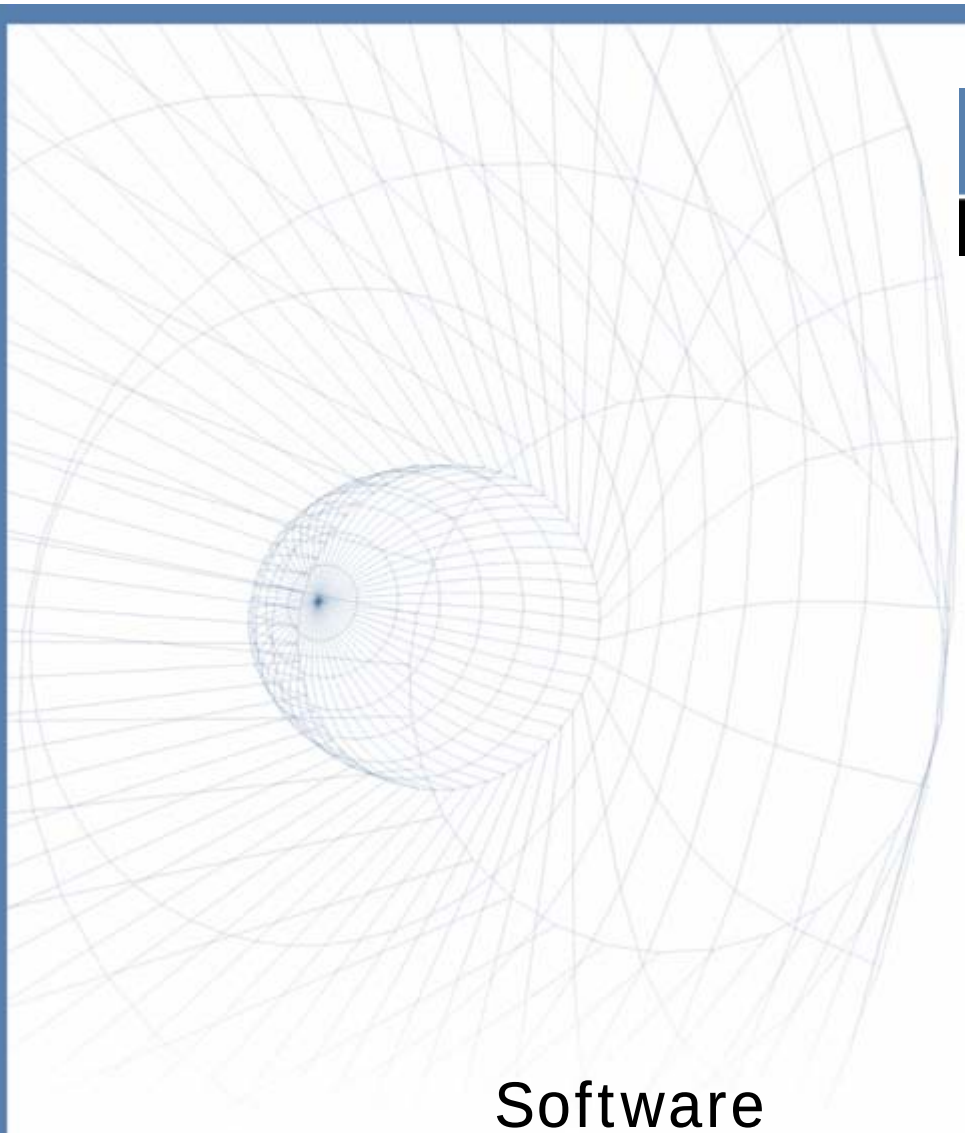


AVEVA PLANT

Software Customisation Guide

www.aveva.com

Disclaimer

Copyright

First published September 2007

AVEVA Solutions Ltd, High Cross, Madingley Road, Cambridge, CB3 0HB, United Kingdom

Trademarks

Software Customisation Guide

Contents **Page**

Customisation Guide

1 Introduction

This manual describes how to use **PML**, the AVEVA Programmable Macro Language. You should use it together with the *Software Customisation Reference Manual.*

You do not have to be a professional programmer to start to learn PML, although you may find this manual difficult to follow unless you have some understanding of programming concepts such as if statements and do loops. If you have no programming experience, you should consider attending a PML Training Course. For details, contact your local support office, whose address is given on the copyright page of this manual.

The current version of PML, sometimes referred to as **PML2**, may simply be seen as an extension to the original **PML1** facilities. However, the more powerful techniques that are available mean that many programming tasks are carried out in different ways.

There are some tasks which are carried out more efficiently using PML 1 facilities. These are described in this manual where necessary, you should also refer to the Database Management Reference Manual.

1.1 Customising a Graphical User Interface

Most AVEVA products make use of a **Graphical User Interface** (GUI) to drive the software. The interfaces provided with your AVEVA software are designed to apply to a wide range of situations and business needs. However, as you become more experienced with AVEVA products you may wish to design an interface that is more closely related to your requirements.

PML 2 has been specifically designed for writing and customising the **Forms and Menus** for AVEVA products. Almost all the facilities available in PML 1 and the older Forms and Menus facilities continue to function as before even if they are not documented here.

Before you begin customising a GUI, you must have a good working knowledge of the command syntax for the AVEVA product you are working with. The commands are described in detail in the reference manuals for the products.

1.2 Serious Warning About Software Customisation

The ability to customise individual Applications to suit your own specific needs gives you great flexibility in the ways in which you use your system.

But it also introduces the risk that your modified macros may ***not*** be compatible with future versions of the software, since they are no longer under AVEVA's control.

Your own Applications may diverge from future standard versions and may not take advantage of product enhancements incorporated into the standard product.

To minimise this risk, it is most important that your in-house customisation policies constrain any changes which you make to the Applications so that they retain maximum compatibility with the standard product at all times.

Remember that AVEVA Ltd can give you full technical support ***only*** for products over which it has control. We cannot guarantee to solve problems caused by software which you have written yourself.

1.3 How to Use this Manual

Chapters 2 to 12 describe the facilities available in PML2, with reference to the differences between PML1 and PML2.

Chapters 13 onwards describe how to use PML2 to customise a GUI.

Appendix A contains detailed information for AVEVA developers on how to use core-code managed objects.

Appendix B contains details of how to manipulate the interactive 2D and 3D views.

1.3.1 Hints on the Trying the Examples

This manual is not really designed as a self-teach tutorial, although users who are already familiar with PML 1 programming may be able to use it as such.

Because PML 2 is specifically designed to be used to create Forms and Menus, the best way of experimenting is to start up an AVEVA product and display the command line.

You can then create a new form definition file, or edit an existing one, display the form, and use it to call the PML Functions which you have defined in separate files.

The source files for some of the larger examples are available from the AVEVA web site, so you can download them rather than having to type them in.

You can find them here under PDMS/Macro Library.

http://support.aveva.com/support/United_Kingdom/index.htm

1.4 Minimising Problems for Future Upgrades

When future versions are released, the product documentation will identify which macro files and which parts of the directory hierarchy have been modified. You must then decide on one of three courses of action for dealing with any customised macros which you may have been using:

- Run your existing macros, thereby ignoring any enhanced features which may have been incorporated into the standard versions.
- Revert to using standard macros instead of your customised versions.
- Update your customised macros to take advantage of any enhanced features now available.

In order to minimise any problems resulting from updating your customised macros, you should adopt the following key principles as part of your customisation policy:

- The directory which holds the macros associated with your Applications is divided into two main areas: a **standard product** area and a **user** area. You should ***make changes only in the User area*** using only ***copies*** of the standard Application macros.

Warning: You should never modify the original files, so that you can always revert to using these if things go wrong.

- The environment variable **PMLLIB** is a search path which must be set to the directory or list of directories under which user-defined PML Functions, objects and forms are stored. The directory specified can contain sub-directories which will automatically be scanned by PML.
- When writing your own PML Application **macros** or **functions**, ***follow the styles and conventions used in the AVEVA standard*** **AppWare**.
- Ensure that the ***names*** of any new forms which you create cannot conflict with the names of any standard forms which may be used in the AVEVA Applications. You should prefix the names of your forms and global variables with a suitable code to distinguish them from AVEVA names. For example, a company XYZ could choose to start all form and global variables names with the characters XYZ.

Current PML2 Naming Convention

The majority of the existing PML2 code has been given a three or four letter prefix to the full name.

Application	Prefix	Comments
Accommodation	ACC	
Administration	ADM	
Access platform, Stairs & Ladders	ASL	
Area Based ADP	ABA	Add-in application
Assembly	ASSY	
Associations	ASSOC	None apparent
Common	COMM	Mixed
Design	DES	Mixed
Draft	DRA	Plus ADP, DGN, DXF
Final Designer	DRA	
Global	GLB	
Hull Design	HULL	
Hull Drafting	HDRA	None
Integrator	INT	
Isodraft	ISO	
Isometric ADP	ISO	Add-in application
Diagrams	DIAG	No appware except main form
Monitor	MON	
Paragon	CAT	
Review Interface	REVI	None - Single file

Application	Prefix	Comments
Router	RTR	
Spooler	SPL	
Training	TRA	

Icon naming

Icons are named without regard to any rules, except on odd occasions.

Folder	Prefix	Comments
Actions		Various
Applications		Various
Access platform, Stairs & Ladders	ASL	
Assembly ADP	AD	
Constructs		Various
Design	DES	Various
Draft	DRAFT	
Profile	PRFL	Various
Schematic Model Viewer	SMV	Various
Shapes		No existing icons
Splash		Various
Styles	STY	Plus FSTY

1.5 Note for Users Familiar with OO Concepts

PML2 is almost an object-oriented language. Its main deficiency is that it lacks inheritance. However, it does provide for classes of built-in, system-defined and user-defined object types.

Objects have **members** (their own variables) and **methods** (their own **functions**). All PML Variables are an instance of a built-in, system-defined or user-defined object type.

Operators and methods are **polymorphic** — what they do (their behaviour) depends on the type of the variable. **Overloading** of functions and operators is supported for all variable types.

There is no concept of **private** members or methods, everything is **public**. There are only two levels of scope for variables: Global and **Local**. **Arguments** to PML Functions are passed-by-reference with Read/Write access so any argument can potentially be used as an output argument.

Warning: Changing the value of an argument inside a PML Functions or method will change the value in the calling function or method.

2 Variables, Objects, Functions and Methods

2.1 PML Variables

Variables are used to store values. Variables have names. The value that is stored can be changed but the name is fixed. You choose the names and decide what is stored by each variable.

