
    OUTPUT_OFFSET     = -360.0;  // to have negative axis positions
END

CONVERTER MyConverter ;
    DESTINATION       = MyAxis ;
    MODE              = POSITION ;
END
```

Part of the application is as follows:

```
...
MyConverter << CosinusCam << MyGenerator;
CONDITION MyConverter ? ready;

MyGenerator <- relative_move (450.0);  // run one cycles
...
```




The axis pipe




TMP, CAM and axis motion

With:

| | |
|---|---|
| Ai: Input amplitude:<br>Input units; | Ao: Output amplitude:<br>Output units; |
| Oi: Input offset:<br>Input units; | Oo: Output offset:<br>Output units; |

## 5.14. CORRECTOR

The corrector pipe block is mainly used to compensate dynamically a mechanical sliding (belt drive for example) or to synchronise the motion with some materials also in motion and distributed in a not regular way (bottles on a blanket for example).
As the corrector is waiting for an external event which is coming from a binary input object, it is recommended to use the binary input number 16 of the ST1 OIO optional board to have the better resolution. The sampling resolution of this input is 28 microsecond.
As the ST1 OIO inputs events are sent to PAM associated with a time stamp, the PAM can compute a correlation between the position and the event with the basic input resolution.

# 6. PAM APPLICATION BEHAVIOUR

Some PAM internal mechanisms are presented in this chapter. This is to help the user to understand the application behaviour and allows the user to select correct values in tasks or actions specifications.

In the chapter 3 some basic concepts have been introduced. It is recommend to read chapter 3 before reading this chapter.

## 6.1. TASK SCHEDULING

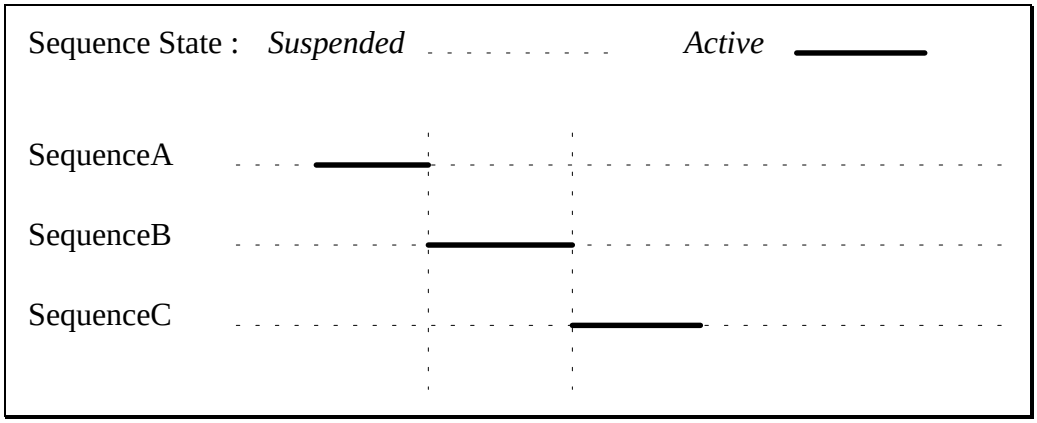
It is very important to keep in mind that when a sequence of a task is alive, the sequence may have one of two following states:

- **active** when executing active statement.
- **suspended** when executing service statement like CONDITION or WAIT_TIME.

The sequence stays in the active state for a very short time, that can be considered as zero comparing to the waiting time for an external phenomenon. In reality this short time is some micro seconds.

If the execution of sequences of several tasks is considered, an active sequence uses the processor until execution of a service statement. Then the next active sequence uses the processor.

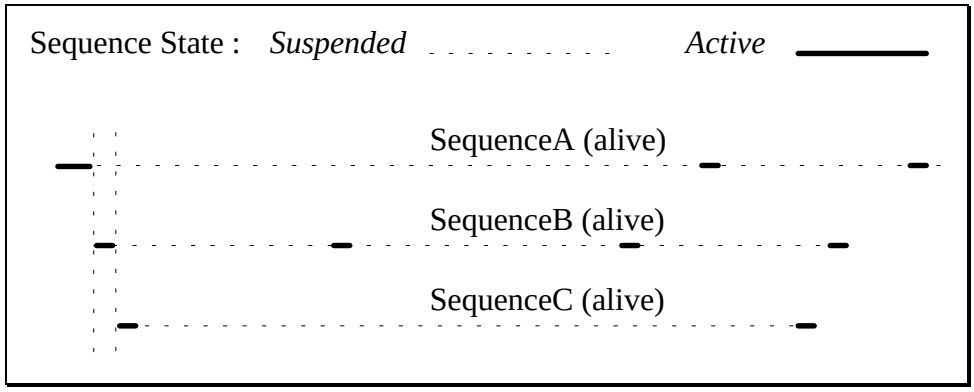
**ACTIVES SEQUENCES IN FUNCTION OF TIME:**




With the sequences A, B, C belonging each to an other task.

The diagram shows that only one sequence is active at one time. The other sequences are suspended, waiting for an event. It shows that as soon as a sequence is suspended an other one may be active.

**ALIVE SEQUENCES LARGER VIEW:**




The three sequences are alive, although each active part is executed sequentially. From an external point of view the tasks are performing their job simultaneously.

### 6.1.1. SCHEDULING BASIC MECHANISM

The task scheduling is based on two lists. The first one is the list of active tasks with a sequence in the active state, the second one is the list of suspended tasks with sequence in the suspended state.
The list of tasks to activate is explored and the execution of the sequence corresponding to the task is started. The sequence execution goes until a service statement is reached. Then the task is removed from the list of active tasks and added into the list of suspended tasks. Exploration of the list of active tasks continue and the next task is then activated.

When the event on which a sequence is waiting for occurs, the corresponding task is removed from the list of suspended tasks and added into the list of active tasks. The sequence which has to be reactivated will be active when its corresponding task will be reached in the list. This explains why the re-activation cannot be immediate and helps the PAM scheduling mechanism understanding.

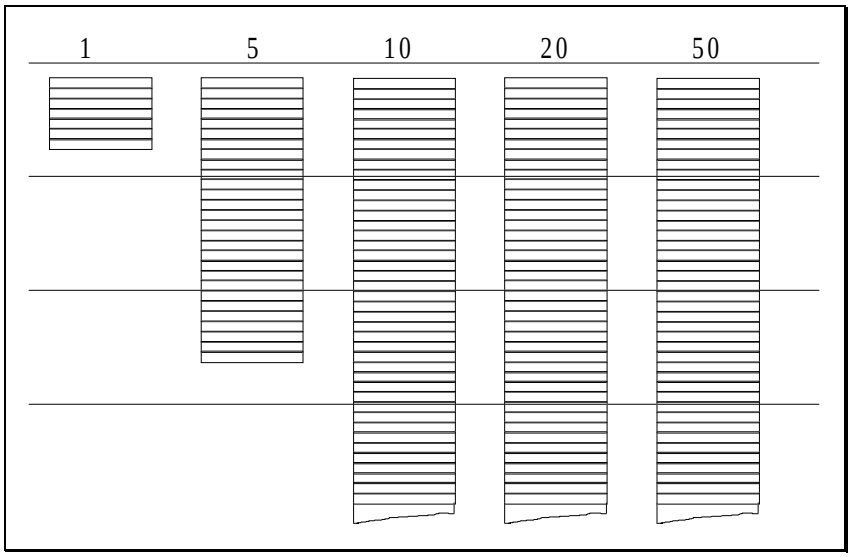
### 6.1.2. SCHEDULING CYCLES

The list of task in active state is divided in five different lists, corresponding to five different number of PAM basic cycle.

At each PAM basic cycle, all lists are explored in the following manner:

① all the items of the list of tasks with CYCLES = 1.
② 1/5 of the items of the list of tasks with CYCLES = 5.
③ 1/10 of the items of the list of tasks with CYCLES = 10.
④ 1/20 of the items of the list of tasks with CYCLES = 20.
⑤ 1/50 of the items of the list of tasks with CYCLES = 50.

This way of exploring active lists allow to increase the number of active task without increasing processor load, by increasing time interval between two successive active state of the same task.




Lists of tasks

According to these lists, the specification of number of cycles is the shortest time between two active states.

So when the event on which a sequence being suspended is waiting for occurs, there is a delay in number of cycles between the insertion of the task in the active list and the re-activation of the sequence. This delay may vary between 1 and the specified number of cycles.

### 6.1.3. NUMBER OF TASKS ALIVE SIMULTANEOUSLY

Suppose we want to load PAM with 20 tasks activated per basic cycle, it is possible for example to have the following configurations :

**EXAMPLE 1:**

| Number of tasks | Cycles number specification | Number of active tasks per basic cycle |
|---|---|---|
| **4** | **1** | **4** |
| (31 ÷ 35) **32** | **5** | **7** |
| (81 ÷ 90) **87** | **10** | **9** |
| Total = **123** | | total = **20** |

**EXAMPLE 2:**

| Number of tasks | Cycles number specification | Number of active tasks per basic cycle |
|---|---|---|
| **4** | **1** | **4** |
| (131 ÷ 140) **136** | **20** | **7** |
| (401 ÷ 450) **449** | **50** | **9** |
| Total = **589** | | total = **20** |

The total number of tasks which can be simultaneously alive is 123 for example 1 and 589 for example 2.

The number of declared tasks can be greater than the number of active tasks.

**MAXIMUM NUMBER OF SIMULTANEOUSLY ALIVE TASKS LIMITATION:**

The maximum number of simultaneously alive tasks is limited by the average maximum number of active task per basic cycle ( approximately 20 for a basic cycle of 1 ms ).

**MEMORY SPACE LIMITATION :**

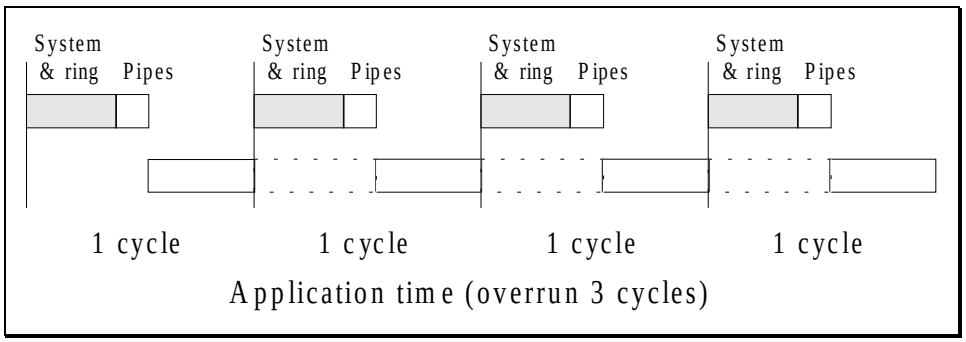
The maximum number of simultaneously alive tasks is limited by the space in memory ( approximately 150 tasks with stack of 4096 bytes)

**PEAK NUMBER OF SIMULTANEOUSLY ALIVE TASKS :**

PAM is build to withstand important peak in number of simultaneously alive tasks. In case of overload the reaction times to events cannot be guaranteed.

### 6.1.4. BASIC CYCLE OVERRUN

In case of overload, generated for example by a number of simultaneously alive tasks peak, the time left for the application in the PAM basic cycle is not sufficient. Then the application cycle will overrun the PAM basic cycle.




The figure above shows an example of an application cycle executed over 4 PAM basic cycle instead of one.

An application overrun has no effect other than a short delay in the application execution. The real time application execution will be recovered as soon as the application overload disappears.

### 6.1.5. CYCLES EFFECT ON LOOP STATEMENT

The END_LOOP statement introduces a time suspension corresponding to the number of cycles in the task specifications.

Example :

```
TASK LoopExample ;
   SPECS
        CYCLES    = 50;
   END_SPECS
   EVENTS
        ON_EVENT StartFlag  XEQ_SEQUENCE Seq1 ;
   END_EVENTS
   SEQUENCE Seq1 ;
       LOOP
           MyLed <- invert;
       END_LOOP !StartFlag ;
    END_SEQUENCE
   POWERON;
      StartFlag  <- set ;
   END_POWERON

END_TASK   /* LoopExample */
```

In this example, the led output "MyLed" state is inverted each 50 PAM basic cycles.

### 6.1.6. TASK SCHEDULING SUMMARY

- ☞ The sequence keeps the active state for a very short time, that can be considered as zero comparing to waiting for external phenomenon.

- ☞ An alive sequence spends near 100% of the elapsed time being suspended in waiting for reactivation due to phenomenon.

- ☞ The shortest suspension time is equal to the number of basic cycles in the task specification.

- ☞ The number of PAM basic cycles in the task specifications are 1, 5, 10, 20, 50.

- ☞ The PAM basic cycle may be 1 ms or more by step of 1/3 ms (1 ms is the regular value).

- ☞ An application overrun could occurs at the time of a peak of load of the application execution.

## 6.2. EVENT AND BOOLEAN EQUATION

This part explains how event are managed inside PAM trough Boolean Equations.

Service statements like:

ON_EVENT <Boolean expression> ....
EXCEPTION <Boolean expression> ....
CONDITION <Boolean expression>

generate automatically a Boolean Equation in order to evaluate the related expression.

### 6.2.1. EVENT MECHANISM

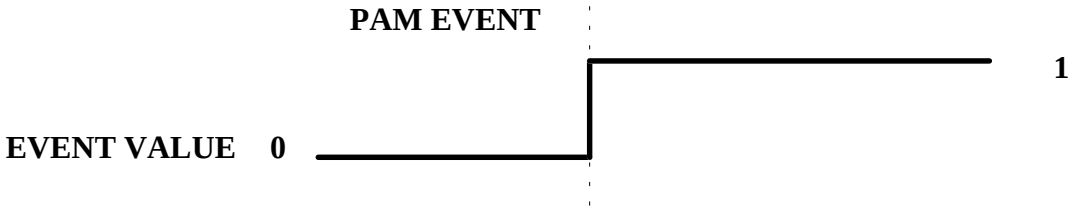
With a global point of view, an event is a change of some value **that has any importance for the system**. The importance attached to a change is due to the fact that any object is waiting for it.

Inside PAM two aspects of event are considered. They are defined as follows:

**Event message:** information telling the system that a change as occurred in a Boolean expression.

**Event value:** instantaneous value of the Boolean expression that must act on sequence or actions when event occurs.

Inside PAM an event is a state transition of the event value from 0 to 1




### 6.2.2. BOOLEAN EQUATION MECHANISM

Boolean Equations are more than the way to know the value of a Boolean Expression. At first the result of the Boolean Expression evaluation is considered as a PAM variable. Then the most important mechanism related to Boolean Equation is the link established between members of the Boolean Expression and the Boolean Equation itself.

The members of a Boolean Expression that are PAM variables have a link with the equation in order to inform the equation of their change.

The information change of members is used:

- *to know if the equation has to be evaluated when reading it.*
- *to send the event message linked to the equation.*

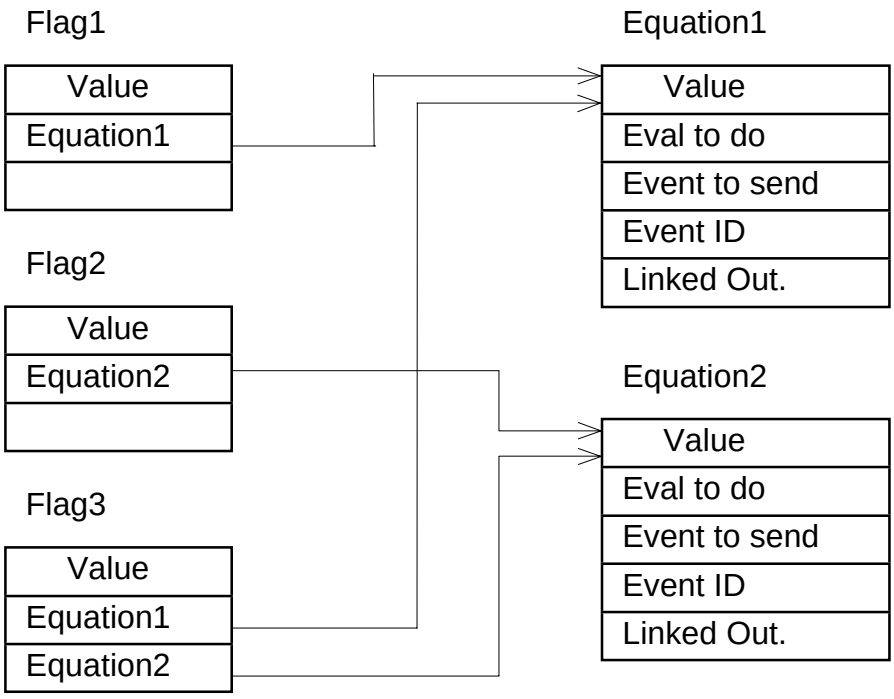
When an equation is evaluated and if the result change, the event message linked to it is sent. But this event message is sent only if the related event was enabled.

An equation event is enabled when any application object is waiting on it. For instance a sequence stopped on a CONDITION statement.

**EQUATION EXAMPLE 1:**

```
BOOLEAN equation1 ;
    EQUATION flag1 * flag3;
END_BOOLEAN
ON_EVENT flag2 * flag3 XEQ_SEQUENCE ... ;
```

Remark : the Boolean Expression `flag2 * flag3` generates a new equation (Equation2).




Equations and variables representation

### 6.2.3. BOOLEAN EXPRESSION AND EQUATION

Boolean expressions composed of PAM variables (except common variables) have the link mechanism with the equation.

Equations members or expressions that are not variables with change propagated to equations are as follows:

*common variables*
*inquire functions like:*
*<axis identifier> ? error_code(<error code mask>)*
*<axis identifier> ? position > MaxPosition*

Equation having this kind of member have an important effect on the application execution. Each time the application asks for equation result, the equation is evaluated. That means the equation must be cyclically evaluated in order to detect a change in the boolean expression, but only if a related equation event is enabled.

It is necessary to avoid frequent use of cyclically evaluated equations to prevent any application overload.

#### EQUATION WITH LINKED OUTPUT

In the Boolean Equation declaration part it is possible to specify an output linked with the equation.
When the result of the equation change, the linked output (binary output, led output or dualport flag output) is automatically updated.

Equation with linked output and with members that force evaluation, are cyclically evaluated.

Default period for cyclically evaluated equation is 50 ms.

### 6.2.4. LINK BETWEEN EQUATION AND EVENT

A link to an event is added to equations generated from Boolean expression involved in ON_EVENT ..., EXCEPTION ..., CONDITION ... statements.

In addition to the event information (static), the Boolean equation must know dynamically if the event message has to be sent or not. If the equation event must be sent, it is enabled. This is related to the fact that event are not able to act all the time.

#### EXAMPLE:

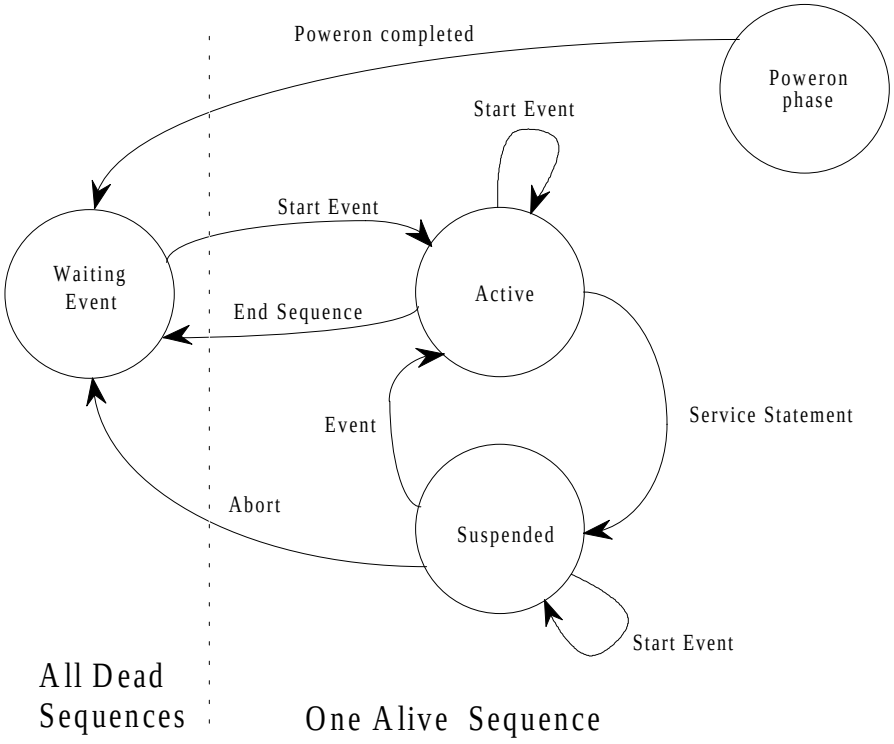
When a sequence is alive, the start event is disabled. When the sequence is dead the start event is enabled.
Within execution of a sequence, the event related to a condition statement is only enabled to be sent by the equation when the condition statement begin, until the condition event occurs.

## 6.3. EVENT AND SEQUENCES

The activity state of a sequence is event depending. In regular situation, the sequence is started by an event (except sequence called from other sequence by XEQ_SEQUENCE or XEQ_TASK statement). When the END_SEQUENCE statement is executed, the sequence is terminated and will be started again by a new event.

This chapter describes sequences and events behaviour, more precisely when "start event" may act or not.

### 6.3.1. TASK STATE



## 6.3.2. SEQUENCE AND START EVENT

In the following example :

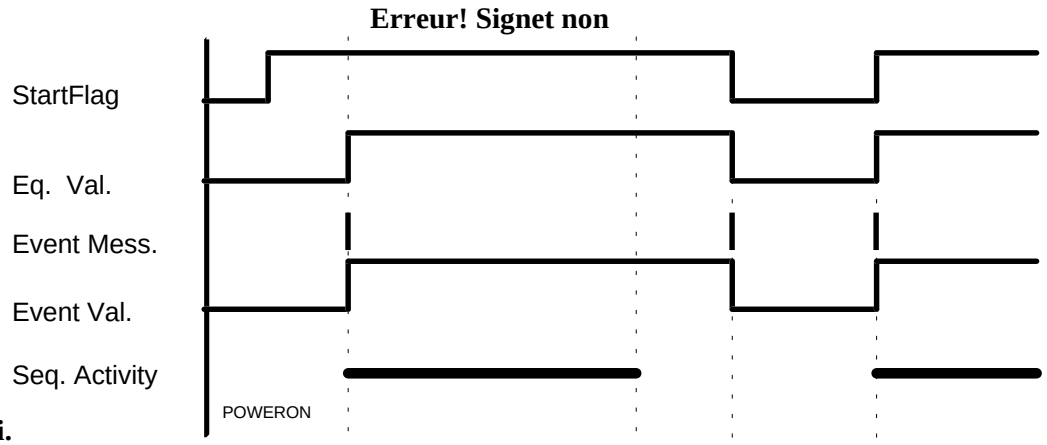
```
TASK EventExample1 ;
    SPECS
        CYCLES      = 10;
    END_SPECS

    EVENTS
        ON_EVENT StartFlag  XEQ_SEQUENCE Seq1 ;
    END_EVENTS

    SEQUENCE Seq1 ;
    ...
    ...
    END_SEQUENCE

    POWERON;
        StartFlag  <- set ;
    END_POWERON

END_TASK   /* EventExample1 */
```

The sequence Seq1 will start immediately after POWERON because the variable used in the event expression is set in the POWERON sequence.

During sequence execution the event is disabled and the system don't care of StartFlag changes.

Diagram of start event processing based on previous example.

#### A) CHANGE AFTER SEQUENCE EXECUTION

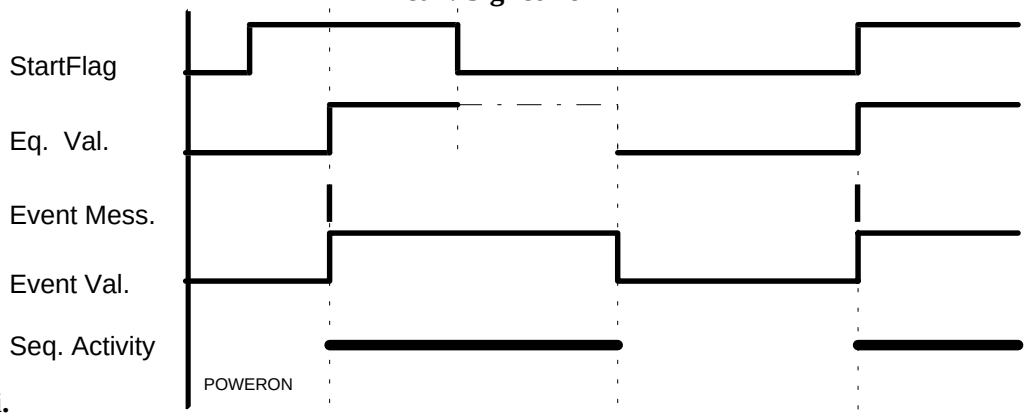



**Erreur! Signet non défini.**

Diagram of start event processing based on previous example.

#### B) CHANGE DURING SEQUENCE EXECUTION

**Erreur! Signet non défini.**

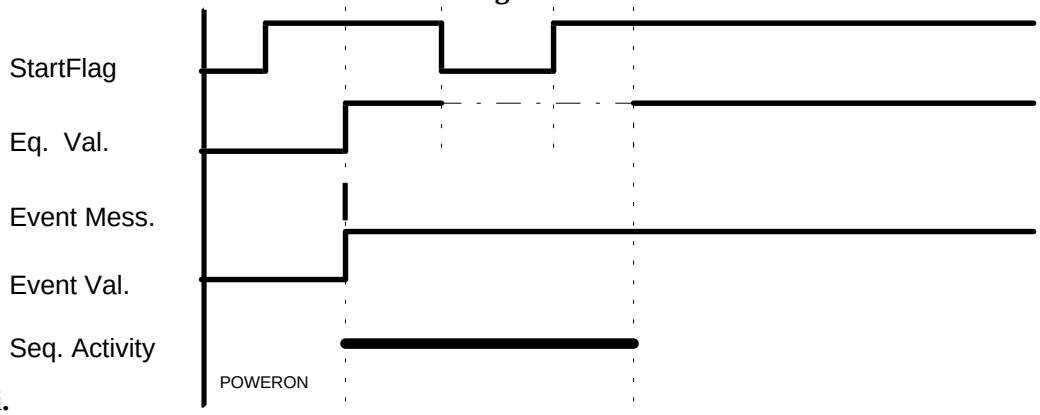



This diagram shows the event value is updated when sequence activity is completed (sequence dead). The equation is evaluated at this moment to get the event value.

Diagram of start event processing based on previous example.

#### C) LOSS OF CHANGE DURING SEQUENCE EXECUTION

**Erreur! Signet non défini.**



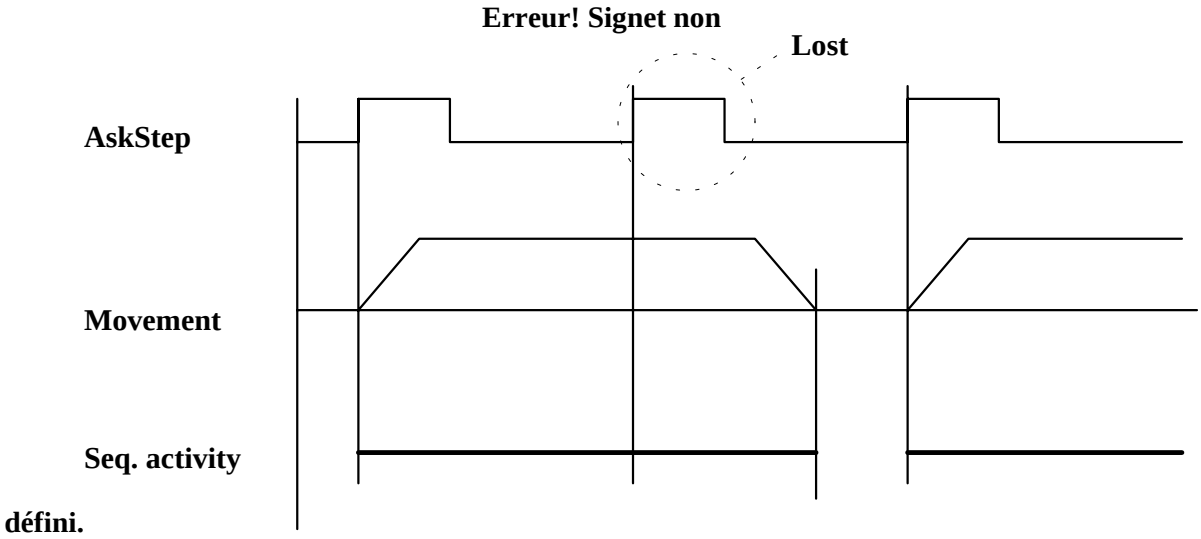

This diagram shows if the change of the Boolean expression happens during sequence execution, than this change is lost because start event are disabled during sequence activity.

### 6.3.3. SEQUENCE AND START EVENT PRACTICAL ASPECT

The following example is a task performing a jog when an input is switched from 0 to 1.

**Erreur! Signet non défini.**