The variables used in PML 2 are objects:

- Every object (variable) has a unique name.
- An object has a set of functions associated with it, which can be called to manipulate this data. These functions are called methods.
- Some methods change the data held by the object, some return a result derived from the data, and some methods do both.
- The data that an object can hold and the functions that are available are fixed by the object type (sometimes called the object's **class**).
- Before you can create an object, the object type must have been defined. The object type definition specifies what the members and methods are for that type of object.

2.2 Object Types

Every PML2 variable has an object type which is set when the variable is created and remains fixed as long as the variable exists.

Attempting to store a value of the wrong type in an existing variable will cause an error. The object type must exist before you can create a variable of that type.

PML 2 is supplied with **built-in** object types, **system-defined** object types, and you can also define your own **user-defined** object types.

The built-in object types include the following:

Object	Description
STRING	This holds any text, including newline and multi-byte characters such as Kanji. In macros written with PML1 where variables are created with the **VAR** command, all variables are of type **STRING**, even when they are storing numbers
REAL	This is used for all numeric values including do loop counters. There is no separate integer type

Object	Description
BOOLEAN	This is used for the result of logical expressions and holds the value **TRUE** or **FALSE** **Note:** These are ***not*** the same as the **STRING** values 'TRUE' and 'FALSE'.
ARRAY	This holds many values of any type and is described in Chapter 6.

There are also system-defined variable types, such as **POSITION** and **ORIENTATION** in PDMS.

User-defined variable types are explained in *User-defined Object Types*.

2.3 Members and Attributes

An object can contain one or more items of data referred to as its *members* or **attributes**. Each member has a name.

- **STRING**, **REAL** and **BOOLEAN** variables just have a value - no members.
- **ARRAY** variables have numbered elements, not members.

2.4 User-defined Object Types

You may find that the object types supplied with PML 2 are enough.

However, you can define new object types if you need to. In any case, the following example may help you understand objects, their members and how they are used.

The following commands define an object type **FACTORY** with two **REAL** members and a **STRING** member:

```
define object FACTORY
 member .Name is STRING
 member .Workers is REAL
 member .Output is REAL
endobject
```

Here is another example where the object type **FACTORY** is itself used for one of the data members of another object type:

```
define object PRODUCT
 member .Code is STRING
 member .Total is REAL
 member .Site is FACTORY
endobject
```

It is a good idea to use upper case for **object-type** names, and mixed upper and lower case for variable names, the objects themselves. For example, a type might be **WORKERS**, and a variable name might be **NumberOfWorkers.**

2.4.1 Storing Object Type Definitions

Object type definitions should normally be stored in a file with a lowercase name matching the name of the object type and a `.pmlobj` suffix in a directory specified in the **PMLLIB search-path**, which can include the current directory. **PML** will load a definition automatically when it is needed.

2.5 Creating Variables (Objects)

Note: There is no limit to the number of objects you can create of each type.

2.5.1 Local and Global Variable Names

PML variables are of two kinds: **global** and **local**.

Global variables last for a whole session (or until you delete them). A *local* variable can be used only from within one PML Function or macro.

These two kinds of variable are distinguished by their names. Names beginning '`!!`' are *global*; names beginning '`!`' are *local*:

```
!SurfaceArea
```

```
!!Area
```

PML Variable names may be any combination of letters and digits, starting with a letter, up to a maximum of 16 characters (plus the '`!!`' or '`!`'). Names are allowed to contain a **dot** (.) but this is now strongly discouraged as a dot has a special meaning in PML2 as a separator between an object and its methods (see *Using the Methods of an Object*), and between the components of form and gadget names.

Rather than using a dot as part of a name we recommend that you use a mixture of upper and lower case to make variable names more meaningful, for example:

```
!!StartInUpperCase
```

2.5.2 Notes on Naming Conventions

Use a naming convention to avoid 'polluting the **namespace**' of global variables. All global names beginning with the letters **CD** are reserved for AVEVA system use.

Note that the most recent AVEVA **Applicationware** (**AppWare**) does not use a prefix. When creating new PML Macros you are responsible for avoiding name-clashes, for example, by using your own prefix.

Here are some guidelines on naming variables:

- Use each variable for one purpose only and give it a meaningful name.
- Limit your use of global variables.
- Where possible use PML Functions rather than macros (macros are described in *Macros*) so as to make use of function return values and output arguments to return results.

- Remember that within the methods of PML Objects and Forms you generally need only local variables. Any values which are to be exported should be made available as members or as method values or returned arguments.
- Only use PML Functions (necessarily global) for tasks which are genuinely globally accessible, i.e. independent of a specific object or form, or which need to be shared by several objects or forms.

Current Global Variables

Application	Prefix	Comments
Area Based ADP	!!ABA	Add-in application
Administration	!!ADM	Plus !!CDA
Application Switching	!!APP	
Batch Handling	!!BAT	
CADCentre	!!CADC	
Draft	!!CDR	
Draft Icon Location	!!CD2D	Icon pathnames
Defaults	!!DFLTS	
General	!!CDC	Plus !!CDD, E, F, G, H, I, L, M, N, P, S, U, V, W
Specialised	!!CE	Plus !!ERROR, !!FMSYS
Various Others		

2.5.3 Creating a Variable with a Built-in Type

You can create variables of any of the built-in types (**REAL**, **STRING**, **BOOLEAN** or **ARRAY**), as in the following example, which creates a **REAL** variable and sets it value to 99:

```
!MyNumber = 99
```

Because the variable is set to a real, PML knows that its type is real. Similarly, you can create a **STRING** variable and set it like this:

```
!MyString = 'Hello World'
```

The PML Variable **!MyString** and the constant **'Hello World'** are both objects of type **STRING**. A **STRING** is a very simple object and the only data it contains is its text value. In this simple expression we are creating a new object, a variable of **STRING** type, by making a copy of the original object, the **STRING** constant `'Hello World'`.

Other examples:

!!Answer = 42	$* creates !!Answer as a GLOBAL REAL variable
\!Name = 'Fred'	$* creates !Name as a LOCAL STRING variable
!Grid = TRUE	$* creates !Grid as a LOCAL BOOLEAN variable

A variable may also be given a type without giving it an initial value, in which case it will have the value **UNSET**:

```
!!Answer = REAL()
!Name = STRING()
!Grid = BOOLEAN()
!Lengths = ARRAY()
```

For more about unset variables, see *UNSET Values and UNDEFINED Variables*.

2.5.4 Creating Other Types of Variable

You create variables of types that are system-defined or user-defined, using the **OBJECT** keyword. For example, to create a variable of type **FACTORY**:

```
!NewPlant = object FACTORY()
```

2.5.5 Using the Member Values of an Object

The way to set individual members of an object is to use the dot notation as follows:

```
!NewPlant = object FACTORY()
!NewPlant.Name = ProcessA
!NewPlant.Workers = 451
!NewPlant.Output = 2001
```

The dot notation is used in a similar way to access the value of a member of an object:

```
!People = !NewPlant.Workers
```

sets the variable **!People** to 451.

2.6 PML Functions and Methods

Functions and Methods may optionally have **arguments** that can be used to return values.

Arguments have a data type which is specified in the function or method definition and they are checked when the Function or Method is called. An argument may be one of the built-in types **REAL**, **STRING** or **BOOLEAN**; an **ARRAY**; a **built-in object** or a **user-defined object** or specified as **ANY**.

Functions and methods can optionally return values as their results. Functions that do not return values are known as **PML Procedures** (see *PML Procedures*).

Here is a definition of a PML Function that has two **REAL** arguments. It also has a **REAL** return value, as shown by the final `is REAL`:

```
define function !!Area( !Length is REAL, !Width is REAL )
is REAL
```

Inside a function, the arguments are referenced just as if they were local PML Variables. The **RETURN** keyword is used to specify the variable that stores the return value:

```
define function !!Area( !Length is REAL, !Width is REAL ) is REAL
 !Area = !Length * !Width
 return !Area
endfunction
```

When a PML Function is called, all the PML Variables passed as arguments must already exist.

In addition, arguments used for input must have a value, which could be a constant, when the function is called:

```
define function !!LengthOfName(!Name is STRING) is REAL
 !TidyName = !Name.trim()
 return !TidyName.Length()
endfunction
```

The function is called to set the value of a variable **!Length** as follows:

```
!Length = !!LengthOfName( '   FRED   ')
```

Here `' FRED '` is a string constant passed as an argument. We could rewrite this function so that it returns its results by changing its arguments. The output argument, **!Length**, will be set to the value required when the function is called:

```
define function !!LengthAndTrim(!Name is STRING, !Length is REAL)
 !Name = !Name.Trim()
 !Length = !Name.Length()
endfunction
```

Arguments used for output must exist prior to the call and one that is also used as an input argument must also have a value:

```
!Name = ' FRED '
!Length = REAL()
```

The function is called to set the value of a variable **!Length** as follows:

```
!!LengthAndTrim(' FRED ', !Length)
```

When an argument value is changed within a PML Function its value outside the function is also changed.

The following call is incorrect, as the function cannot modify a constant:

```
!!LengthAndTrim('  FRED  ' ,4 )      $* WRONG
```

A PML Function returning a value may be used wherever an expression or PML Variable can be used, for example, this call to the `!Area` function defined above:

```
!PartLength = 7
!PartWidth = 6
```

```
!SurfaceArea = !!Area(!PartLength, !PartWidth)
```

Note: You cannot switch to a different PDMS module if a PML Function is running. Use **EXIT** to exit from the function if it has failed to complete.

2.6.1 Storing and Loading PML Functions

When a PML Function is called it is loaded automatically from its source file in a directory located via the environment variable `PMLLIB`. The name of the external file must be lowercase and must have the `.pmlfnc` suffix. The source of a PML Function invoked as `!!AR`**EA** or `!!Area` or `!!area` all correspond to the file named `area.pmlfnc`.

Note: The `!!` signifies that the function is user-defined and that it is global - but `!!` does not form part of the external filename. All user-defined functions are global and only one may be defined per file.

Note: The `define function` must be the first line in the file and that its name and the file name must correspond.

2.6.2 PML Procedures

A PML Procedure is a PML Function that does not return a result.

A function is defined as a procedure by omitting the data type at the end of the `define function` statement:

```
define function !!Area( !Length is REAL, !Width is REAL,
 !Result is REAL)
!Result = !Length * !Width
endfunction
```

Here we are using an output argument, **!Result**, to return the result rather than using a function `return` value.

The arguments to the **!!Area** procedure can be set as follows, and then the procedure invoked with the `call` command.

```
!SurfaceArea = REAL()
!Partlength = 7
!PartWidth = 6
call !!Area(!PartLength, !PartWidth, !SurfaceArea)
```

There will be an error if you attempt to assign the result of a PML Procedure because there is no return value to assign. So, for example you can say:

```
call !!Area(!PartLength, !PartWidth, !SurfaceArea)
!Answer = !SurfaceArea
```

But you cannot say:

```
!Answer =  !!Area(!PartLength, !PartWidth,    $* WRONG
 !SurfaceArea)
```

The ( ) parentheses after the name of a procedure or function must always be present — even for procedures that do not need arguments:

```
define function !!Initialise()
 !TotalWeight = 0
 !!MaxWeight = 0
endfunction
call !!Initialise()
```

Although the `call` keyword is strictly speaking optional, its use is recommended with procedures.

Note: As well as procedures, you can invoke a PML Function that has a return value using *`call`*, in which case the function result value is discarded.

2.7 Arguments of type ANY

You may specify **ANY** as the type of an argument (and even as the type of the function return value).

Note: The use of **ANY** should be the exception rather than the rule as it switches off argument type checking - an important feature of PML Functions to help ensure correct functioning of your PML.

In the case an argument of type **ANY**, a value of any type may be passed as the argument to the function:

```
define function !!Print(!Argument is ANY)
 $P $!Argument
endfunction
```

Where an argument of type **ANY** is used, you may need to find out its actual type before you can do anything with it. The `ObjectType()` method can be used for this purpose:

```
define function !!AnyType(!Argument is ANY)
Type = !Argument.pmlobjectType()
if ( !Type EQ 'STRING' ) then
 - - do something with a STRING
elseif ( !Type EQ 'REAL' ) then
 - - do something with a REAL
elseif ( !Type EQ 'DBREF' ) then
 - - do something with a DB Reference
else
 - - do something with all other types or give an error
  endif
endfunction
```

2.8 Using the Methods of an Object

Note: A method is a function that is specific to an object.

The *Software Customisation Reference Manual* contains a table of the object types supplied as part of PML 2. For each object type, there is a list of the associated methods and members.

For each object type, the table shows:

Name	The name of the method or member. For example, a **REAL** object has a method named `Cosine`. If there are any arguments, they are indicated in the brackets () after the name. For example, the **REAL** object has a method named `BETWEEN` which takes two **REAL** arguments.
Result or Type	For members, we use the heading Type; for methods, we use the heading Result. The type of the member or the result describes what kind of object we are expecting to see as a member or result of the method. For example, the result of the method Cosine is a **REAL** value. Some methods do not return a value: these are shown as **NO RESULT**.
Status	This column is used to give other information about the method or member. For methods, this column tells you whether the method modifies the state of the object. For members, this column tells you whether the member is **Mutable** (by the user) or **Read Only**. Note that for the system-defined PDMS object types, members correspond to PDMS attributes
Purpose	This column tells you what the member or method does.

2.8.1 Example

This section explains how to use methods, using a **STRING** variable as an example. Although the **STRING** is a simple object type, it has a large number of methods which can be called.

For example, if you are interested in the length of the string value, look under the list in the *Software Customisation Reference Manual* for **STRING** objects, and you will find a method named `Length`.

This method returns a **REAL** value (the number of characters in the string), but has ***no effect*** on the variable itself.

You can extract the number of characters in the string and store it in a new variable**, !Nchars**, by calling the method as follows:

```
!Nchars = !MyString.length()        $* A method call
```

Notice the dot separator between the name of the variable and the name of the method. Also note the ( ) brackets following the name of the method. The brackets are used to enclose the arguments of the method, but they must be present even if there are no arguments.

2.9 Methods on User-Defined Object Types

When you define a new object type, you can also define methods which can be used to handle all objects created with the new type. PML Method definitions are stored in the same file as the object definition, after the `endobject` command.

Here is an example of an object which defines three methods to illustrate the main ideas. Note that within a method, !This represents the object which invoked the method and `!This.Answer` is the way to refer to member **Answer** of this object.

The Code	What it Does

`define object LIFE`
` member .Answer is REAL`
`endobject`
	Defines the object, with a single member, **Answer**.

`define method .Life()`
` !This.Answer = 42`
`endmethod`
	Defines a method with no arguments but the same name as the type of the object is called the **default constructor** method. If the default constructor method is present, PML will call it automatically to initialise the object whenever an object of that type is created.

`define method .Answer() IS REAL`
` return !This.Answer`
`endmethod`
`define method .Answer( !Value Is REAL)`
` !This.Answer = !Value`
`endmethod`
	A method may return a result in just the same way as a PML Function using the `return` command. Set a member of the object using **!This.membername**.

These methods might be used in the following way:

```
!Marvin = object LIFE()
 -- The method .Life() was called automatically
!Number = !Marvin.Answer()
```

```
 -- !Number is set to the value 42
!Marvin.Answer(40)
!Number = !Marvin.Answer()
 -- !Number now has the value 40
```

Warning: When you create a new object type, or change an existing definition, you must load the definition by giving the command:

```
pml reload object _name_
```

2.9.1 Method Overloading

Two or more methods on an object may share the same name providing they have different arguments. This is called **method overloading**. PML will invoke the method with the arguments which match the method call. It is common practice:

- To use a method with the same name as a member and one argument of the same type to set the member's value. For example:

  ```
  !Marvin.Answer(65)
  ```

- To use a method of the same name as a member returning a value of the same type but with no arguments to get the member's value. For example:

  ```
  !Number = !Marvin.Answer()
  ```

2.9.2 Constructor Methods with Arguments

When an object is created, it is possible to supply arguments that are passed by PML to a constructor method with matching arguments instead of the default constructor method:

```
!Marvin = object LIFE(40)
```

This would invoke a method:

```
define method .Life(!Value IS REAL)
```

2.9.3 Overloading with ANY

As with PML Functions, the type of a method argument may be specified as **ANY**. If method overloading is being used, PML will invoke the method with a matching set of explicitly-typed arguments in preference to calling a method with arguments of type **ANY**, irrespective of the order the methods appeared in the object definition file:

```
define method .SetValue( !Argument Is ANY)
define method .SetValue( !Argument Is REAL)
```

Then:

```
!SomeObject.SetValue(100)
```

will invoke the method with the **REAL** argument, but

```
!SomeObject.SetValue('Priceless' )
```

will invoke the method with the **ANY** argument.

2.9.4 Invoking a Method from Another Method

Within a method `!This.Methodname()` refers to another method on the same object. So our second **LIFE** constructor method, the one with an argument, could be defined as:

```
define method .Life(!Value IS REAL)
 !This.Answer(!Value)
endmethod
```

2.9.5 Developing a PML Object with Methods

Whenever you add a new method to an object, you need to tell PML to re-read the object definition, by giving the command:

```
pml reload object life
```

It is not necessary to use this command if you are simply editing an existing method (although you will have to use it if you edit a **form definition file**, and change the default constructor method, described in *Form Definition File*.)

2.10 Unset Variable Representations

Each new data type supports a `String()` method that returns a string representing the value of the variable. For example:

!X = 2.5	$* defines a variable X of type REAL with 2.5 as its numeric value
!S = !X.String()	$* will be a variable of type STRING, with the value “2.5”

UNSET variables of all built-in data types have an unset representation:

!X = REAL()!	$* yields the string ’(the empty string)
S = !X.String()	
!X = BOOLEAN()	$* yields the string ‘’ (the empty string)
!S = !X.String()	
!X = STRING()	$* yields the string ‘Unset’
!S = !X.String()	
!X = ARRAY()	$* yields the string ‘ARRAY’
!S = !X.String()	

Other variable types are system-defined variables. Most of these have adopted the unset string ‘`Unset’`. For example:

!X = DIRECTION()	$* yields the string ‘Unset’
!S = !X.String()	

User-defined data types can also provide a `String()` method. These also support an **UNSET** representation, and usually adopt the **UNSET** representation 'Unset'.

2.11 UNSET Values and UNDEFINED Variables

All data types can have the value UNSET which indicates that a variable does not have a value. A variable created without giving it an initial value in fact has the value **UNSET**:

```
!X = REAL()
```

Variables with an **UNSET** value can be passed around and assigned, but use of an **UNSET** value where a valid item of data is required will always result in a PML error.

The presence of an **UNSET** value may be tested either with functions or methods:

Functions	**Methods**
if (Unset(!X)) then	if (!X.Unset()) then
if (Set(!X)) then	if (!X.Set()) then

An **UNDEFINED** variable is one that does not exist. The existence of a variable may be tested with these functions:

```
if ( Undefined(!!Y) ) then . . .
if ( Defined(!!Y) ) then
```

There is no equivalent method call. If the variable does not exist, attempting to call a method would result in an error.

2.12 Deleting PML Variables

A variable that exists can be explicitly made **UNDEFINED** with the `Delete()`method:

```
!!Y.Delete()
```

Warning: You must not attempt to delete members of objects or forms.

2.13 Forms as Global Variables

In PML 2, forms are a type of global variable. This means that a form cannot have the same name as any other global variable or any other form.

Note that a form definition is also the definition of an object, so a form cannot have the same name as any other object type.

3 General Features of PML

3.1 Functions, Macros and Object Definitions

Functions and Macros are PML Files that contain stored sequences of commands. The PML File is invoked whenever this sequence of commands is required.

The PML File may also include control logic which alters the order in which the commands are carried out and special commands for handling errors.

PML Files are normally created using a text editor.

PML Functions and methods on objects (including forms) are the recommended way of storing command sequences because:

- There is a check that they have been called with the right type of arguments.
- Arguments can return values.
- A PML Function or method can return a result of any type.

  Most new AppWare code is written as methods on objects. PML Macro**s** are explained in *Macros* as they are the basis of the older AppWare code.

PML Macros are normally stored in a directory under the `PDMSUI` search-path.

PML Functions are automatically loaded from a directory under the `PMLLIB` search-path.

3.1.1 Comments in PML Files

Comments are additional text included in a PML File for the benefit of someone reading the PML code.

The PML processor ignores comments and so they do not affect the way the code executes.

For a simple one line comment, begin the line with - - (two dashes) or $* (dollar and asterisk).

```
- - This is a new-style PML comment
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
$* The following lines calculate the new angle
```

You can also use $* to add an **inline comment** to any line of PML:

```
!Z = !X + !Y      $* We are assuming both !X and !Y are REAL
```

A comment may extend over several lines provided it is enclosed in the **escape sequences** $(and $) .

```
$( A comment containing
```

```
more than one line $)
```

A comment of this kind can be used temporarily to **comment-out** lines of PML to prevent them from being executed, but without deleting them from the file:

```
$(
 skip if (!X EQ !Y)
$)
```

3.1.2 Leaving a PML File with the RETURN Command

At any point within a PML File a `return` command will stop further execution of the file and return to the calling PML File, if there is one:

```
if ( count EQ 0 ) then
 return
endif
```

For clarity, `return` can be used as the final line of a PML Macro. However, the use of this command is not essential and there will be no error if it is not used.