```
TASK JogExample ;

   SPECS
        CYCLES      = 10;
   END_SPECS

   EVENTS
        ON_EVENT AskStep  XEQ_SEQUENCE Axis1Step ;
   END_EVENTS

   SEQUENCE Axis1Step ;
      EXCEPTION AskStopStep SEQUENCE Axis1StopStep;

      Axis1  <- relative_move(3600) ;  // 10 turns
      CONDITION Axis1 ? ready ;
   END_SEQUENCE

   SEQUENCE Axis1StopStep ;
      Axis1  <- stop ;
      CONDITION Axis1 ? ready ;
   END_SEQUENCE

   POWERON;
      Axis1  <- power_on ;
   END_POWERON

END_TASK   /* JogExample */
```




The diagram shows if the operator ask rapidly for 3 steps, only 2 will be executed.

**Events are not fiforised !**

**EXAMPLE WITH A SOLUTION TO AVOID THIS EFFECT:**

```
/* Action for step request management */
ACTIONS StepManager ;
    SPECS
        CYCLES = 5;
    END_SPECS
    ON_EVENT AskStep ACTION
        StepCounter <- StepCounter + 1 ;
    END_ACTION
    POWERON;
        StepCounter <- 0 ;
    END_POWERON
END_ACTIONS
TASK AxisJogSolution ;
    SPECS
        CYCLES      = 10;
    END_SPECS

    EVENTS
        ON_EVENT StepCounter > 0  XEQ_SEQUENCE ExecuteSteps ;
    END_EVENTS

    SEQUENCE ExecuteSteps ;
    EXCEPTION AskStopStep SEQUENCE Axis1StopStep;
        LOOP
            Axis1  <- relative_move(3600) ;  // 10 turns
            CONDITION Axis1 ? ready ;
            StepCounter <- StepCounter - 1 ;
        END_LOOP StepCounter = 0 ;
    END_SEQUENCE

    SEQUENCE Axis1StopStep ;
        Axis1  <- stop ;
        CONDITION Axis1 ? ready ;
    END_SEQUENCE

    POWERON;
        Axis1  <- power_on ;
    END_POWERON

END_TASK   /* AxisJogSolution */
```

The step counter associated with a loop inside the sequence allows to take care of all steps commands. The action is used to increment the step counter at each request. As action execution time is nearly zero, it is not possible to miss an event.

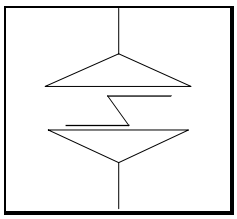
### 6.3.4. START EVENT AND POWERON

After execution of the power on phase the start events of tasks are enabled for the first time. In this particular situation, the sequence is started **if the Boolean expression is true.**
When event is enabled again the sequence start only on a change from false to true of the Boolean expression.

# 6.4. CONDITIONS AND EXCEPTIONS

This chapter gives more information about "event driven" statements like CONDITION and WAIT_TIME and more about EXCEPTION particularly when exception act during sequence execution.

## 6.4.1. CONDITIONS AND EVENT

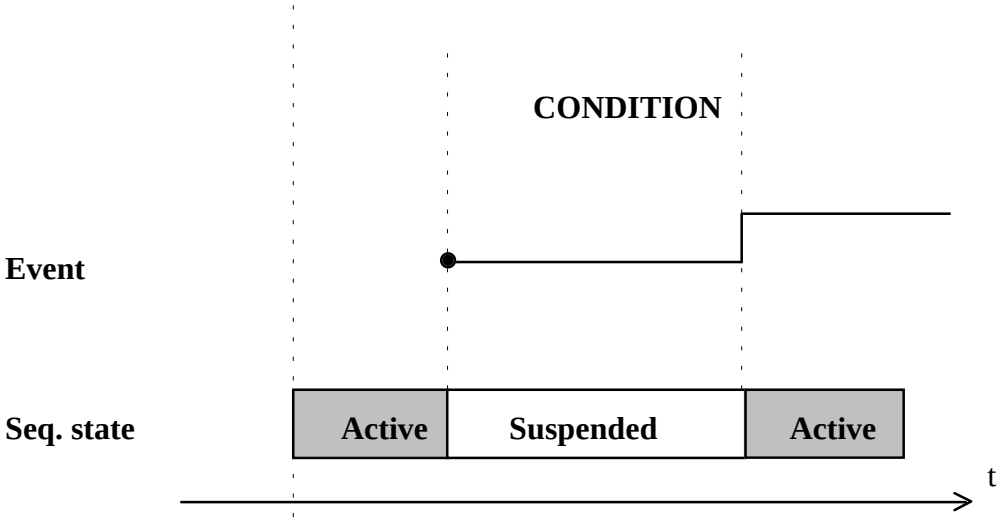


CONDITION <Boolean expression> [TIMEOUT <time>];

This service statement takes place into a sequence to perform the function:

**wait until Boolean expression is true.**

This function is implemented as follows:




The Boolean expression represents the event on which the sequence waits to be reactivated.

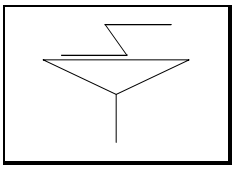
The sequence is suspended and the event enabled when the execution of condition statement begins.

The sequence is reactivated when the event occurs. There is delay between the change of the Boolean expression and reactivating the sequence. This delay is proportional to the specified number of cycles of the task.

**REMARK:**

If the Boolean expression is true before CONDITION execution, the sequence is not suspended and the statement following the CONDITION will be immediately executed.

## 6.4.2. MORE ABOUT EXCEPTIONS



Exception statements are service statements that modify the behaviour of a sequences within a task.

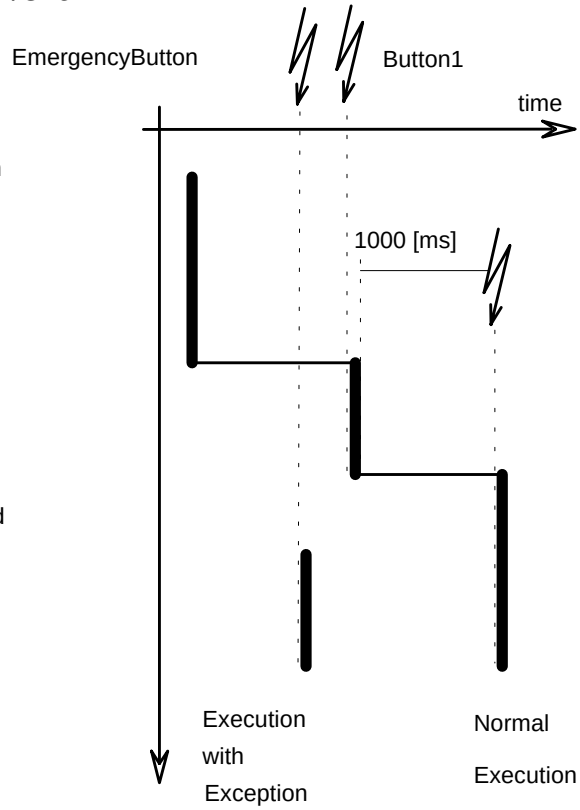
Exceptions can only act on alive tasks during the suspended state of sequences.

Two types of events could act on a sequence:
a) Events related to a CONDITION or WAIT_TIME service statement, that will be called condition events.
b) Events related to the EXCEPTION statement , that will be called exception events.

When exception in performed, the sequence is first aborted and then reactivated on a new entry point or an other sequence is activated.

### Sequence Execution and Event

```
SEQUENCE Seq1
        EXCEPTION EmergencyButton
                ENTRY Seq1End
        Led1 <- set;
        Led2 <- set;
        CONDITION Button1;
        Led2 <-blink;
        WAIT_TIME 1000;
        EXCEPTION_ENTRY Seq1End
        Led1 <- reset;
        Led2 <- reset;
END_SEQUENCE
```




**EXCEPTION EVENT PRIORITY:**
In case of a condition event and an exception event occurring simultaneously, only the action related to the exception event will be executed.

**ALREADY TRUE EXCEPTION:**
If the Boolean expression of an exception is already true at the time the exception statement is executed, the exception act immediately on the sequence without dead time.

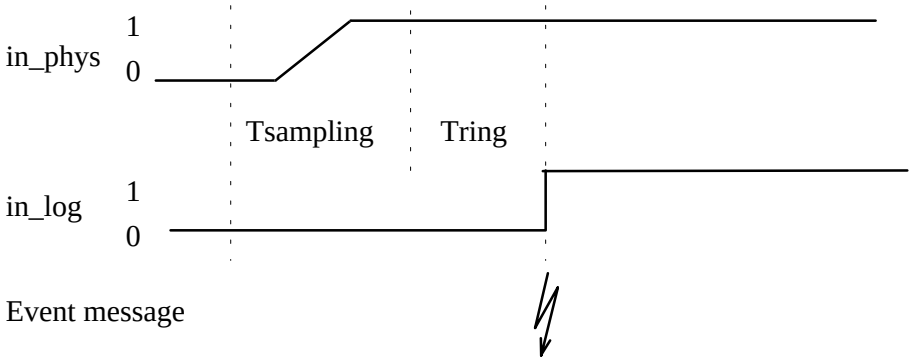
## 6.5. REACTION TIME TO EVENT

This part describes the composition of the delay between an external phenomenon occurrence and its effect on a sequence.

### 6.5.1. DETECTION TIME

If the event is based on an input change of a peripheral on the PAM ring, there is some basic cycles between the physical change and the detection inside of PAM.

Inputs have a specification of sampling period in [ms] and a debounce count may be added.




The detection time is the sum of the sampling time (Tsampling) and the overall propagation time trough the ring (Tring).

**Tsampling:**

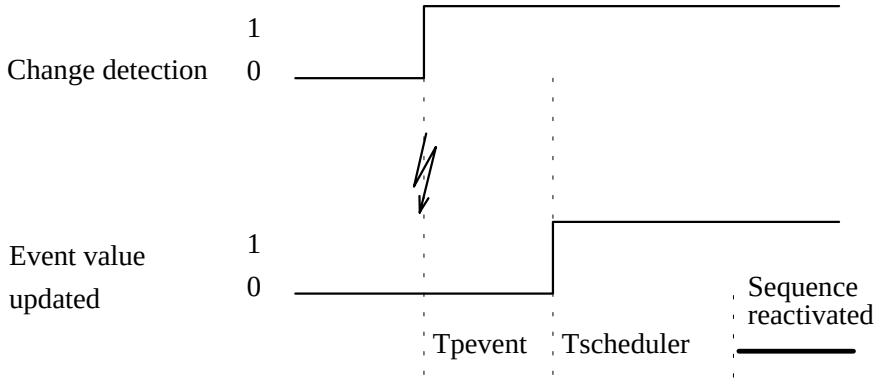
Can be between 0 and the specified sampling period of the input.
Tsampling is **n * period** if a DEBOUNCE value is given.

**Tring:**

Propagation time from peripheral to PAM trough ring, normally 3 PAM basic cycles.

### 6.5.2. REACTION TIME

The reaction time between event detection inside PAM (change detected in the Boolean expression) and the effect on a sequence may be decomposed in two parts:




**Tpevent**:

Time between detection of change (event message emitted) and processing of the message. This time is proportional to the specified number of basic cycles for the task or action. All the incoming messages are distributed in 5 lists corresponding to the 5 value of basic cycles.
**Tpevent** may be between 1 and the specified number of basic cycles.

**Tsheduler:**

Time between processing of the message and effect onto the sequence. This time is due to the multiple active task lists of the PAM scheduling mechanism.
**Tsheduler** may be between 1 and the specified number of basic cycles.

Reaction time to event (inside PAM) = **2 * specified number of basic cycle**s ( maximum value under normal load)

### 6.5.3. REACTION TIME EXAMPLE

```
TASK JogExample ;

    SPECS
        CYCLES      = 10;
    END_SPECS

    EVENTS
        ON_EVENT AskStep  XEQ_SEQUENCE Axis1Step ;
    END_EVENTS

    SEQUENCE Axis1Step ;
        ...
        ...
```

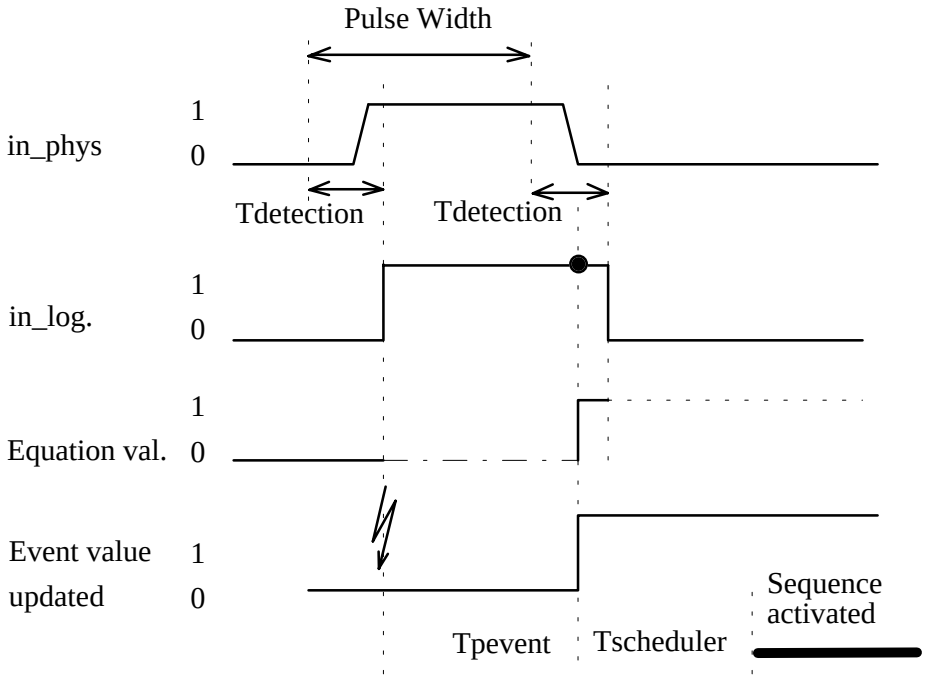
Considering AskStep is a Binary Input with a period value of 1 and with a Pam basic cycle of 3 units ( 1 ms) we get :

Detection time = 1 ms + 3 cycles at 1 ms = 4 ms.
Internal reaction = 10 cycles + 10 cycles = 20 ms ( maximum value under normal load).

Total reaction time = 24 ms from physical change to sequence activation.

## 6.5.4. MINIMUM PULSE WIDTH

In case of a pulse used as event, it is important to know the minimum pulse width that is possible to capture.




Because of the time to carry on the event message, the signal must keep the high state until the event value can be updated inside PAM.

The minimum pulse width will be **Tsampling + Tpevent.** Where Tsampling is the scanning period of the input.

**EXAMPLE :**