Note: This is a different use of `return` from the command used to set return values of variables.

3.2 Case Independence

Everything written in PML, including keywords such as `if`, `do` and `else` means the same thing in upper or lower case.

The exception is text enclosed between quotes or vertical bars, which is used exactly as it is typed. Throughout this document you will see examples of PML code using a mixture of upper and lower case for readability.

3.3 Abbreviations

Many commands have a minimum abbreviation which can be used in place of the full command.

For readability it is recommended that you use the full form of the command — note that it is ***not*** less efficient than using the abbreviation.

PML keywords such as `if, else` and `do` have no abbreviations.

3.4 Special Character $

The `$` character has a special meaning in PML. It is an **escape character**, which means that together with the character which follows it are treated as a special instruction to PML.

The pair of characters beginning with `$` is known as an **escape sequence**. `$P` is a commonly encountered example, which is used to output a message to the screen:

```
$P This text will be output to the screen
```

A number of other escape sequences will be described later in this manual. The important point to note here that if you need the dollar character itself as part of a command, you will need to double it:

```
$$
```

As the last character on a line, $ means that the next line is a continuation line.

For example:

```
$P This is an example of a much longer message that will be $
output to the screen
```

3.5 Text Delimiters

Text strings must be enclosed in either single quotes, or apostrophes, or vertical bars:

```
'apostrophes' or vertical bars.
```

The apostrophes and vertical bars are known as **delimiters**. Take great care to avoid unmatched delimiters as this can lead to many lines of PML code being included as part of the text string, and so lost.

3.6 Filename Extensions

The naming conventions are as follows:

Extension	Use for
.pmlfnc	PML Function definition files
.pmlobj	PML object type definition files
.pmlfrm	PML Form definition files

Note: All filename extensions ***must*** be entered in lower case.

3.7 Storing and Loading PML Files

PML Files must be stored in directories pointed to by the `PMLLIB` environment variable.

When an AVEVA product is started up, PML scans each directory in the `PMLLIB` search path and creates a file named `pml.index` in each directory.

This index file contains a list of all the PML Files in all the directories under the directory given in the `PMLLIB` search path. All the PML Files listed in the `pml.index` files are loaded automatically when the product is started up.

When you are working on PML Files, you should store them in your own PML working directory, and add the path of your PML working directory as the ***first*** entry in your `PMLLIB` search path. Then, when you add a new PML File (after you have started up an AVEVA product), you will need to tell PML to rebuild its file index by giving the command:

```
pml rehash
```

This command scans all the files under the first directory in your PMLLIB path, and updates the pml.index file.

If other users have added PML Files and updated the appropriate pml.index files, you can access the new files by giving the command:

```
pml index
```

This command re-reads all the pml.index files in your search path without rebuilding them.

When you are not running an AVEVA product, you can update the pml.index file in a given directory by giving the command:

```
pmlscan directory_name
```

This command runs a utility supplied with AVEVA products.

Note: The commands pml rehash and pml index are a type of command known as **PML directives**: they are used outside PML Files to direct PML itself to take certain actions. More information about using PML directives with form definitions is given in *PML Directives*

3.7.1 Rebuilding All PML File Indexes

Note: This process can take some time, and it is possible that if another user gives the command at the same time, the index files may not be re-built properly. Hence it is recommended that this command should only be used when necessary. System Administrators are advised to maintain firm control over the directories in the PMLLIB search path by removing write access for most users.

The following command scans all the files in all the directories in your PMLLIB path, and updates the pml.index files.

```
pml rehash all
```

3.7.2 Querying the Location of PML Files

If you are not sure where a given file is stored, you can query the path by giving the following command, using the appropriate file suffix:

```
q var !!PML.getpathname( 'filename.pmlobj' )
```

4 PML Expressions

An **expression** consists of **operators** and **operands**.

For example:

```
2 + 3
```

is an expression, where 2 and 3 are the operands, and + is the operator. The **result** of this expression is 5.

Expressions can (and normally do) contain variables and, in PDMS, expressions will often contain names of PDMS **element types**, **attributes** and **pseudo-attributes**.

Expressions can also contain arithmetic and trigonometric functions, such as the `SIN` trigonometric function.

Often, expressions will contain logical operators:

```
!height GT !width
```

Here the logical operator `GT` is being be used to test if **!height** is greater than **!width**.

Each expression has a type such as **REAL**, **STRING** or **BOOLEAN**. All the elements in an expression must be of the correct type. For example:

```
!X + 'text'        $* wrong, Wrong, WRONG!
```

is meaningless if **!X** is **REAL** and will result in an error. (But see *Concatenation* for using the concatenation operator to convert different types to **STRING**.)

4.1 Format of Expressions

The use of brackets, spaces and quotes in an expression is important. If you do not follow the rules given below you will get error messages:

Text must be enclosed in quotes, either `'` apostrophes or `|` vertical bars:

```
'This is text ' |and this is text|
```

There must be a space before and after an operator. For example:

```
!X + !Y
```

In general, you do not need spaces before or after brackets, except when a name is followed by a bracket. If there is no space, the bracket will be read as part of the name. For example:

```
(Name EQ /VESS1 )
```

4.2 Operator Precedence

Operators are evaluated in the order of the following list: the ones at the top of the list are evaluated first.

Operator	Comments
BRACKETS	Brackets can be used to control the order in which operators are evaluated, in the same way as in normal arithmetic. For example `(!A +!B) * 2.`
FUNCTIONS	
*** /**	
+ -	
NE NEQ GT LT GE GEQ LE LEQ	
NOT	
AND	
OR	

4.3 Boolean Operators

The Boolean, sometimes called Logical, operators have the following meanings:

The logical operators available are:

Operator	Comments
EQ	**TRUE** if two expressions have the same value.
NE	**TRUE** if two expressions have different values.
LT	**TRUE** if the first expression is less than the second.
GT	**TRUE** if the first expression is greater than the second.
LE OR LEQ	**TRUE** if the first expression is less than or equal to the second.
GE OR GEQ	**TRUE** if the first expression is greater then or equal to the second.
NOT	**TRUE** if the expression is **FALSE.**
AND	**TRUE** if both expressions are **TRUE**
OR	**TRUE** if either or both expressions are **TRUE.**

Note: The operators **EQ**, **NE**, **LT**, **GT**, **LE** and **GE** are sometimes referred to as *comparator* or *relational* operators; **NOT**, **AND,** and **OR** are sometimes referred to as ***Boolean***

operators. See also Section C.11, *Precisions of Comparisons* for tolerances in comparing numbers.

4.4 Concatenation

The & operator concatenates two **STRING** values (joins them end-to-end) to make a result of type **STRING**. Values of any type are automatically converted to a **STRING** first:

```
!X = 64                $* !Z is the STRING 6432
!Y = 32!
Z = !X & !Y
```

4.5 Nesting Expressions

Expressions can be nested using brackets. For example:

```
( (SIN(!angleA) * 2) / SIN(!angleB) )
```

4.6 PML 1 and PML 2 Expressions

There are now two styles of expressions available:

- PML 2 expressions are used in `if` and `do` commands and when giving a PML Variable a value using the = (assignment) operator.
- PML 1 expressions must be used in all other situations, in particular when using an expression as an argument to a command. For more information regarding PML 1 refer to Database Management Reference Manual.

4.7 PML 2 Expressions

PML2 introduces enhanced facilities for expressions in `if` and `do` commands and when giving a PML Variable a value using = assignment.

For example, the value of a PML Variable can be used by giving the variable name as the expression to the right of '=':

```
!SavedValue = !Number
```

Users familiar with PML1 should note the absence of $ preceding the variable name. These new expressions do not need to be enclosed in ( ) parentheses:

```
!NewValue = !OldValue + 1
```

PML 2 expressions:

- May be of any complexity.
- May contain calls to PML Functions and Methods.
- May include **form gadget** values (described later), and object members and methods. For example:

```
!NewValue = !!MyFunction(!OldValue) * !!Form.Gadget.Val /
!MyArray.Method()
```

5 Control Logic

There are four types of construct for implementing control logic within a PML Function or Macro. These are:

- The `if` construct for conditional execution of commands.
- The `do` command for looping and the associated `break` and `skip`.
- The `golabel` for jumping to a line with a `label`.
- The `handle` construct for dealing with errors.

5.1 IF Construct

The full form of an **if-construct** is as follows:

```
if (!Word EQ 'Peanuts' OR !Word EQ 'Crisps') then!Snacks
= !Snacks + 1
 !Meal = FALSE
elseif ( !Word EQ 'Soup') then

 !Starters = !Starters + 1
 !Meal = TRUE
elseif (!Word EQ 'Fruit' Or !Word EQ 'IceCream' ) then

 !Puddings = !Puddings + 1
 !Meal = TRUE
else

 !MainCourse = !MainCourse + 1
 !Meal = TRUE
endif
```

Each **BOOLEAN** expression, such as `(!Word EQ 'Soup')`, is examined in turn to see whether it is **TRUE** or **FALSE**. As soon as an expression that is **TRUE** is encountered, the following block of commands is executed.

Once a block of commands has been executed, everything else up to the `endif` is ignored.

The `else` command is optional. If it is included, you can be sure that exactly one command block within the if-construct will be executed.

The `elseif` commands are also optional. Once one of the `elseif` expressions has been found to be TRUE, any remaining `elseif` commands are ignored.

Thus the simplest form of the `if`-construct is:

```
if ( !Number LT 0 ) then
     !Negative = TRUE
endif
```

You may not concatenate the commands into one line, so the following ***are not allowed***:

```
if ( !Number LT 0 ) then  !Negative = TRUE endif   $* WRONG
if ( !Number LT 0 )   !Negative = TRUE             $* WRONG
```

Note: That expressions such as:

```
if ( !TrueValue OR !UnsetValue)
if ( !FalseValue AND !UnsetValue)
```

ignore !**UnsetValue** if the value is not required to determine the outcome of the expression. The same is true for PML Functions which have returned an error.