```
TASK StepManager ;
    SPECS
        CYCLES = 5;
    END_SPECS
    EVENTS
        ON_EVENT AskStep XEQ_SEQUENCE ExecuteStep ;
    END_EVENTS

    SEQUENCE ExecuteStep
        ...
```

Considering AskStep is an Binary Input with a period value of 1 ms and with a Pam basic cycle of 3 units ( 1 ms) we get :

Minimum Pulse width = 1 ms + 5 cycles = 6 ms

Remark: in case of PAM overload this minimal value can not be guaranteed, so it is very important to enlarge pulse as much as possible.

# 6.6. ACTIONS

An Action is a set of active statements that is managed in PAM like a sub routine call.

## 6.6.1. ON STATE ACTIONS

ON_STATE <Boolean expression> ACTION

All the calls to execute ON_STATE action body are installed in and removed from five special TASKs.
A call is installed when the state of the Boolean expression is true and removed when state is false.

The five TASKs to execute ON_STATE actions calls are corresponding to the five cycles value (1, 5, 10, 20, 50 ).

When an actions TASK is active, all actions calls are sequentially executed. This way of executing actions may introduce a peak of load.

## 6.6.2. ON EVENT ACTIONS

ON_EVENT <Boolean expression> ACTION

When the Boolean expression changes from 0 to 1, the action is installed in a TASK executed at each PAM basic cycles. The action is removed after execution.

The delay to install the action to execute is corresponding to the specified number of cycles of the actions specification. The action is executed the cycle after the cycle of installation.

The time between event detection and execution is **number cycles specified + 1 cycle**
(the installation delay may vary between 1 and the specified number of cycles).

**EXAMPLE :**

With a binary input with period of 1 ms and PAM basic cycle set to 1 ms and action specification of 1 cycle we get.

Time to detect event is 1 ms + 3 cycles at 1 ms = 4 ms.
time to install and execute action = 1 + 1 = 2 cycles at 1 ms.
Total reaction time = 4 + 2 = 6 ms.

# 7. SYSTEM CONSIDERATIONS

This chapter exposes different considerations about the physical part of the SOCAPEL multi-axes product.
Some of these considerations are for system understanding only, some other are useful for design purpose.

## 7.1. MAIN PARTS

The main parts which compose the system are:

- PAM board (for SIMATIC S5, VME, etc.)
- PAM-Ring
- Peripherals (ST1, Smart-IO, etc.)

For technical information about the different versions of PAM board, please refers to Technical Manuals.
For specific information about the peripherals, please refers to related documents.

## 7.2. PAM BOARD

The PAM board has been developed around a powerful hardware architecture based on a 32bits RISC microprocessor (Intel i960KB-20MHz).
This microprocessor allows very high rate of data management and computation due to it's integrated floating point unit and instruction cache memory.

Several types of memory are implemented around the microprocessor which are:

- 1 Mbytes of dynamic RAM for firmware and application execution,
- 128 Kbytes of fast static RAM for Kernel execution,
- 128 Kbytes of EEPROM for application storage,
- 512 Kbytes of EPROM for firmware storage.

The communication support to connect the different systems on PAM are:

- RS-232 V24 serial line for maintenance and application debug,
- RS-485 for application communication,
- 16 bits backplane bus (SIMATIC S5, VME, etc.),
- Dualport memory of 4 Kbytes,
- PAM-Ring optical fiber interface.

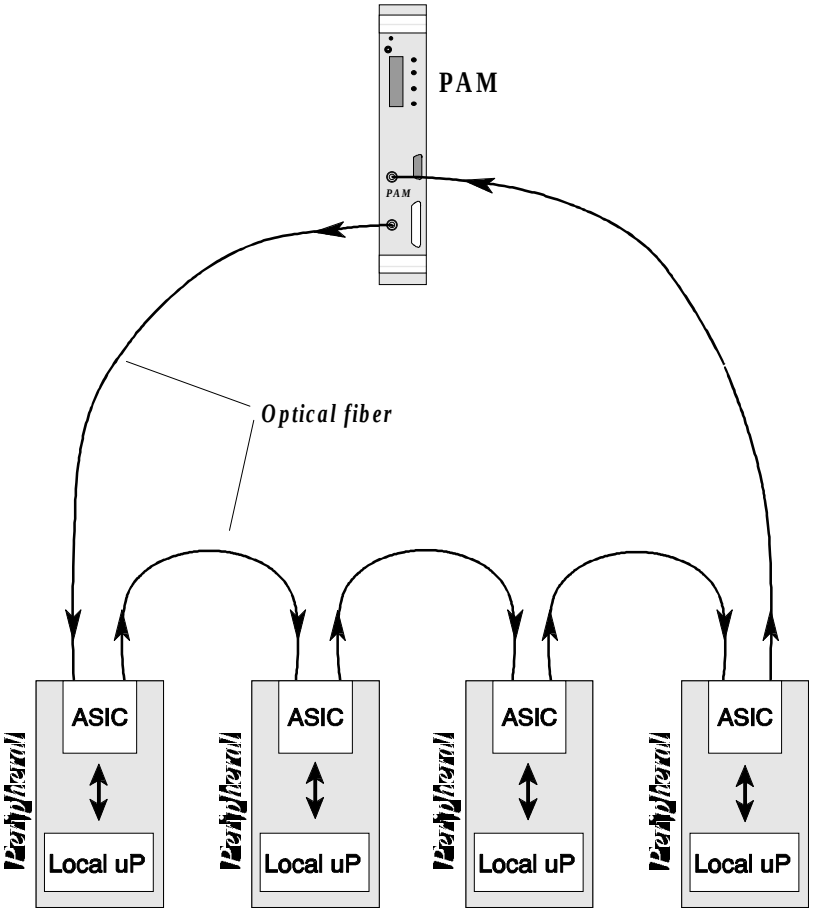
A simple human interface is provided for simple maintenance and diagnostic operation. This interface is composed of one alpha-numerical 8 digits display and a keyboard with 4 keys.

A real time clock (date and time) is integrated in PAM.

# 7.3. PAM RING

## 7.3.1. PAM-RING PHILOSOPHY

PAM-Ring first characteristic is to be configured as a ring. The information goes only in one direction, it is generated by PAM and goes out to the first node. It comes into each node, is processed by the Socapel PAM-Ring ASIC which sends information to or reads from the local microprocessor, than goes out to the next node. After the last node, the information goes back into the PAM to be processed.




PAM-Ring information flow

The basic philosophy is to send on the PAM-Ring only events or data which have been modified. That's to limit the amount of data in the PAM-Ring and to avoid any bottleneck.

PAM and it's peripherals converse between them by sending messages. In networking dialect, a message is called a **frame**.

PAM can send speed set-point or commands to axes by using a frame per set-point or command. If several axes have to receive the same set-point or command, PAM uses broadcasting or multicasting mode and needs only one frame for the axes group.
PAM can send command (set, reset, etc... ) to one output by using one frame.

A node (ST1 and Smart-IO) sends only the state of the declared IO's (by the application) and only if the state of the IO has changed. To send the event, the node generates a message and waits for a free token. When a free frame reaches the node, the slave communication manager (Socapel ASIC) takes the token and put the message into the frame, than the frame goes to the ring.

## 7.3.2. PAM-RING FRAMES

The information is composed of frame. The frames travels into the ring in a synchronous way (there is no blank, but some frames can be free and used for token communication). In other words, PAM and all nodes are always receiving from and sending frames on the PAM-Ring without interruption.

The internal composition of a frame is:

| Synchronisation byte (8 bits) |
|---|
| Mode (4 bits) and 4 bits of system use (token, extension) |
| Destination node address (8 bits) |
| 4 bytes of data (32 bits) |
| CRC for error checking (8 bits) |
| End delimiter (4 bits) |

### 7.3.2.1. THE SYNCHRONISATION FRAME:

The synchronisation frame is used to phase each ST1 (1 microsecond of accuracy) and to synchronise the communication with each node.
This frame is send as last frame of each PAM cycle. The cycle can be any multiple of 1/3 ms starting from 1 ms.
The node takes it's data only when it receives the phased synchronisation signal.

## 7.3.3. PAM-RING FUNCTIONS

The main functions of the ring are:

- PAM sends a message to one node (addressed mode)
- PAM reads a message from one node (addressed mode)
- PAM sends a message to all nodes of the same type (ST1, Smart-IO) (broadcast mode)
- PAM sends a message to a group of node of the same type (ST1, Smart-IO) (multicast mode)
- A node sends spontaneously a message to PAM or to an other node placed between itself and PAM by taking a free token on the ring (each free frame contains a token).
- PAM sends a synchronisation frame to all nodes to synchronise them.

### 7.3.4. APPLICATION EXAMPLE

Imagine a system with 10 different axes. The worst case is that the 10 axis must receive a different set-point each millisecond (ms).

① In this case, PAM uses 10 frame per millisecond to transmit the set-points to the 10 axes.

② There are 21 frames per ms, and with a basic cycle time of 1ms, there are 20 frames free for the communication with the nodes (1 frame used for the synchronisation).

③ So the free part for other communication is:
20 frames - 10 set-point frames = 10 frames.

④ With 10 frames per ms, it is possible to handle 10 inputs or outputs per ms.

⑤ For example, if PAM has to set 3 outputs in the current ms, there are 7 free frames which have a token. These 7 free frames can be used by any node to communicate an event to PAM (change on an input or change on a ST1 status). So it's possible to communicate to PAM the change of 7 inputs.

### 7.3.5. PAM-RING TECHNICAL SPECIFICATIONS

The technical specifications of the PAM-Ring are given below.

General specifications:

| | |
|---|---|
| Maximum number of nodes | 254 |
| Minimum distance between 2 nodes | 30 cm |
| Maximum distance between 2 nodes | 100 m |

Rate specifications:

| | |
|---|---|
| Frames per millisecond | 21 |
| Number of bits per frames | 68 |
| Bits per second | 1.43 Mbits/s |

Optical fiber characteristics:

| | |
|---|---|
| Code diameter | 100 µm |
| Cladding diameter | 140 µm |
| Maximum attenuation (830nm/1300nm) | 6.0/5.0 dB/Km |

# 8. PAM DISPLAY AND KEYS

This chapter describes the use of the PAM 8 digit display associated with 4 keys to display an search errors into the local queue of error messages.

## 8.1. INTRODUCTION

The eight digit PAM display and the four keys named "**Cmd**", "**Enter**", "**Up**", "**Down**", allow the user to get information's about PAM and application version and about errors and messages who did appear.

### 8.1.1. DISPLAY FEATURES

- Error, warning, message and value can be displayed.
- Date & time are added to all messages.
- PAM Language functions that allow to send messages (errors, warnings or messages) from an application sequence.
- PAM Language functions that allow to send values (monitoring) on the PAM display. (values are not sent to PAMDEBUGGER).
- All incoming messages (all types except monitoring) are stored into PAM volatile memory, in 3 list of 100 items each.
- Handling and exploring information lists contents.

### 8.1.2. SYSTEM 2.1 NEW FEATURES

- Introduction of the fatal errors management.

After PAM reset or after power up, the lists contents is lost !

### 8.1.3. MESSAGES HANDLING

When a message is emitted, the contents of the message is stored in a list and the PAM code of the message is displayed.
The messages are separated in three main classes:

ERRORS, WARNINGS & MESSAGES

There is three lists for storage.

#### 8.1.3.1. PAM CODE

The 8 hexadecimal digits of PAM message code is structured as follow:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| **O** | **O** | **R** | **X** | **T** | **N** | **N** | **N** |

With:

**OO** gives the origin of the message (00 for application message)

**R** gives an extra information, (0 means non real time message and 8 mean real time message).

**X** not used (its value is 0).

**T** gives the type of the message with:

- **M** message,
- **W** warning message,
- **F** fatal error (stored into the error list),
- **E** error message (stored into the error list),
- **S** end of error (stored into the error list).

**NNN** gives the message code number of the corresponding type (in hexadecimal).

**EXAMPLE:**

The following message is displayed: 0780E013

| | | | |
|---|---|---|---|
| OO | = | 07 | message related to the logic controller part. |
| R | = | 8 | means real time message. |
| T | = | E | means error message. |
| NNN | = | 013 | is the code number of the error. |

### 8.1.4. MONITORING OF VALUES

The PAM display can be used to monitor a value from the application. The value is displayed under floating point format and the range is from $10^{-307}$ to $10^{308}$.

**EXAMPLE:**

The value $1.23456 \times 10^{13}$ is displayed 1.234e13

### 8.1.5. USER INTERFACE

The four keys allow the user to access and select functions into menus.

#### 8.1.5.1. FUNCTIONS AVAILABLE

- scanning of errors list, selection of an item to obtain detailed information on this item ( origin, value of numerical fields, date, time),
- configuration function: scanning filter and display filters,
- reset function: clearing of lists and clear of the current displayed error,
- inquire of PAM & compiler version, application version.

#### 8.1.5.2. KEY USAGE

The "**Cmd**" key is mainly used to return from a sub menu to an upper level menu.
Exception: the Cmd key activates the main menu from the default display.
The "**Enter**" key is mainly used to select an item or sub menu
Exception: the enter key is used to show the contents of the blinking error code.
The "**Up**" key is used to move to the **previous** item of a sub menu or list.
The "**Down**" key is used to move to the **next** item of a sub menu or list.

# 8.2. DISPLAY HANDLING

## 8.2.1. DEFAULT DISPLAY

After PAM reset or at powerup, the PAM firmware version (EPROM) is displayed within the following format:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| **V** | **v** | **v** | **.** | **v** | **.** | **v** | **v** |

**"boot"** is displayed when the PAMDEBUGGER is connected and is waiting for start.

When PAM is running, application information's are scrolled onto the display (application name & version, date & time of creation).

## 8.2.2. DISPLAY OF ERRORS

The code of the first emitted error is blinking on the display. No other errors or messages are displayed.

## 8.2.3. DISPLAY OF WARNINGS

If no errors, the emitted warning codes are displayed successively.

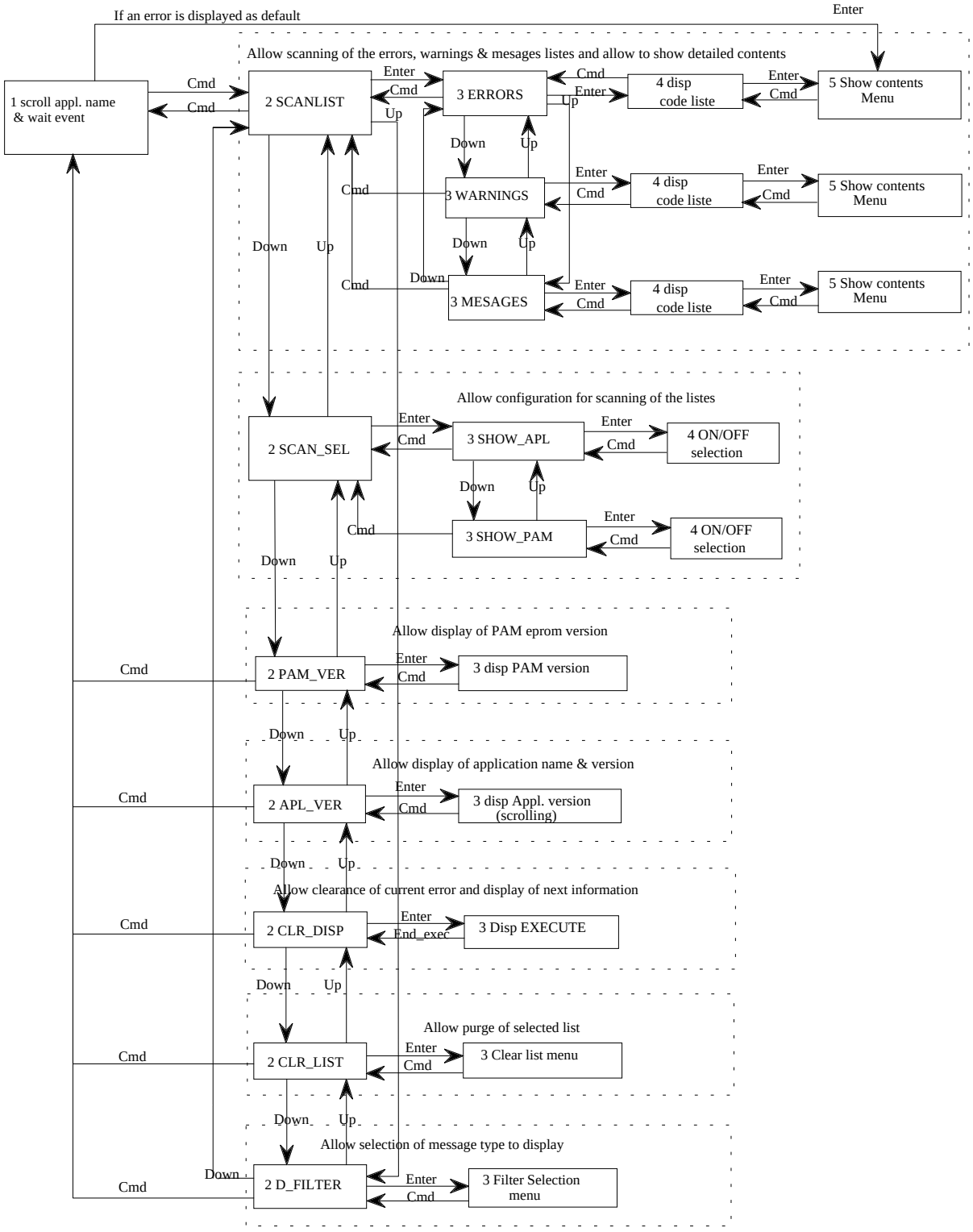
## 8.2.4. DISPLAY OF MESSAGES

The display duration of a message code is limited at 2 seconds. But messages can be successively displayed and after the delay the current display is restored (current warning code or default display).

## 8.2.5. DISPLAY OF VALUES (MONITORING)

The display duration of a value is limited at 2 seconds. But values can be successively displayed and after the delay the current display is restored (current warning code or default display). If the application asks to display a new value before the previous one has reached the 2 seconds limit, the new value will be displayed immediately. That allows to obtain a monitoring of any variable.
Values and messages codes may appears successively.

# 8.3. MENUS DESCRIPTION



## 8.4. MAIN MENU FUNCTIONS

The main menu is activated by the "**Cmd**" key.

The main menu contents 7 items**:**

| | |
|---|---|
| **SCANLIST** | scanning of lists contents |
| **SCAN_SEL** | select scanning option |
| **PAM_VER** | show PAM version |
| **APL_VER** | show application version |
| **CLR_DISP** | clear displayed error |
| **CLR_LIST** | clear of lists |
| **D_FILTER** | select display filter |

### 8.4.1. SCANNING OF LISTS CONTENTS

This function allows the scanning of the three lists (errors, warnings & messages).

The prompt of this menu is **SCANLIST**, by pressing the "**Enter**" key a sub menu with three new items is reached :

**ERRORS**
**WARNINGS**
**MESSAGES**

Use "**Down**" or "**Up**" key to explore the menu items and "**Enter**" to select a list.

When the list is selected, a PAM code is displayed or "**EMPTY**" if the list is empty.

Use the "**Down**" key to see the next item of the list or the "**Up**" key to see the previous item of the list, the display flash between to displayed code to mark user action in the case of two items with the same code.
The prompt **-NEWEST-** is displayed when there is no more next item, the prompt **-OLDEST-** is displayed when there is no more previous item .