5.1.1 Nesting if-constructs

Any `if`-construct may contain further `if ... elseif … endif` constructs:

```
if (!Number LT 0) then
    !Negative = TRUE
if (!Number EQ -1 ) then
 !OnlyJustNegative = TRUE
endif
endif
```

It is particularly helpful with nested if constructs to indent the code so that the logic is clear.

5.1.2 BOOLEAN Expressions and if Statements

New expressions based on the operators such as **EQ** and **GT** give a **BOOLEAN** result that can be used directly in a PML2 `if` test:

```
if (!NewValue - 1 GT 0) then
```

The expression can be a simple variable provided it is a **BOOLEAN** type variable:

```
 !Success = !NewValue GT 0
if (!Success) then
```

The expression could be a user-defined PML Function provided it returns a `BOOLEAN` result:

```
if (!!MyFunction() ) then
```

Note: The **BOOLEAN** constants **TRUE**, **FALSE**, **YES** and **NO** and their single-letter abbreviations ***not enclosed in quotes*** return **BOOLEAN** results and so can be used directly in expressions. For example:

Code	Result Type
if (TRUE)	**BOOLEAN**
if (FALSE)	

if (T)	**BOOLEAN**
if (F)	
if (YES)	**BOOLEAN**
if (NO)	
if (Y)	**BOOLEAN**

The following do not return **BOOLEAN** values and are therefore invalid:

Code	**Result Type**
if (1)	**REAL**
if (0)	
if ('TRUE')	**STRING**
if ('FALSE')	
if ('T')	**STRING**
if ('F')	
Variable = 1	**REAL**
if ($Variable)	

For upward compatibility with PML1, **STRING** variables set to '**TRUE'**, '**FALSE'**, 'YES' or 'NO' or their single-letter abbreviations can be used in an `if`-test as long as they are evaluated with a preceding $. For example:

Code	**Result Type**
Variable = 'TRUE'	**STRING**
if ($Variable)	

There is a built-in PML Method and a function for converting a value to **BOOLEAN**:

```
!MyString = 'TRUE'
if (!MyString.Boolean() ) then . . .
```

The Boolean conversion is as follows:

Code	**Result**
REAL zero	**FALSE**
$* All other positive and negative REAL values	**TRUE**
STRING 'FALSE, 'F', 'NO' and 'N'	**FALSE**
STRING 'false, 'f', 'no' and 'n'	**FALSE**
STRING 'TRUE', 'T', 'YES' AND 'Y'	**TRUE**
STRING 'true', 't', 'yes' and 'y'	**TRUE**

5.1.3 IF TRUE Expression

IFT/RUE will return a value of the type defined by the second and third arguments.

If the initial Boolean expression is true, the result of the first expression is returned. If false, the result of the second expression is returned.

Both the second and third arguments are fully evaluated regardless of the value of the Boolean. This is because the function is evaluated using reverse polish procedure (as is the case of the expression design). This allows the IF statement to be nestable with any of the arguments to IF capable of including other IF functions.

If *logical1* expression is set to true, then value of typeX1expression is returned

If *logical1* expression is set to false, then value of typeX1 expression is returned

typeX1 and typeX2 are two arguments of the **same** type which may be:

Logical
Logical Array
Real
Real Array
ID
ID Array
Text
Position
Direction

5.2 DO Loops

A ***do-loop*** enables a series of commands to be repeated more than once. The number of times the series is repeated is controlled by a counter.

Control is passed to the line following the `enddo` command after executing the commands within the loop with the counter variable at its final value.

The full format of the `do`-loop is as follows:

```
do !x from 10 to 100 by 10
 !Total = !Total + !X
enddo
```

The `enddo` must be present at the end of the loop.

The `!X` , `from`, `to` and `by` are optional, therefore in its simplest form you may use:

```
do
 commands block
enddo
```

This will loop forever unless something in the commands block stops the loop (see `break` and `golabel` below)

The elements of the loop are as follows:

Element	Purpose
!X	A PML local (or global) variable that is automatically updated with the value of the loop-counter every time round the loop. If you do not supply a loop variable, a hidden unnamed variable will be used for the counter. Note that the loop variable is **REAL**. PML will delete any pre-existing variable of the same name. After the loop has finished, the counter variable will retain the value it had last time through the loop.
from	Defines the value of the counter for the first time round the loop. If you do not give a value, it will default to 1.
to	Defines the value of the counter for the final time round the loop. The default is infinity.
by	Defines the step size (positive or negative) by which the counter value is changed each time round the loop. The default value is +1.

5.2.1 Stopping a DO loop: break and breakif

If the `to` option is not given, the command block within the loop will be repeated indefinitely. To stop looping, use either the `break` or `break if` command:

```
do !Number
 if (!Number GT 100) then
 break
 endif
 !Result = !Result + !Number
enddo
do !Number
 break if (!Number GT 100)
 !Result = !Result + !Number
enddo
```

Any expression may be used with the `breakif` command, even a PML Function, provided the result is **BOOLEAN** (i.e. **TRUE** or **FALSE**).

The ( ) parentheses are optional but recommended.

The loop counter variable retains the value it had when the break from the loop occurred.

5.2.2 Skipping Commands in a DO Loop using Skip or Skip if

A `skip` command is used to go back to the beginning of the `do` loop, to repeat the loop with the next value of the `do` loop counter, omitting any commands following the `skip` command. The following example skips odd numbers:

```
do !X
 !Number = !Sample[!X]
 if ((INT(!Number/2) NE (!Number/2)) then
    skip
 endif
 !Result = !Result + !Number
enddo
```

The `skip if` command can sometimes be more convenient:

```
do !X
  !Number = !Sample[!X]
  skip if (INT(!Number/2) NE (!Number/2))
  !Result = !Result + !Number
 enddo
```

5.2.3 Nested DO Loops

You can nest `do`-loops one inside the other. The counter for each loop must have a different name.

```
do !X to 10
 do !Y
 !Z = !X + !Y
 break if (!Y GT 5)
 enddo
enddo
```

The inner-most loop goes through all the values of `!Y` before the outer loop moves on to the second value of `!X`. Note that the `break` (or `skip`) command acts just on the loop containing it - in this case the inner-most loop.

5.3 Jumping to a Labelled Line

The `golabel` command in PML allow you to jump to a line with a matching label name

```
LABEL /FRED
:
:
```

```
GOLABEL /FRED
```

The label name /FRED has a maximum length of 16 characters, excluding the / slash which must be present. The next line to be executed will be the line following LABEL /FRED, which could be before or after the GOLABEL command.

5.3.1 Conditional Jumping to a Labelled Line

```
do !A
 do !B to 3
 !C = !A * !B
 golabel /finished if (!C GT 100)
 !Total = !Total + !C
 enddo
enddo
label /finished
$P Total is $!Total
```

If the expression !C GT 100 is **TRUE** there will be a jump to LABEL /FINISHED and PML execution will continue with the line

```
$P Total is $!Total
```

If the expression is **FALSE**, PML execution will continue with the line:

```
!Total = !Total + !C
```

5.3.2 Illegal Jumping

The following is an illegal jump into a nested do block. It is however permitted to jump out of a nested block to a line in an enclosing block.

```
golabel /illegaL                    $* WRONG
do !Count to  5
 !Total = !Total + !Count
 label /illegal
enddo
```

6 Arrays

6.1 Creating Array Variables

An **ARRAY** variable can contain many values, each of which is called an **array element**. An Array is created automatically by creating one of its array elements.

For example to create element one of a new array named `NewArray`:

```
!NewArray[1] = !NewValue
```

This will create an array variable named **!NewArray** if it does not already exist and will set element 1 of that array to the value of **!NewValue**. If **!NewArray** already exists as a simple variable, you will get an error message and the command will be ignored.

Note: How array elements are referred to by means of a subscript expression in [ ] square brackets and that there must be no space between the end of the array name and the subscript. Array elements are accessed in the same way when using the value in an expression:

```
!Average =  ( !Sample[1] + !Sample[2] + !Sample[3] ) / 3
```

It is also possible to create an empty array which has no elements:

```
!X = ARRAY()
```

The array subscript may be any expression which evaluates to a number. One of the simplest expressions is the value of a variable:

```
!MyArray[!Next] = 15
```

The individual elements of an array variable can be set independently and in any order. Thus you can set `!X[1]` and `!X[10]` without setting any of the intervening elements `!X[2]` to `!X[9]`. In other words PML arrays are allowed to be ‘sparse’ and to have gaps between the subscript numbers which have values set.

Note: Negative subscripts are no longer permitted.

An array subscript of zero is allowed but you are advised against using it as many of the array facilities ignore array element zero.

An array subscript may be an expression of any complexity provided it evaluates to a positive **REAL** result and may even include a call to a PML Function:

```
!Value = !MyArray[!A + (!B * !!MyFunction() ) + !C ]
```

PML Arrays may be heterogeneous. That is to say the elements of a PML array do not have to be all of the same type. Array elements may even be user-defined objects.

Non-existent array elements — after the last set element of the array and the non-existent elements in the gaps of a sparse array — are all **UNDEFINED**: the function `Undefined()` will return **TRUE** and the function `Defined()` will return **FALSE** for all these subscripts.

6.2 Arrays of Arrays (Multi-dimensional Arrays)

An array element may itself be an array:

```
!Surname = 1
!Forname = 2
!Employee[1][!Surname] = 'Smith'
!Employee[1][!Forname] = 'Samuel'
!Employee[2][!Surname] = 'Bull'
!Employee[2][!Forname] = 'John'
!FullName = !Employee[1][!Forname] & ' ' &
!Employee[1][!Surname]
```

The value **!FullName** will be **'Samuel Smith'**.

It is also possible to assign an entire array to an array element:

```
!TempName[!Surname] = 'Truman'
!TempName[!Forname] = 'Harry'
!Employee[3] = !TempName
!FullName = !Employee[3][!Forname] & ' ' &
!Employee[3][!Surname]
```

The value **!FullName** will now be 'Harry Truman'.

6.3 Array Methods

Note: The *Software Customisation Reference Manual* contains a list of all PML Methods.

Array methods are built-in functions for performing a variety of operations on the array. These methods are invoked with a dot following the array name. The method name must be followed by ( ) parentheses - even if the function has no arguments:

```
!Nelements = !MyArray.Size()
```

This method sets `!Nelements` to the number of elements currently in the array. This is an example of a ***no-effect*** method which does not alter the array but returns a **REAL** result which can be assigned to another variable.

Here is method which does change the array:

```
!MyArray.Clear()
```

This is an example of a method which ***modifies*** the array by deleting all the array elements but produces ***no-result*** value, so there is nothing to assign to another variable.

There is a third kind of method which both changes the array and returns a result value. Here the result is assigned to a variable:

```
!NewArray = !OldArray.RemoveFrom(5,10)
```

This is an example of a method result which modifies the array by removing 10 elements, starting at element 5. **NewArray** value is a new array containing the 10 removed elements.

If not required, the result can be simply discarded by invoking the method as a command and not assigning the result to a variable:

```
!OldArray.RemoveFrom(5,10)
```

Note: Always use an array method, if one is available, in preference to constructing a **do**-loop as it is far more efficient.

6.3.1 Appending a New Element to an Existing Array

To set a new element at the end of an existing array without needing to know which elements are already set, use the `Append()` method thus:

```
!Result.Append(!NewValue)
```

The new array element to be created is determined automatically by adding 1 to the highest existing index for the array **!Result**. The data are stored in **!Result[1]** if the array does not yet contain any elements.

The array **!Result** must exist before you can call this method. If necessary you can create an empty array (with no elements) beforehand:

```
!Result = ARRAY()
```

If **!Result** exists already as a simple variable, you will get an error and the command is ignored.

6.3.2 Deleting an Array Element

To destroy an array element use the `Delete()` method.

```
!MyArray[N].Delete()
```

The deleted array element would test as **UNDEFINED**. Note that the array continues to exist even when you have deleted all its elements.

6.3.3 Deleting an Array

To delete the entire array, together with any existing array elements, use the Delete method on the array itself:

```
!MyArray.Delete()
```

6.3.4 Splitting a Text String into Array Elements

You can split a text string into its component fields and store each field in a separate array element using a text string's `Split()` method. This can be useful after reading a record from a file into a variable. By default, the delimiter is any white-space character (tab, space or newline):

```
!Line = '123 456 789'
!ArrayOfFields = !Line.split()
```

The white-space is treated as a special case since consecutive white-spaces are treated as a single delimiter and white-spaces at the start or finish of the string are ignored.

The result is to create the array-variable **!ArrayOfFields** (if it does not already exist) with the element **!FIELDS[1]** set to **'123'**, **!FIELDS[2]** set to **'456'**, and **!FIELDS[3]** set to **'789'**.

To specify a different delimiter, specify the required (single) character as the argument to the `Split()` method:

```
!Line = '123 ,456 ,,789'

!ArrayOfFields = !Line.split(',')
```

In this example, comma is used as the delimiter. **!ArrayOfFields** is created if it does not already exist with the element **!FIELDS[1]** set to **'123'**, **!FIELDS[2]** set to **'456'**, **!FIELDS[3]** created but set to zero length, and **!FIELDS[4]** set to **'789'**. Note that in this case, unlike the white-space delimiter, consecutive occurrences of the comma define empty elements.

Note: The only way to set the special white-space delimiter is by default; that is, by not specifying any delimiter as an argument to the `Split()` method. If a space is specified explicitly as the delimiter (as ' '), it will behave in the same way as comma in this example.

You can combine an array-append method with a text-string `Split()` method in a single command to append the fields of a text string to the end of an existing array variable, thus:

```
!ArrayOfFields.AppendArray(!Line.Split())
```

6.3.5 Length of String Array Elements

The length of the longest element in an array can be a useful thing to know, for example when you are outputting the values in table form. You can do this using the `Width()` method.

For example, in the array **!LIST**:

```
!LIST[1]'One'

!LIST[2]'Two'

!LIST[3]'Three'
```

The command:

```
!Width = !List.width()
```

would set to **!Width** to the value 5, the length of the longest element in **!LIST**, which is **'Three'**.

Note: If the array contained elements that were not strings, these are ignored when calculating the maximum width.

6.4 Sorting Arrays Using Array Methods

The simplest way of sorting an array is to use the `Sort()` method:

```
!MyArray.Sort()
```

This is a no-result method that modifies the array by performing a sort in-situ.

The sort is into ascending order and will be an ASCII sort on an array of **STRING** elements and a **NUMERIC** sort on an array of **REAL** values.

The `Sort()` method returns the array itself as a list result, so it is possible to follow the call to the `Sort()` method immediately with a call to the `Invert()` method, which will return a descending sort:

```
!MyArray.Sort().Invert()
```

An alternative approach is an indirect sort using the method `SortedIndices()` which returns a **REAL** array representing new index positions for the array elements in their sorted positions:

```
!NewPositions = !MyArray.SortedIndices()
```

The array can be sorted by applying the new index values using the `ReIndex()` method:

```
!MyArray.ReIndex(!NewPositions)
```

More important, the index values in **!NewPositions** can be used to sort other arrays as well.

To use some simple examples, imagine we had the array **!Animals** that contained:

Index	Animal
[1]	Wombat
[2]	Kangaroo
[3]	Gnu
[4]	Aardvark
[5]	Antelope

The command:

```
!Animals.Sort ()
```

Would move the array elements so that they now appeared in the following order:

Index	Animal
[1]	Aardvark
[2]	Antelope
[3]	Gnu
[4]	Kangaroo
[5]	Wombat

On the other hand, the command:

```
!Index = !Animals.SortedIndices()
```

Would create a new array **!index** representing the subscript values of the array elements in **!Animals** then sort these index values (without altering the array **!Animals**). After the sort, **!index** would look like this:

Index	Subscript
[1]	4
[2]	5
[3]	3
[4]	2
[5]	1

The command:

```
!Index.Invert()
```

Would result in **!index** now looking like:

Index	Subscript
[1]	1
[2]	2
[3]	3
[4]	5
[5]	4

6.5 Sorting Arrays using the VAR Command

These facilities were designed to work with arrays of STRINGS.

Where a multi-level sort is required it is still necessary to use the older facilities of the **VAR** command to perform the sort. Using a different example, look at the arrays !Car, !Colour and !Year:

	!Car	!Colour	!Year
[1]	CHARIOT	MUD	26
[2]	FORD	RED	1978
[3]	VAUXHALL	YELLOW	1990
[4]	FORD	YELLOW	1993
[5]	FORD	(Unset)	1986
[6]	(Unset)	BLUE	1993

[7]	Ford	GREEN	(Unset)
[8]	(Unset)	(Unset)	(Unset)
[9]	vauxhall	YELLOW	1994
[10	FORD	YELLOW	1993

If you want to sort the arrays by car model, then on colour, and then on year, you would give the command:

```
VAR !Index SORT !Car CIASCII !Colour !Year NUMERIC
```

The sorted index values are put into the array **!Index**, (for clarity shown here with the rows of the original arrays they are pointing to),

	!Index		**!Car**	**!Colour**	**!Year**
[1]	1	⇨	CHARIOT	MUD	2
[2]	7	⇨	Ford	GREEN	(Unset)
[3]	2	⇨	FORD	RED	1978
[4]	4	⇨	FORD	YELLOW	1993
[5]	10	⇨	FORD	YELLOW	1993
[6]	5	⇨	FORD	(Unset)	1986
[7]	3	⇨	VAUXHALL	YELLOW	1990
[8]	9	⇨	vauxhall	YELLOW	1994
[9]	6	⇨	(Unset)	BLUE	1993
[10	8	⇨	(Unset)	(Unset)	(Unset)

By default values are sorted ascending order using the ASCII character-codes. Other options which may be supplied after each array name are:

Sorting Option	**Effect**
CIASCII	Sorts in ascending **case-independent alphabetic** order.
DESCENDING	Sorts in **alphabetic in reverse** order
CIASCII DESCENDING	Sorts in descending **case-independent alphabetic** order.
NUMERIC	Forces an ascending **numerical sort** on numbers held as strings.
NUMERIC DESCENDING	Forces a descending **numerical sort** on numbers held as strings.

You can also modify the array to eliminate empty or repeated elements:

Option	Effect
UNIQUE	Eliminates instances of duplicated data. For example: **VAR !Index SORT UNIQUE !Car CIASCII !Colour !Year NUMERIC** **!Index** would then have only 9 elements as the value in the original row 10 was a second reference to a 1993 yellow Ford which is discarded.
NOUNSET	Eliminates rows that contain only **UNSET** values.
NOEMPTY	**VAR !Index SORT NOUNSET !Car CIASCII !Colour ! Year NUMERIC** Again **!Index** would have only 9 elements as the original row 8 is discarded. The option **NOEMPTY** discards rows with values that are all blanks.

To sort these arrays and identify the last occurrence of each group of the same car type, use the **LASTINGROUP** option:

```
VAR   !Index   SORT     !Car   LASTINGROUP   !Group
```

This would create **!Index** by sorting the values in the array **!Car** , but would also would also create the array **!Last**:

	!Index	!Group		!Car
[1]	1	1	⇨	CHARIOT
[2]	2		⇨	FORD
[3]	4		⇨	FORD
[4]	5		⇨	FORD
[5]	10	2	⇨	FORD
[6]	3	3	⇨	VAUXHALL
[7]	7	4	⇨	ford
[8]	9	5	⇨	vauxhall
[9]	6		⇨	(Unset)
[10]	8	6	⇨	(Unset)

A similar option **FIRSTINGROUP** identifies the first row of each group.

6.6 Subtotalling Arrays with the VAR Command

Suppose we had sorted the array **!Car** and another array **!Value** using the command:

```
VAR !Index SORT !Car !Value LASTINGROUP !Group
```

The arrays would look like this:

	!Index	!Group		!Car	!Value
[1]	1	1	⇨	CHARIOT	50000
[2]	2		⇨	FORD	1000
[3]	4		⇨	FORD	100
[4]	5		⇨	FORD	8000
[5]	10	2	⇨	FORD	7000
[6]	3	3	⇨	VAUXHALL	2000
[7]	7	4	⇨	ford	1000
[8]	9	5	⇨	vauxhall	3000
[9]	6		⇨	(Unset)	9000
[10]	8	6	⇨	(Unset)	6000

We can then generate an array of subtotals for each type of car with the following command:

```
VAR !Totals SUBTOTAL !Values !Index !Group
```

This would produce the result:

	!Index	!Group		!Car	!Value	!Totals
[1]	1	1	⇨	CHARIOT	50000	50000
[2]	2		⇨	FORD	1000	
[3]	4		⇨	FORD	100	
[4]	5		⇨	FORD	8000	
[5]	10	2	⇨	FORD	7000	16100
[6]	3	3	⇨	VAUXHALL	2000	2000
[7]	7	4	⇨	ford	1000	1000
[8]	9	5	⇨	vauxhall	3000	3000

[9]	6		⇨	(Unset)	9000	
[10]	8	6	⇨	(Unset)	6000	15000

6.7 Arrays of Objects

6.7.1 DO Values with Arrays of Objects

Imagine that we have already defined an object type **EMPLOYEE** which contains a member of type **STRING**. We create an array of **EMPLOYEE** objects and set the **Name** member of each of each object.

```
!MyEmployees[1] = object EMPLOYEE()
!MyEmployees[2] = object EMPLOYEE()
!MyEmployees[3] = object EMPLOYEE()
!MyEmployees[1].Name = ‘James’
!MyEmployees[2].Name = ‘Joe’
!MyEmployees[3].Name = ‘Mike’
```

We then wish to create a new array containing just the employees’ names.

```
!Names = ARRAY()
do !Person VALUES !MyEmployees
 !Names.Append(!Person.Name)
enddo
```

Each time round the do-loop, the do-variable becomes an object of the appropriate type, even if the array is a heterogeneous mixture of objects.