Pressing "**Enter**" to activate the "Show contents menu" for the selected code.

The displayed code is the newest one at the first scanning of the list. After that, the displayed code is the last selected code of the previous scan session.

#### 8.4.1.1. SHOW CONTENTS MENU

This menu allows to get more information's about the selected code. This menu contents four items:

**ORIGIN**
**FIELDS**
**DATE**
**TIME**

### 8.4.1.2. SHOW ORIGIN

Press "**Enter**" to get the name of the sender process.
"Down" & "Up" keys allow to read names by field of 8 character.
Press **"Cmd**" to return back to **ORIGIN** prompt.

The three forms of message origin (sender context) are as follows:

| | | |
|---|---|---|
| Background context: | PAM task name (max.11 char. ) | PAM multiprocess name (max.11 char.) |
| Real time context: | Real time process name ( max.11 char.) | |
| Application sequence: (real time) | **S**xxx**I**xx S for sequence I for index (hexadecimal value) | |

### 8.4.1.3. SHOW FIELDS

Press "**Enter**" to show the first field.
Use "**Down**" key to see next field and "**Up**" key to see previous field.
Fields are 8 characters long. The field number is displayed for a short time, before the field contents.
Fields contents is corresponding to the hexadecimal representation of a 32 bits value.
Press **"Cmd"** to return back to **FIELDS** prompt.

Remark: the text part of a message can only be displayed with PAMTERM or PAMDEBUG.

### 8.4.1.4. SHOW DATE

Press "**Enter**" to get the date of message appearance. Date format is **YY.MM.DD.**
Press "**Cmd**" to return back to **DATE** prompt.

### 8.4.1.5. SHOW TIME

Press "**Enter"** to get the time of message appearance, time format is **hh:mm:ss**.
Press "**Cmd**" to return back to **TIME** prompt.

Press "Cmd" to return back to "selected code.

## 8.4.2. SCAN SELECTION

This menu allows the configuration of scanning. It is possible to enable or disable the scanning for errors, warnings or messages sent from application or sent by PAM.

The prompt of this menu is **SCAN_SEL.** By pressing the "**Enter**" key, a two items sub menu is reached:

**SHOW_APL**
**SHOW_PAM**

Use "**Down**" or "**Up**" keys to select and "**Enter**" to activate the ON\OFF selection.

Remark: default value of SCAN_SEL is ON for SHOW_APL & SHOW_PAM. The default value is established after PAM reset.

### 8.4.2.1. ON\OFF SELECTION

The ON\OFF selection sub menu allows to read or modify the selected state.

Example: with SHOW_APL activated we get :

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| **O** | **N** | | | | | | ***** |

The ***** indicate that SHOW_APL is ON. That means application messages are shown when the lists are scanned.

To turn SHOW_APL OFF, press the "**Down**" key to choice OFF and "**Enter**" to set:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| **O** | **F** | **F** | | | | | * |

The ***** indicate that SHOW_APL now is OFF. That means application messages are virtually removed from the lists.

### 8.4.2.2. RETURN TO SCAN_SEL PROMPT

Press "**Cmd**" to live ON\OFF selection and "**Cmd**" again to return to the SCAN_SEL prompt.

## 8.4.3. PAM FIRMWARE VERSION

The prompt of this menu is **PAM_VER**, by pressing the "**Enter**" key, the PAM firmware version in eprom and PAMCOMP (PAM compiler) version used to compile the application, .is scrolling on the display under the format:

| |
|---|
| **PAM vv.v.vv COMP v.vv** |

Press "**Cmd**" to abort display and return to PAM_VER prompt.

### 8.4.4. APPLICATION VERSION

The prompt of this menu is **APL_VER**.
By pressing the **"Enter"** key, you get the application information's scrolling on the display with the format**:**

**name v.vv CREATED month DD YYYY hh:mm**

Press "**Cmd**" to abort display and return to APL_VER prompt.

### 8.4.5. CLEAR OF DISPLAY

This function is useful only if an error code is blinking on the display !
This function allows to clear the current error and allows to display messages that are emitted after the clear display operation.

The prompt of this function is **CLR_DISP**. By pressing the "**Ente**r" key, you start the clear operation. The prompt **EXECUTE** is displayed to acknowledge the execution, followed by an automatic return to CLR_DISP prompt.

### 8.4.6. CLEAR OF LISTS

This function allows to clear all items of one list (errors, warnings or messages).

The prompt of this function is **CLR_LIST**.
By pressing the "**Enter**" key, a three items sub menu is reached:

**ERRORS**
**WARNINGS**
**MESSAGES**

Use **"Down"** or **"Up"** key to explore the menu items and **"Enter"** to select a list.
When a list is selected, the prompt **CLEAR ?** appear. Press **"Cmd"** to abort, **"Enter"** to execute. During the clear list operation, the prompt **EXECUTE** is displayed to acknowledge the execution, followed by an automatic return to the selected list prompt.

### 8.4.7. DISPLAY FILTERS

This menu allows the configuration of display filters, in the aim of disabling the display of some messages type.

The prompt of this menu is **D_FILTER**.
By pressing the "**Enter**" key , a seven items sub menu is reached**:**

| | |
|---|---|
| **MONITOR** | filter for monitoring values |
| **APL_MESG** | filter for application messages |
| **PAM_MESG** | filter for PAM messages |
| **APL_WARN** | filter for application warnings |
| **PAM_WARN** | filter for PAM warnings |
| **APL_ERR** | filter for application errors |
| **ALL** | filter for all messages type, except PAM errors ! |

Use "**Down**" or "**Up**" keys to choice a filter and "**Ente**r" to activate the ON\OFF selection sub menu.

Remark: default value of D_FILTER is OFF for all filters. The default value is established after PAM reset.

**Filter ON = No display**

#### 8.4.7.1. ON\OFF SELECTION

The ON\OFF selection sub menu allows to read or modify the state of the selected filter.

Example:

With APL_MESG activated we get:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| **O** | **N** | | | | | | |

The lack of ***** indicates the APL_MESG is OFF. That means application messages are not filtered, so they are displayed.

To turn APL_MESG filter ON, press "**Enter**", the ***** will appear:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| **O** | **N** | | | | | | ***** |

The * indicates that APL_MESG filter now is ON. That means application messages are filtered **(not displayed)**

To turn again the filter OFF, press the "**Down**" key and "**Ente**r" key. You will get:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| **O** | **F** | **F** | | | | | ***** |

#### 8.4.7.2. RETURN TO D_FILTER PROMPT

Press "**Cmd**" to live ON\OFF selection. You will get the filter prompt and you may choice an other filter. Press "**Cmd**" again to return to the D_FILTER prompt.

## 8.5. FIRST EMITTED ERROR

In regular situation, the code of the first emitted error is blinking on the display. No other errors or messages are displayed.
In fatal error situation, even if the fatal error occurs after a regular error, the fatal error will be displayed.

When an error code is blinking, the "**Enter**" key let to bypass the menu and reach directly the "show contents menu" corresponding to this error code. So it is possible to get quickly information's about this error.

It is possible to leave the "show contents menu" to go at the SCANLIST - ERRORS level by pressing "**Cmd** ". The first emitted error (oldest) is then displayed and it is possible to can scan the list with the "**Down**" key.

### 8.5.1. RETURN TO THE DEFAULT DISPLAY

To return to the default display (in this case, the blinking error code), press the "**Cmd**" key three times; 1st to return to the ERROR prompt, 2nd to the SCANLIST prompt and 3rd to return to default display.

## 8.6. FATAL SYSTEM ERROR

When a fatal system error occurs in PAM, the **green LED** on the front panel of the PAM module **blink.**

The PAM display shows the fatal error code.

On the debugger error window we get:

"date and time" "fatal error code" "fatal error name"
"fatal error explanations"
"date and time" [0380F02D] real time error
rt_kernel real-time halted

This error is always displayed after the error that is the cause of the fatal system error.
To suppress the fatal system error message, it is necessary to correct the cause of the fatal error. The way is to solve the problem that causes prior errors then restart PAM. The previous errors can be scanned on the list.

If the fatal error become from the PAM application level, the led remain alight without blinking.

### 8.6.1. CAUSE OF FATAL SYSTEM ERRORS

- *Workspace Error [0380F02C]*

The size of a workspace is too small for one sequence !
Refer to the reference manual for workspace size specification and refer to appendix B of this manual for workspace size control.

- *High priority cycle overflow [0280F060]*

The time spend in the part under highest priority interrupt (for ring and pipes handling) is greater than the value of the PAM basic cycles. To solve this problem it is necessary or to increase the PERIOD parameter of some pipes blocks or to increase the PAM_BASIC_CYCLE value.

### 8.6.2. FAULT HANDLER FATAL SYSTEM ERRORS

Some Fatal system errors are detected by the PAM processor itself (arithmetic fault, operation fault, etc.). In this case, PAM fail in fatal system error but do not display the fatal system error code [0380F02D].

PAM **stop running** when in handler fatal system error !
So it is not possible to scan previous error on PAM !

**EXAMPLE:**

Division by zero in application.

The PAM display shows: **0200FFF2**
and the green led is blinking.

On the debugger error window shows:

date an time [0200FFF2] error
PC: ..... AC: ..... F2 : .....
FIP: ..... IP1: ..... IP2: .....

### 8.6.3. FAULT HANDLER ERROR CODES

| | |
|---|---|
| [0200FFF0] | Trace fault . |
| [0200FFF1] | Operation fault. |
| [0200FFF2] | Arithmetic fault. |
| [0200FFF3] | Floating point fault. |
| [0200FFF4] | Constraint fault. |
| [0200FFF5] | Protection fault. |
| [0200FFF6] | Machine fault. |
| [0200FFF7] | Structural fault. |
| [0200FFF8] | Type fault. |

# 9. ADVANCED CONCEPTS

This chapter is focused on the advanced concept of multiple system and how multiple objects are handled.

## 9.1. KIND OF SYSTEMS

From a PAM point of view, there are three basic kind of systems.

### 9.1.1. DEFINITIONS

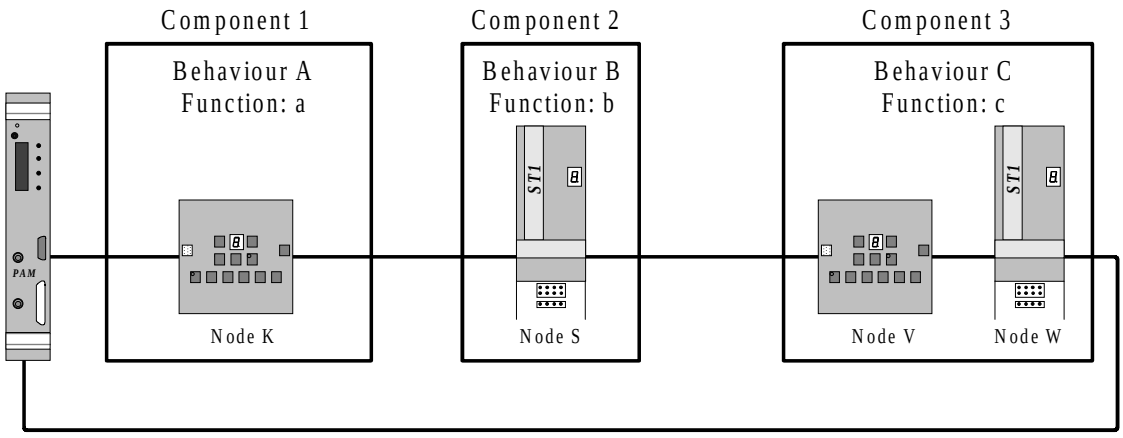
The **behaviour** of a component is the description of what a component do and how he do it. This description is made using the AGL language into a PAM application.

The **function** of a component is the set of operations made by this component in the system.

**Example**
Imagine a system build up with two drilling machines (two components). The drilling machines have drills of different diameter. This two components have the same behaviour (how to do a hole), but different functions (holes of different diameter).

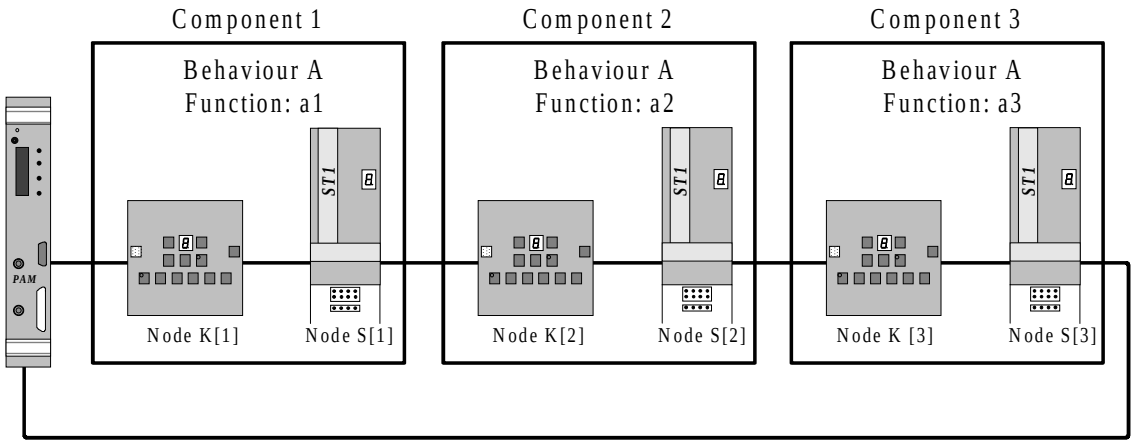
### 9.1.2. SINGLE SYSTEM

Each component of the system has its own behaviour, different of the other system's components behaviour. That means that all ring nodes of all components must be present and operational to have a working system.
The figure below shows a "single" system.



### 9.1.3. MULTIPLE SYSTEM WITH STATIC CONFIGURATION

Several components of the system have the same behaviour, but have different functions. The static configuration needs that all ring nodes of all components must be present and operational to have a working system.
The figure below shows a "multiple system with static configuration".

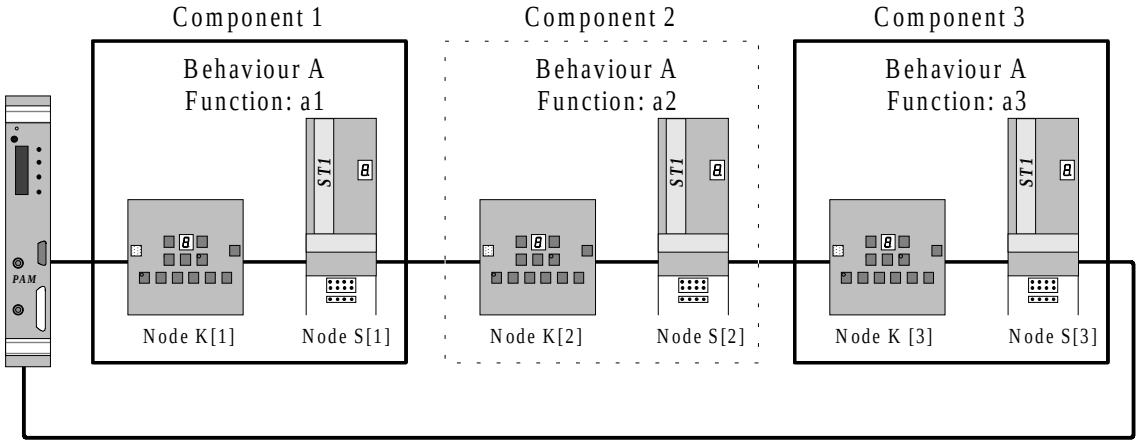



Multiple systems with static configuration use multiple objects with static size.

A multiple object with static size is an object contening a fixed number of items having the same behaviour.

## 9.1.4. MULTIPLE SYSTEM WITH DYNAMIC CONFIGURATION

Several components of the system have the same behaviour. The dynamic configuration allow that the system can work even if some of these components are not present or operational. Figure below shows a "multiple system with dynamic configuration".




Multiple systems with dynamic configuration have two main advantages:

- auto-configuration,
- node fault tolerance.

**Auto-configuration**:

The same application can drive systems build up with 1, 2, 3 or n identical components without modification or recompilation of the application.

**Node fault tolerance**:

If a component's node is faulty, by bypassing it on the ring, the whole faulty component 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 what are the nodes that build up a component. **Nodes groups** are used for this purpose.

Multiple systems with dynamic configuration use multiple objects with dynamic size.

A multiple objects with dynamic size is an object containing a number of items limited to a maximum value. Actual size is related to the real number of operational components having the same behaviour.

# 9.2. OBJECTS FOR MULTIPLE SYSTEMS

## 9.2.1. MULTIPLE OBJECTS WITH STATIC SIZE

The field **NUMBER** in object specification with a value greater than 1 indicates that the object is multiple with a static size.

All objects like NODES, AXIS, VARIABLES, TASKS and ACTIONS, must use the same value for their **NUMBER** specification.

## 9.2.2. MULTIPLE OBJECTS WITH DYNAMIC SIZE

The field **NODES_GROUP** in object specification indicates that the object is multiple with a dynamic size. The size is known at run time when all working components are identified.

All objects like NODES, AXIS, VARIABLES, TASK and ACTIONS, must refer to the same NODES_GROUP.

## 9.2.3. NODES GROUP

The declaration syntax of a nodes group is as follows:

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

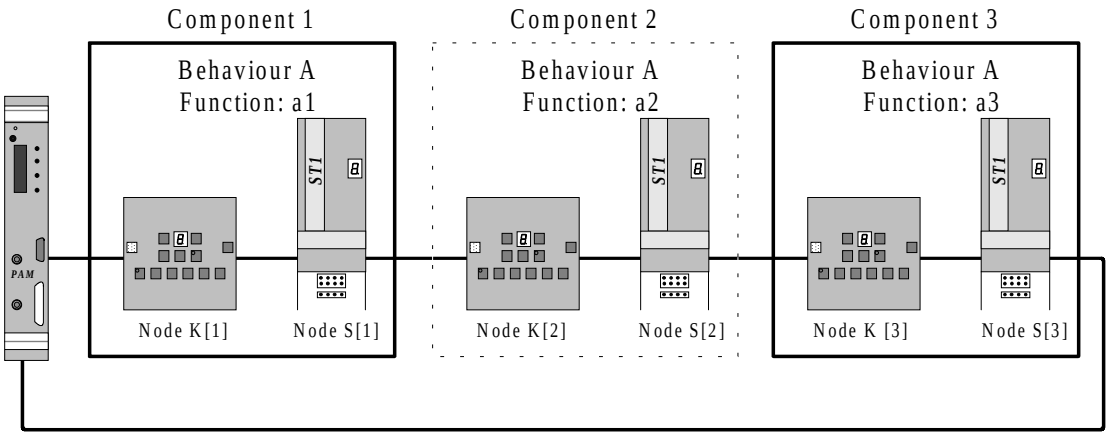
*<identifier>: the name of the nodes group.*

*<components number>: the number of identical components for all multiple objects that refer to the nodes group.*

**Purpose :**

The nodes group is the way to define what are the nodes that build up a component in the aim of using auto_configuration or node fault tolerance.

This example shows the declarations requested for the description of the system below:




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

NODE S;
    NODES_GROUP    = Component ;
    ADDRESS    = 1;   //  Address of the first S node
    TYPE       = ST1 ;
END
NODE K;
    NODES_GROUP    = Component ;
    ADDRESS    = 101;   // #65 Address of the first K node
    TYPE       = SMART_IO ;
END
```