6.7.2 Block Evaluation And Arrays

A convenient way of performing the same command on each element of an array is to use **block evaluation**. This in effect provides a way of building your own array-methods.

There are two steps involved: first create a **BLOCK** object from the command text, then apply this **BLOCK** to that array with the **EVALUATE** array method. Here is an illustration:

First we create a **BLOCK** out of the command to be processed for each array element:

```
!ExtractName = object
BLOCK('!MyEmployees[!Evalindex].Name')
```

Note that the command is enclosed in quotes. The special variable **Evalindex** is automatically incremented during the evaluation of the block to the index of each array element in turn. Finally we use the `Evaluate()` method to invoke block processing on the array:

```
!Names = !MyEmployees.Evaluate(!ExtractName)
```

!Names will contain the names as an array of **STRING** elements.

Take care that the expression does in fact return a value so that new array's elements can be created and assigned to it.

Alternatively, you may use an evaluation expression which does not return a result provided you invoke the evaluation directly without trying to assign the result. For example, if you have defined a `!!Print()` function the following is valid:

```
!Output = object
BLOCK('!!PRINT(!MyEmployees[!Evalindex].Name)')

!MyEmployees.Evaluate(!Output)
```

Note: In this release there is no object-oriented shorthand representing 'this element'. It is necessary to give the array name along with a subscript expression using **!Evalindex**.

6.7.3 Using Block Evaluate to Sort an Array of Objects

There are no built-in array methods yet for sorting an array of objects. A possible technique is to use a block evaluation to create an array of keys to sort:

```
 - - Create a block to extract the object member on which to sort
!ExtractKeys = object BLOCK ('!MyEmployees[!Evalindex].Name')
 - - Evaluate the block to generate an array of sort keys
!SortKeys = !MyEmployees.Evaluate(!ExtractKeys)
 - - Create a new array containing the sorted element positions
!NewOrder = !SortKeys.SortedIndices()
 - - Apply the sorted element positions to the original array
!MyEmployees.ReIndex(!NewOrder)
```

This Evaluate method is the equivalent of the older command:

```
VAR .. EVALUATE .. INDICES ARRAY
```

Note: In particular that there is no automatic database navigation.

6.7.4 DO VALUES and DO INDICES with Arrays

With do values the counter takes the value of each array element in turn:

```
!Pump[1] = 'Fred'
!Pump[20] = 'Tom'
!Pump[10] = 'Jim'
do !Name values !Pump
 Sp array element is $!Name
enddo
```

This will produce the following output:

```
Array Element is Fred
Array Element is Jim
Array Element is Tom
```

On exit from a `do values` loop PML destroys the loop counter variable.

With `do indices` the counter takes the value of each array subscript at which an array element is stored:

```
!Pump[1] = 'Fred'
!Pump[20] = 'Tom'
!Pump[10] = 'Jim'
do !IN indices !Pump
 !Value = !Pump[!IN]
 SP Array Element $!N is $!Value
enddo
```

This will produce the following output:

```
Array Element 1 is Fred
Array Element 10 is Jim
Array Element 20 is Tom
```

7 Macros

7.1 PML Macros

Macros are command sequences that are stored in text files.

To access a macro file and input the command sequence to your program, you **run** the macro. The file is scanned line-by-line, with exactly the same effect as if you were typing the lines in from a keyboard.

Macro files may include synonyms and user-defined variables. They may also act on data which you define when you give the command to run the macro (parameterised macros). Macros are permanent records which may be called from within any working session of a program.

7.1.1 Naming and Running Macros

A macro file can be given any name that conforms to the naming conventions of the operating system.

The suffix `.mac` is often used to indicate that the filename is that of a macro, but this is optional.

The process of reading the contents of a macro file as program input is known as **running the macro**. The program does not discriminate between input from the GUI, the keyboard, or a macro file.

To run a macro, enter:

`$M` *`filename`*

where ***filename*** is the pathname of the macro file. The filename may optionally be preceded by a slash (/) character.

If you give only the name of the file, the program will look for it in the directory from which you executed the program.

An error message will be output if the file cannot be found and opened for reading.

7.1.2 Macros with Arguments

It is often convenient to write a macro in a generalised form, using parameters to represent dimensions, part numbers etc, and to assign specific values to those parameters only when the macro is to be run.

In the simplest case, parameters are allocated positions in the command lines as **macro arguments** by inserting escape codes of the form

$*n*

where ***n*** is an integer in the range 1 to 9. These arguments are then specified as part of the command to run the macro. They follow the macro name, with spaces separating the individual arguments.

For example, if a macro named `beam.mac` includes the command line

```
NEW BOX XLEN $1 YLEN $2 ZLEN $3
```

then the macro call

```
$M/BEAM.MAC 5000 200 300
```

will run the macro and will set the lengths defined as **$1**, **$2**, **$3** to 5000, 200 and 300 respectively.

Arguments may be either values or text, but note that a space in a text string will be interpreted as a separator between two different arguments.

Apostrophes in text arguments are treated as parts of the arguments, ***not*** as separators between them. For example, if a demonstration macro `arg.mac` includes the lines:

```
$P First Argument is $1
$P Second Argument is $2
$P Third Argument is $3
```

and is called by the command

```
$M arg.mac 'Arg1a Arg1b' 'Arg2' 'Arg3'
```

the resulting output will be

```
First Argument is 'Arg1a'
Second Argument is 'Arg1b'
Third Argument is 'Arg2'
```

whereas the intended output was

```
First Argument is 'Arg1a Arg1b'
Second Argument is 'Arg2'
Third Argument is 'Arg3'
```

If you need to include spaces or newlines in an argument, you must enclose the argument between the escape codes $< and $>.

The correct form for the preceding example is therefore

```
$M/arg.mac $<'Arg1a Arg1b'$> 'Arg2' 'Arg3'
```

As an alternative, you may redefine the separator between arguments to be the escape code

$,

instead of a space.

If you do this, you must end the argument list with the escape code

$.

Note: The full stop is part of the escape code, not punctuation.

Using this convention, the preceding example becomes

```
$M/ARG.MAC $,'Arg1a Arg1b'$,'Arg2'$,'Arg3'$.
```

If an argument is omitted when a macro is called, the $n in the macro command line is ignored. Whether or not this leaves a valid command depends upon the syntax which applies.

To avoid possible errors, a default setting may be defined for each argument. If the argument is omitted when the macro is called, the default setting will be used.

To define an argument's default setting, use the command

```
$Dn = default_string
```

where ***n*** is the argument number (1-9) and ***default_string*** is any sequence of characters ending in a newline.

The default setting may be specified at any point in the macro, but it will only be applied from that point onwards. It is usually best, therefore, to define any defaults at the beginning of the macro. If an argument has been specifically defined, a subsequent default specification is ignored.

Arguments may be omitted in the following ways:

- If the normal macro calling sequence is used (spaces as separators, Return as end-of-line marker), trailing arguments may simply be omitted
- If a non-trailing argument is to be omitted, the escape code $<$> must be used to replace the argument which is not required
- If the $, argument separator is being used, the argument may be omitted from the list

For example, if the macro `demo.mac` expects three arguments, the following calls to run the macro all omit one of the arguments:

Macro Call	Effect
$M/demo.mac arg1 arg2	Omits third argument
$M/demo.mac arg1 $<$> arg3	Omits second argument
$M/demo.mac $,$,arg2$,arg3$.	Omits first argument

7.2 Using Macros and Functions Together

The existing mechanism for invoking a macro using $M continues to be available alongside the new PML Functions. Most PML code in existing macros should continue to work unmodified.

To call a macro, even from within a PML Function, use the $M command. Variables used as arguments to a macro must be **STRING** values. PML Variables must be converted to a **STRING** with a preceding $:

```
$M filename $!X $!Y $!Z
```

Note: To gain the full benefits of using PML Functions it is best not to mix Macros and Functions extensively. Many of the advantages of PML Functions depend on using variables without $ which is not possible with arguments to macros.

7.3 Synonyms in PML Macros and Functions

You may use **synonyms** in macros, although you can ***not*** use them in PML Functions.

This restriction was introduced because changing a synonym (which is global to the whole program) can cause an unintentional change in behaviour of individual macros.

There is also a large performance penalty on every line of PML executed where synonyms are permitted.

8 Using PML in AVEVA Products

Most of this manual describes how to use PML 2 to create and customise Forms and Menus. This chapter describes how to use PML within AVEVA products. Note that for tasks such as defining Rules and Report Templates, you are restricted to the PML 1 expressions package described in the *Software Customisation Reference Manual*, and it is also sometimes necessary to use the VAR command.

8.1 Composing Text

It is sometimes necessary to arrange text in multiple columns. The **COMPOSE** facility of the **VAR** command will help you do this.

For example, if we create the following text string:

```
!A = 'The quick brown fox jumped over the lazy dogs'
```

To compose the text we might use the following command:

VAR !Table COMPOSE |$!A| WIDTH 11 C SPACES 2 |$!A| WIDTH 15 R

This would give the following output:

Index	**Value**
[1]	The quick The quick brown'
[2]	brown fox fox jumped over'
[3]	jumped over the lazy dogs'
[4]	the lazy '
[5]	dogs '

COMPOSE always returns an array with at least one element. The number of array elements depends on the length of the text strings supplied and the width of each column. Notice that all of the **STRING** array elements are space-padded to the same length.

Following the **COMPOSE** keyword is a list of column definitions. For each column, there is a text string, such as **|$!A|** which evaluates to a text string, followed by the column layout keywords in any order:

Keyword	Effect
WIDTH *n*	Specifies the space-padded width of this column. If not specified the width of the column will be the length of the input string.
SPACES *n*	Specifies the number spaces between this and the next column.
L **LEFT**	Specifies text is to be aligned along the left edge of the column.
R **RIGHT**	Specifies text aligned along the right edge of the column
C **CENTRE**	Specifies justification in the centre of the column.
DELIMITER ' '	This can optionally be used to specify an alternative delimiter at which to break the input. By default the text will be split at a white-space character such as space which may be removed from the output text. If the delimiter is specified as an empty string, the text will be split at the column edge whatever the content.