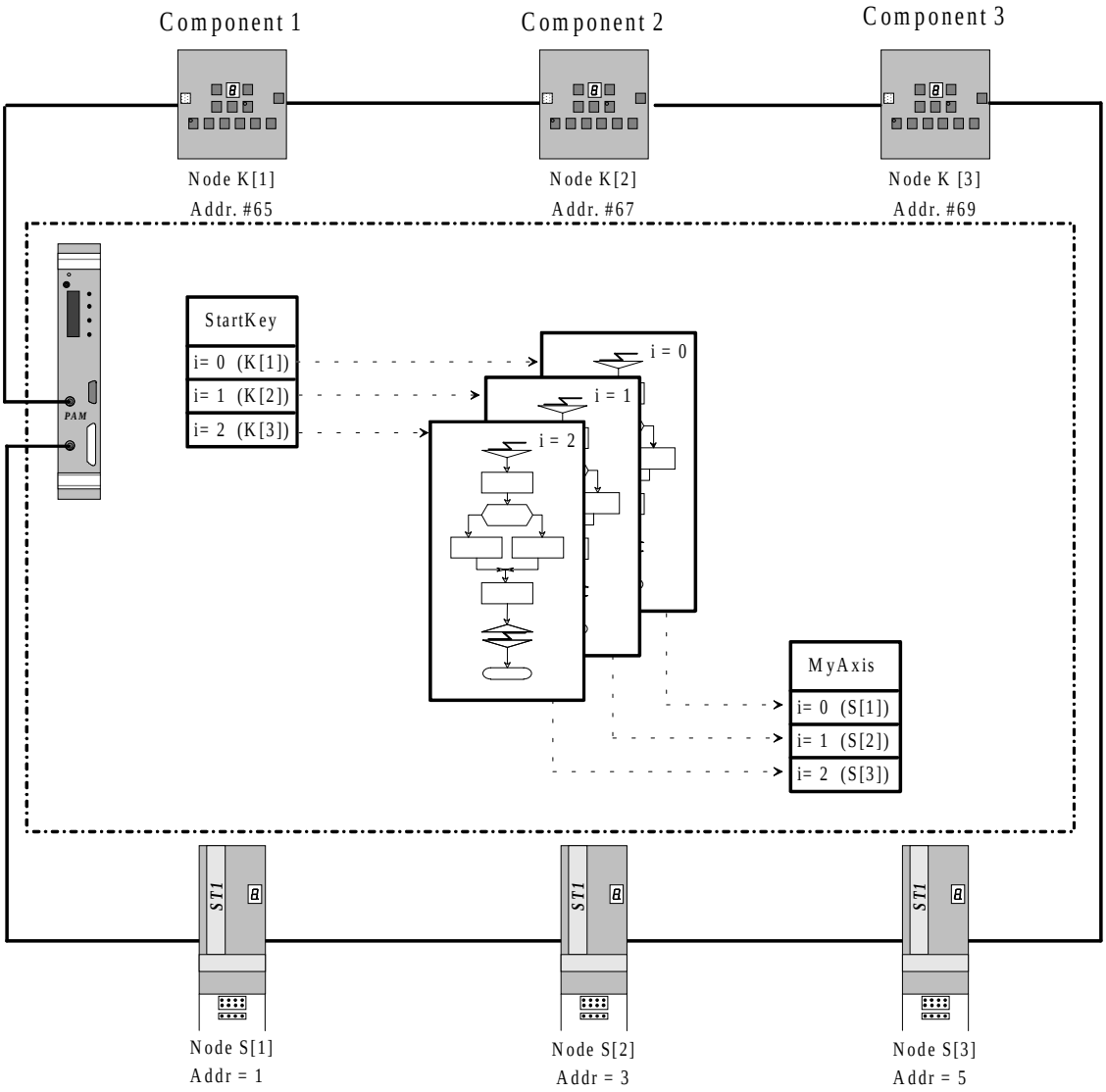
All objects in relation with these components must also be dynamically configured to match with them. So these objects must be linked to the same nodes group.

The application task that describes the behaviour of the components must also be linked to the same nodes group in the way to have a dynamic configured number of instances.

## 9.2.4. AUTO CONFIGURATION AND FAULT TOLERANCE

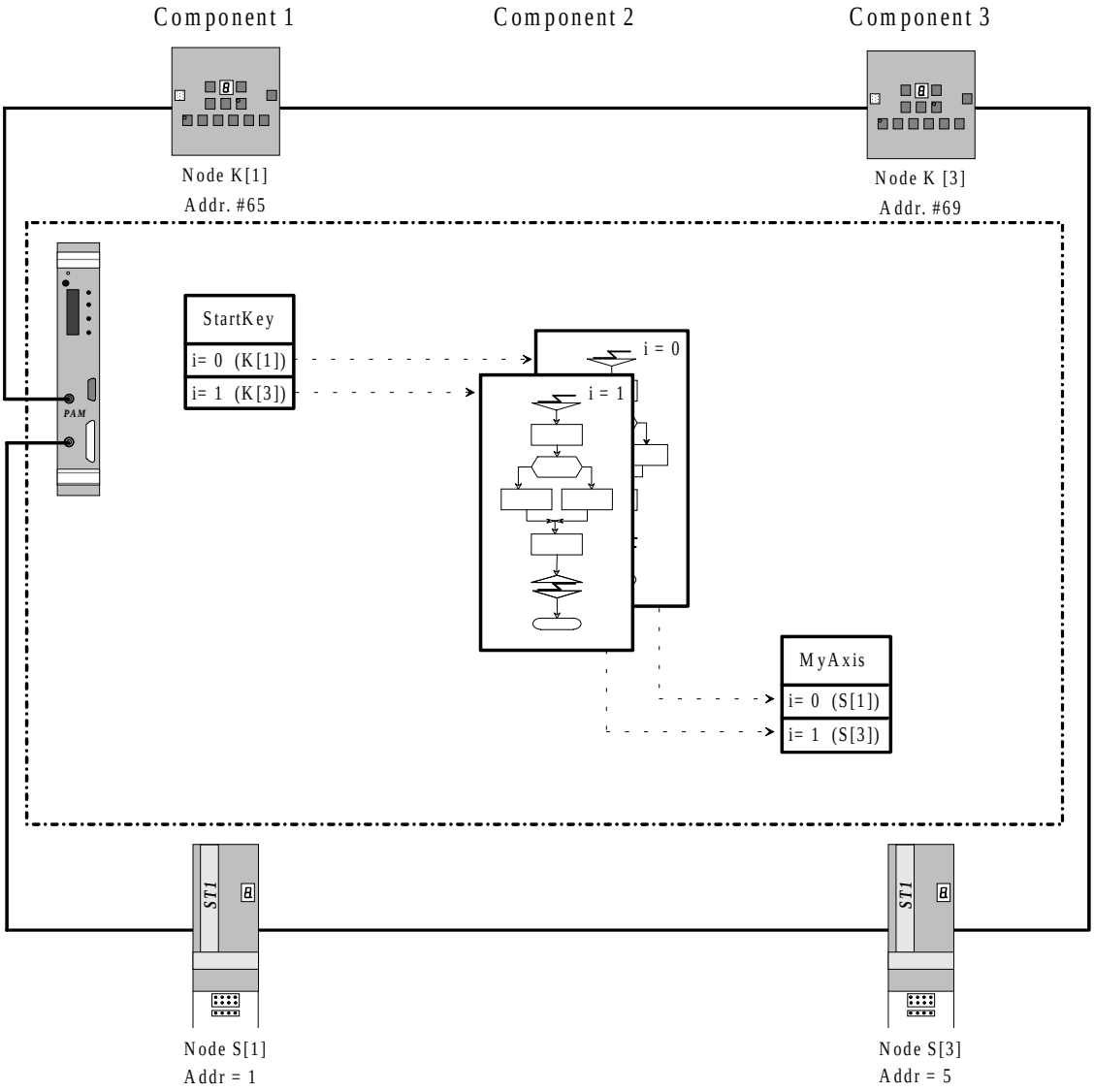
### SITUATION WITH ALL COMPONENTS WORKING

The diagram below shows a multiple system with dynamic configuration with all components working (it also shows a multiple system with a static configuration of 3 components).
When the key named "StartKey" on component K[1] is pressed, the item with logical index 0 of the peripheral variable StartKey will change and the instance with logical index 0 of the task will be executed.



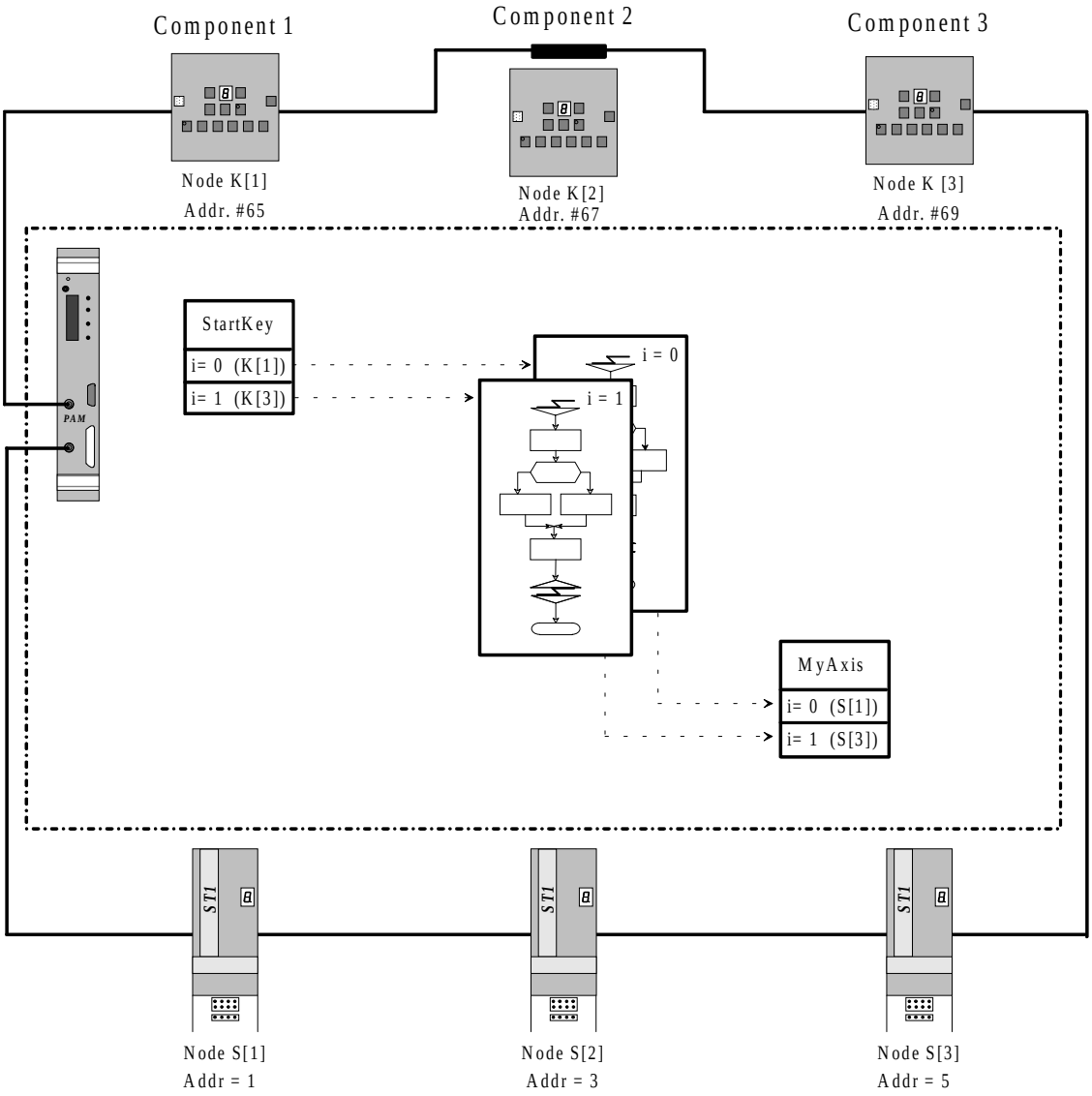
### SITUATION WITH AUTO CONFIGURATION

Component 2 is not physically present in the configuration. In this case we get a configuration with two items whose **logical index** range from 0 to 1. The **physical index** is corresponding to the physical position of the component given by the node address. With Component 2 missing, objects with logical index 0 are working with component 1 (physical index 1), objects with logical index 1 are working with component 3 (physical index 3).



### SITUATION WITH NODE FAULT TOLERANCE

In case of node failure (node K[2] not working), this node must be bypassed (optical bypass) in order to be able to work with a reduced configuration.




In order to have a match of the size of all objects referring to the nodes group, the node S[2] is automatically discarded from the configuration although it is able to work. We get the same configuration alike entire component 2 is not present.

### 9.2.5. MULTIPLE TAG

**[i]** : multiple tag for access to an item of a multiple object.

**syntax :** <*object identifier*>**[i]**

**It is only possible to access an item of a multiple object by using the multiple tag [i], not by using an index value !**

The multiple tag **[i]** gives the current value of the logical index coresponding to the component index.

**EXAMPLE:**

```
TASK ExampleMultipleTag ;
    SPECS
        NODES_GROUP = Component ;
        CYCLES = 10 ;
    END_SPECS
    EVENTS
        ON_EVENT StartKey[i] XEQ_SEQUENCE ExecMovement[i] ;
    END_EVENTS
    SEQUENCE ExecMovement[i] ;
        DrillInExecution[i] <- set ;
        DrillingHead[i] <- travel_speed(DrillingSpeed) ;
        DrillingHead[i] <- absolute_move(EndDrillPosition[i]) ;
        CONDITION DrillingHead[i] ? ready ;
        ...
```

**[all]** : multiple tag for access to all items of a multiple object.

**syntax :** <*object identifier*>**[all]**

**Example :**

```
    ...
    POWERON ;
        DrillInExecution[all] <- reset ;
    END_POWERON
END_TASK
```

### 9.2.6. MULTIPLE OPERATORS

The multiple operators are as follows:

**[all+]** the multiple OR operator.
**[all*]** the multiple AND operator.

The **[all+]** operator means "do the boolean sum (OR operator) of all items of the multiple object or expression". The result is single.

The **[all*]** operator means "do the boolean product (AND operator) of all items of the multiple object or expression". The result is single.

**MULTIPLE OPERATORS EXAMPLES:**

```
CONDITION  (DrillingHead[i] ? ready)[all*] ;
```

```
BOOLEAN equationExample;
    EQUATION
    DrillInExecution[all+] * !emergency ;
END_BOOLEAN
```

### 9.2.7. MULTIPLE TASK EXAMPLE

To make holes in parts, lets imagine a drilling machine followed by a maximum of three boring machines working in parallel to reduce boring time.
Suppose each boring head is moved by a screw drived by a motor powered by a ST1. Assume we don't care of the spindle. Still suppose a probe to detect the part, wired to the SMART_IO, is used to start the boring operation.

It is possible to write only one multiple task to execute the n boring operations.

```
TASK BoringOperation ;
    SPECS
        NODES_GROUP = BoringHeads ;
        CYCLES = 10 ;
    END_SPECS
    EVENTS
        ON_EVENT BoringRequest[i] XEQ_SEQUENCE ExecBoringMovement[i] ;
    END_EVENTS
    SEQUENCE ExecBoringMovement[i] ;
        BoringInExecution[i] <- set ;
        BoringHead[i] <- travel_speed(BoringSpeed) ;
        BoringHead[i] <- absolute_move(EndBoringPosition[i]) ;
        CONDITION BoringHead[i] ? ready ;
        BoringHead[i] <- travel_speed(FastSpeed) ;
        BoringHead[i] <- absolute_move(0) ;
        CONDITION BoringHead[i] ? ready ;
        BoringInExecution[i] <- reset ;
    END_SEQUENCE
    POWERON ;
        BoringInExecution[all] <- reset ;
        EndBoringPosition[all] <- 4000 ;
        BoringSpeed <- 500;
        FastSpeed <- 2000;
    END_POWERON
END_TASK
```

This task will be executed for three different values of the index [i]. The value of the index is given by the current index of BoringRequest[i].(corresponding to one of the three probes with the same behaviour).

## 9.2.8. DECLARATIONS FOR MULTIPLE TASK EXAMPLE

Nodes groupe declaration:

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

Nodes declaration :

```
NODE BoringHeadAxisNode ;
    NODES_GROUP = BoringHeads ;
    ADDRESS = 1;
    TYPE = ST1 ;
END
NODE BoringHeadKeyboardNode ;
    NODES_GROUP   = BoringHeads ;
    ADDRESS    = 101;   // #65 Address of the first
    TYPE       = SMART_IO ;
END
```

The addresses of the three ST1 that must be coded onto ST1 LPO board are 1, 3, 5.
The addresses of the three SMART_IO that must be coded onto board are #65, #67, #69.

**PAM ring nodes must have successive odd addresses.**

With the choice of the user lenght unit of the 1/100 of milimeter, the axis declaration is:

```
AXIS BoringHead ;
    NODE BoringHeadAxisNode ;
    PULSE PER UNIT = 1717986.9184 ; // 2^32 / 2500 ( 25 mm per turn)
    TRAVEL SPEED   = 2000 ; // 2000 1/100 mm per second
    ACCELERATION   = 4000 ; // 4000 1/100 mm per square second
    DECELERATION   = 4000 ; // 4000 1/100 mm per square second
    POSITION RANGE = - 200 10000 ; // min max in user units
END
```

Peripheral declaration for the three probes:

```
BINARY INPUT BoringRequest ;
    NODE BoringHeadKeyboardNode ;
    ADDRESS = 101 ; // first input of first module
    ACTIVE = HIGH ;
    PERIODE = 1 ;
    DEBOUNCE = 1 ;
END
```

Declaration of the multiple internal flag variable :

```
INTERNAL FLAG_VAR BoringInExecution ;
    NODES_GROUP = BoringHeads ;
END
```

Declaration of the simple internal variables used as parameter:

```
INTERNAL FLAG_VAR BoringSpeed ;
INTERNAL FLAG_VAR FastSpeed ;
```

The EndBoringPosition initial value for each head is supposed to be manually adjustable by increments using the keyboard. So an internal variable is used.

```
INTERNAL WORD_VAR EndBoringPosition ;
    NODES_GROUP = BoringHeads ;
END
```

### 9.2.9. DUALPORT MULTIPLE VARIABLE

The access to the physical items of the dualport variables are made through the physical index of the multiple variable.

When the dualport variable is a multiple variable with dynamic configuration, PAM makes a conversion from physical index to logical index when reading from the dualport and a logical to physical conversion when writing into the dualport.

Suppose we have the following declarations:

Nodes groupe declaration:

```
NODES_GROUP Heads ;
        NUMBER = 8 ;         // Maximum components number
END
```

Nodes declaration :

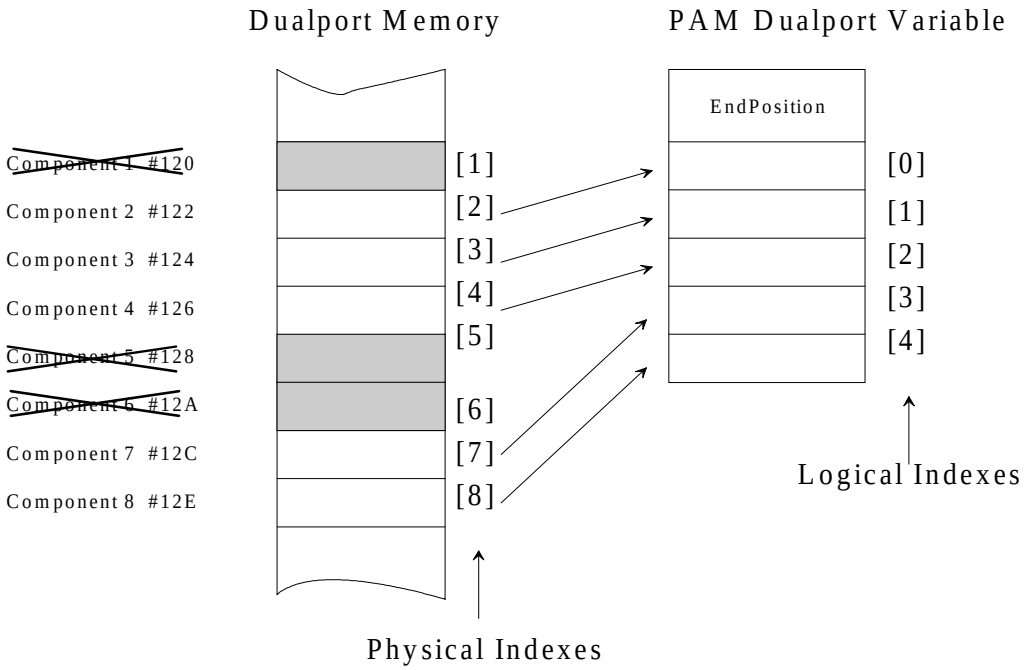
```
NODE HeadAxisNode ;
    NODES_GROUP = Heads ;
    ADDRESS = 1;
    TYPE = ST1 ;
END
```

```
NODE HeadKeyboardNode ;
    NODES_GROUP = Heads ;
    ADDRESS = 101;   // #65 Address of the first
    TYPE = SMART_IO ;
END
```

And the dualport variable declaration (Simatic version) :

```
DUALPORT_IN WORD_VAR EndPosition ;
    NODES_GROUP = Heads ;
    ADDRESS = #120
END
```

We suppose a situation of auto configuration with components 1, 5 and 6 missing, we get the following situation :




## 9.2.10. PHYSICAL INDEX VARIABLE

The PAM application language provide a special local variable **"i".**

This read only variable is usable in actions, boolean equations and sequences.

The **"i"** variable returns the physical index **(1..n)**

If the context (actions, boolean equations or sequence) is single, the value of "i" is 1 (not 0).

**Purpose :**
The i variable is useful when it is requested to manage a value (time, position or speed) in relation with the physical index of a component.

Example :

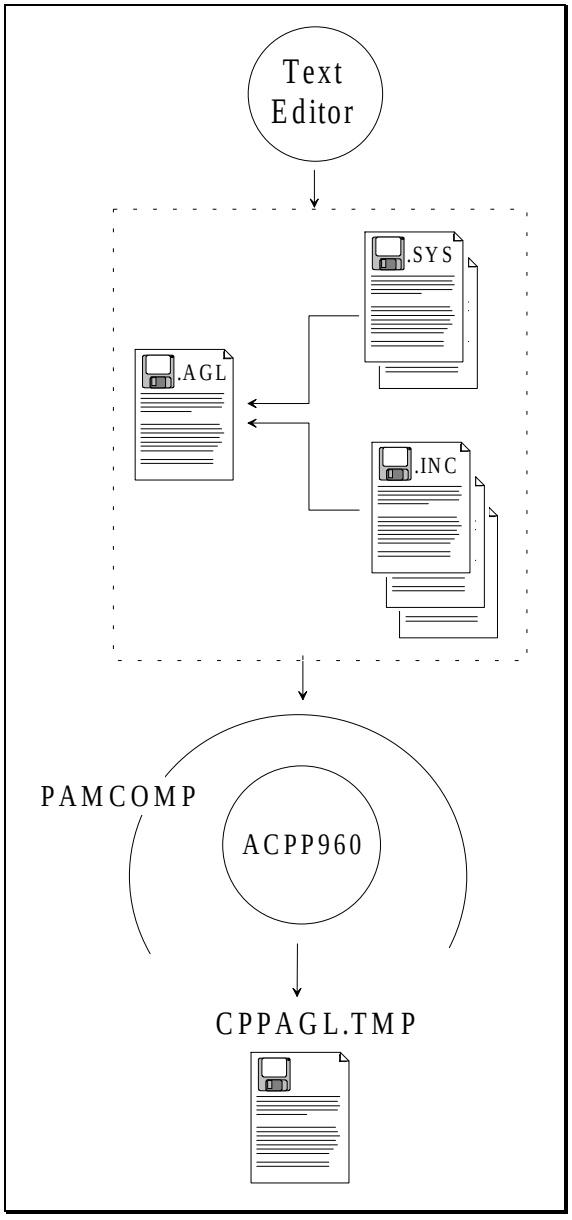
```
...
WAIT_TIME ( i * 20) ;// i proportional delay
My7segDisplay <- display ( i,1); // display value of i
...
```

# APPENDIX A PREPROCESSOR

## SOURCE FILES

It is a good practice to split the application into several source files:

1. The main file.
2. The include files.
3. The system files.



The main source file is a text file that contain application language statements. By convention, main source file is given the extension .AGL .

The include source files are text files that contain application language statements. By convention, include files are given the extension .INC .

The system files are text files that contain constant definitions. These files are located in the PAM Tools directories. By convention, system files are given the extension .SYS .

All the files above can contain preprocessor directives.

## RESULT FILE

Preprocessing result files is a text file that contains .AGL, .INC and .SYS file contents. Constants and macro replacements are done. The name of the preprocessiong result file is CPPAGL.TMP.

## COMPILER SWITCHES

- N : no preprocessing.
*Do not run the preprocessor.*

-T : leave the preprocessor result.
*Do not delete the CPPAGL.TMP file when the compilation is completed. The CPPAGL.TMP file is located in the directory for temporary files (PAMTEMP environment variable).*

## PREPROCESSOR

ACPP960 is a macro preprocessor which operates in the same way as the "C" preprocessor used by most "C" compilers. ACPP960 is called by the PAMCOMP compiler.

# PREPROCESSOR DIRECTIVES

## File Inclusion

The `#include` directive is used to incorporate other files into the current file. There are two forms of the directive:

```
#include <filename>

#include "filename"
```

If `filename` is surrounded by double quotes, then the preprocessor looks for the file in the same directory as the file which contains the `#include` directive.

If `filename` is surrounded by angle brackets, then the preprocessor looks for the file in the directories of the PAMINC environment variable, then looks for Socapel's system files.

## One-line String Macro

To define an one-line macro, use:

```
#define name text
```

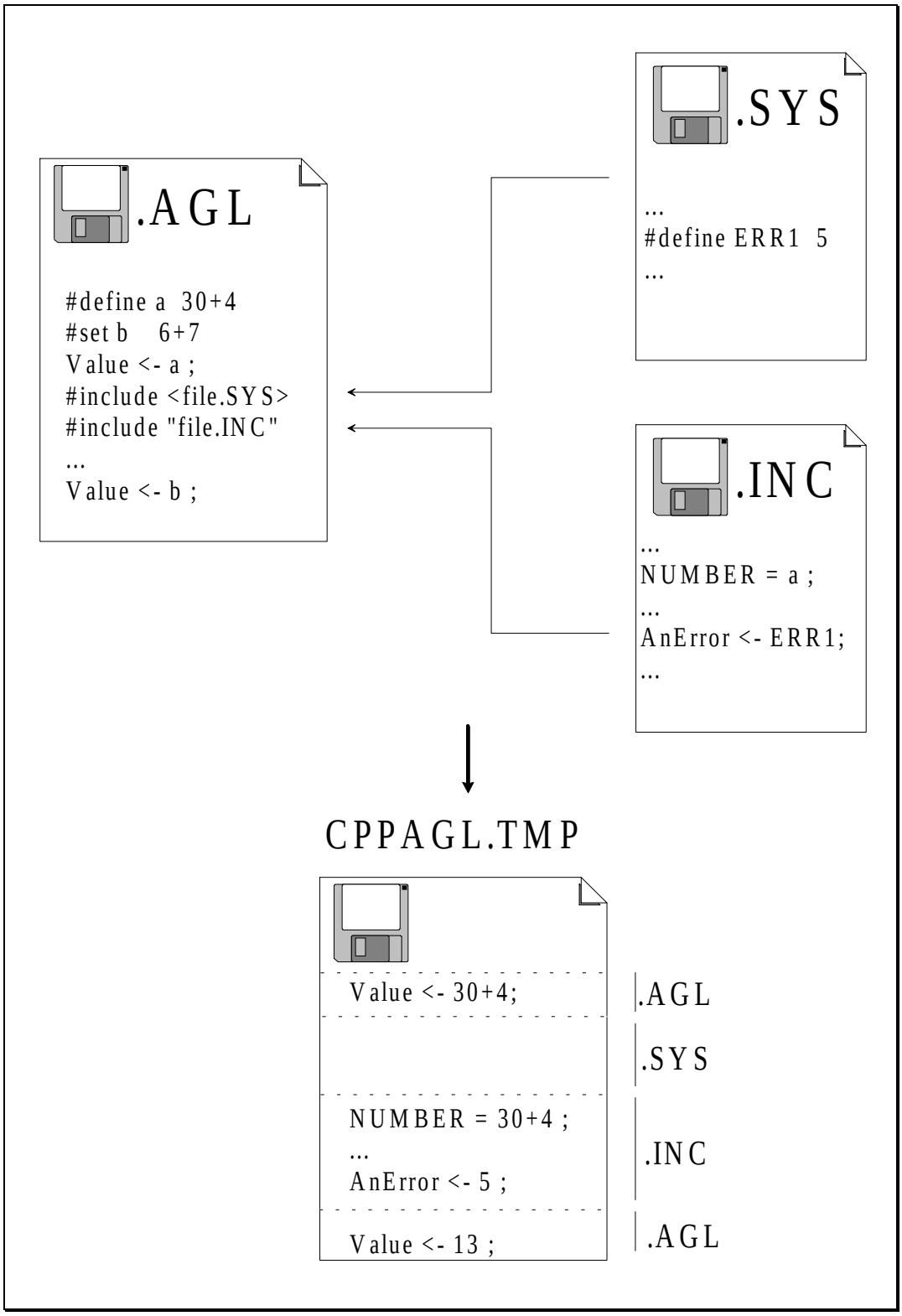
Any time `name` is found in the input after being defined, it is replaced by `text`, which means everything after the `name` up to the end of the line.

## Numeric Macro

The `#define` construct is called string macro because it produces an arbitrary string as result. The numeric macro expands into a string which is guaranteed to have the form of a numeric constant:

```
#set name <expr>
```

The argument <expr> is an expression build up with constants, arithmetic operators, logical operators and bitwise logical operators.




Preprocessor directives example

# APPENDIX B WORKSPACE SIZE CONTROL

The workspace of a sequence is the amount of PAM internal memory which is allocated for its execution. Each time PAM starts the execution of a sequence it reserves a workspace for it. The default workspace size could limit the execution of a large number of simple sequences by going over the physical memory limit of PAM.

Refer to the reference manual for workspace size specification

To control the workspace size used by the sequences, they can be listed with the PAM debugger.

To get the workspace size information of a sequence, it is necessary to write the sequence ID (identifier number) into the PAM memory at the address 30f1c8.

> Use the function **modify memory**, type : **<ctrl> O** and type the memory address and then <enter>.

To see workspace size used by Sequences with ID 0 to 3, type 1 and <enter>. Then to see the list press <Esc>.
To see workspace size used by Sequence ID n to n+3, type n+1 and <enter>. Then to see the list press <Esc>.

The list of workspace size is displayed in the PAM MESSAGES windows and looks like as follows:

```
date and time   [0380M000] real time message
rt_kernel       Snnn xxx zzz Snnn xxx zzz Snnn xxx zzz Snnn xxx zzz
```

Snnn is the Sequence number nnn (ID).
xxx is the workspace size (decimal) specified (or default value) for this sequence.
zzz is the workspace size (decimal) maximum used by this sequence.

# INDEX

**T**



**V**



**W**