8.2 Defining Variables for Late Evaluation (Rules)

The following are used to pass the ***name*** of a variable into PDMS as part of a stored expression so that the value is taken when PDMS processes the stored expression rather than PML extracting the value at the time the line is read:

```
VVALUE( !X )
VTEXT( !AString )
VLOGICAL( !Aboolean)
```

8.3 Using PML in PDMS

The following facilities are only applicable to PDMS.

8.3.1 Accessing DB Elements As Objects

A special global variable **!!CE** has been provided which always refers to the **current element** in the database. **!!CE** can be used to obtain the **DB reference** of the current element:

```
!Item = !!CE
```

!!CE is of type **DBREF** so the new variable **!Item** will also be of type **DBREF**. The **dot notation** can be used to access attributes and pseudo-attributes of a database element:

```
!Bore = !Item.bore
```

This form of access can also be used directly on the **!!CE** variable:

```
!Owner = !!CE.owner
```

It is also possible to follow references between DB elements using this mechanism:

```
!Rating = !!CE.cref.pspec.rating
```

Assigning a new reference to the **!!CE** variable makes the new reference the current element by navigating to it in the database:

```
!!CE = !!CE.owner
```

P-points are accessed using the P-point number like an array subscript. For example, to access the direction of **P-point[1]**:

```
!Dir  = !!CE.Pdirection[1]      $* !Dir is a DIRECTION object
```

To access the position of **P-point[3]**:

```
!Pos = !!CE.Pposition[3]        $* !Pos is a POSITION object
```

A **NULREF** is treated as **UNSET**, so a **NULREF** can be tested for in two ways:

```
if ( !MyDBRef  EQ  NULREF ) then
if ( UNSET( !MyDBRef ) ) then
```

There is also the function **BADREF** which will detect whether a database reference is unset or invalid (i.e. impossible to navigate to):

```
if (BADREF( !MyDBRef ) ) then . . .
```

Note: Use full-length names for attributes as listed in the Appendix for compatibility with future releases.

8.3.2 Assigning Values to Element Attributes

You can assign a new value to a **DBREF** attribute, ensuring that the type of the new value matches the type of the attribut

```
!!CE.Built = TRUE
```

You can still assign an attribute value in this way even if the PML **DBREF** is not the current element':

```
!A = !!CE
!!CE = !!CE.Owner
!A.Built = TRUE
```

You can even assign a PML object, such as **POSITION**, where this corresponds to the type of the attribute:

```
!!CE .Position = !NewPosition
```

Note that where the type of an attribute is a PML object, it is not possible to set an object member value, such as the **up** value of a **POSITION**, directly - this must be done in two stages:

```
!Pos = !!CE.Position
!Pos.Up = 2000
!!CE.Position = !Pos
```

8.3.3 Accessing Information About a Session

A number of special commands have been provided to set a PML Variable with information about the current session. The commands are:

```
Current Session
Sessions
Projects
Teams
Users
MDBs
DBs
```

These commands can be used as in the following example. The **SESSION** object has a method that returns name of the **MDB** in the current session. Hence:

```
!C = current session
!CurrentMDB = !C.MDB()
```

This will set **!CurrentMDB** to the name of the current **MDB**.

8.3.4 Evaluating Selected DB Elements

Using the facilities described here you can create an expression and have it evaluated for all elements which satisfy particular selection criteria. The results of the expression are then placed in a named array.

The command syntax is:

```
VAR !Array EVALUATE (Expression) FOR select COUNTVAR !Counter
```

Where:

!Array	is the name of the array that will be created to contain the results of (***expression***) for all the elements selected within `select`.
(expression)	is the expression that will be carried out for all the elements that match the `select` criteria.

`select`	is the selection criteria (see above, and the relevant Reference Manual for your product for details of selection criteria)
COUNTVAR	is an optional command which allows you to record how often the expression is calculated in **Counter** , which is increased by one each time the expression is evaluated.

You can append the results of such an evaluation to an existing array using the APPEND keyword. For example:

```
VAR !BOXES APPEND EVALUATE ( XLEN*YLEN ) FOR ALL BOXES
```

will add the values calculated from the expression for all **BOXES** to the (already existing) array BOXES.

You can also overwrite elements in the array by specifying the first index in the array which you want to be overwritten. The specified index, and the indexes following it, will be overwritten by the results of the evaluation. For example:

```
VAR !BOXES[99] EVALUATE ( XLEN*YLEN ) FOR ALL BOXES
```

will place the result of the first evaluation for the selected elements at index 99, overwriting any existing item, and the following results in the subsequent array elements.

8.3.5 RAW keyword When setting Variables with VAR

Programs that use the Forms and Menus interface (e.g. the PDMS DESIGN, DRAFT and ISODRAFT modules) strip out line feeds and compress consecutive spaces to a single space before text strings are assigned to array variables. If this is not the effect you want, you can suppress this automatic editing by including the RAW keyword into the variable-setting command line for these programs.

The syntax for setting array variable elements to **unedited text strings** is

```
VAR !VarName  RAW  …
```

where … represents any of the standard VAR syntax for setting variables.

8.3.6 Undo and Redo

The Undo and Redo functionality has been exposed to PML so you can create your own set of undoable events.

There are several ways PDMS adds entries to the undo system:

- Using the **MARKDB / ENDMARKDB** commands. The syntax is as follows:

  ```
  MARKDB 'text'
  ```

- where text is an optional description to be included with the mark. This causes a mark to be made in the database and an entry made in the undo stack. You should make your database changes, and then use the command

  ```
  ENDMARKDB
  ```

- By creating a PML undoable object and adding it to the undo stack. See the *Software Customisation Reference Manual* for a fuller description of this object. You should create an undoable object, set up the undo and redo execution strings, and then call the method `add()` to mark the database and add the undoable to the undo stack. Make your database changes, and then call the method `endUndoable()`.

- Automatically whenever a model element is moved using graphical interaction techniques in Model Editor.

  Additionally you may register to be informed whenever an undo or redo operation has taken place, using the PML PostEvents object. See the *Software Customisation Reference Manual* for a fuller description of this object.

After an undoable has been removed from the stack and its state recovered then the user-supplied method on the PostEvents object is called, and will be passed the description text that was associated with the undoable object.

9 Copies and References (for Advanced Users)

Assignment using = , the **assignment operator**, usually does what you would expect, but a detailed explanation of how the different data types are handled may be helpful.

9.1 Assignment

Assignment always makes a **copy** of the right-hand-side to replace what is on the left-hand-side. The following command copies **!Y** into **!X**:

```
!X = !Y
```

If, for example, **!Y** is an array, this command will duplicate the entire array making a copy of each of the array elements. The same is true if **!Y** is an **OBJECT**.

The technical term for this is **deep copy**. Following this copy

```
!X[1] = 'New Value'
```

will change **!X[1]** but leave the original array **!Y** unchanged.

9.2 Form and Gadget References

!!Form and **!!Form.Gadget** are both **PML references**. After the command:

```
!X = !!Form.Gadget
```

!X is now a new reference, but the **gadget** itself has not been copied. Both PML Variables now refer to the same gadget.

```
!X.val = 'New Value'
```

```
!!Form.Gadget = 'New Value'
```

will both have the same effect and will assign a new value original gadget. You can think of a reference as another name for the same object or variable.

9.3 DB References

!!CE is a **DB reference**. Following the command:

```
!X = !!CE
```

!X is now a new reference to the same **DB element**, but the element itself has not been copied.

```
!Value =  !X.Attribute
```

will now return an attribute of the current element.

When the current element changes, **!!CE** will point to a new DB element. In this example, **!X** will not be changed but remains as a reference to the ***previous*** current element. Because **!!CE** is special,

```
!!CE = !X
```

will **navigate** to a new current element provided **!X** is another DB reference. In this example the command would navigate to the previous current element.

9.3.1 Deep Copy involving References:

Where **!Y** is an array or object:

```
!X = !Y
```

will make a deep copy of **!Y**. However, if any of the array elements is a reference (e.g. to a gadget or DB element), a copy is made of the ***reference***, but ***not*** of the object it refers to. In other words a deep copy stops copying when it reached a reference.

9.4 Function Arguments

A **function argument** is a PML reference to a value or object outside the function. In effect the argument is another name for the original PML Variable.

If we define a PML Function such as the following:

```
define function !!ChangeIt ( !Argument is STRING)
 !Argument = 'New Value'
 $P !Argument
 $P !!Global Var
endfunction
```

Then invoke the function like this:

```
!!GlobalVar = ‘Old Value’
!!ChangeIt (!!GlobalVar)
```

The values printed for **!Argument** and **!!GlobalVar** will both be **'NewValue'**.

Warning: Be very careful about changing function arguments. It is a powerful feature capable of causing unexpected results.

9.4.1 Constants as Function Arguments

Passing a **constant** as a function argument, such as a **STRING** in quotes, means the argument is **read only** and cannot be assigned a new value. So if we define a function

```
define function !!ChangeString( !Argument is STRING)
 !Argument = 'New Value'
endfunction
```

The following will change the value of **!S**:

```
!S = ‘Old Value’
!!ChangeString ( !S )
```

However, the following will result in a PML error message because the value passed to the function as an argument is a **CONSTANT STRING** value which cannot be modified.

```
!!ChangeString ( 'OldValue' )          $* WRONG
```

9.4.2 Form and Gadget Properties Passed as Function Arguments

A form or gadget value passed as a function argument is read only so cannot be assigned a new value. If you wish to change the value of a gadget passed as an argument, pass the gadget itself as an argument, not its value:

```
define function !!ChangeValue( !Argument is GADGET)
 !Argument.val = 'NewValue'
endfunction
```

9.5 Database Attributes

PDMS Database Attribute are read only, and so they cannot be given new values by assigning to them.

10 Errors and Error Handling

10.1 Error Conditions

An **error condition** can occur because a command could not complete successfully or because of a mistake in a PML Macro or function. The presence of an error normally has three effects:

- An Alert box appears which the user must acknowledge
- An error message is output together with a traceback of any calling macros or functions.
- Any currently running PML Macros and functions are abandoned.

This example of an error is caused by an attempt to use a PML Variable that does not exist:

```
(46,28) ERROR - Variable FRED not defined
```

The **46** is the **module** or program section which identified the error and is the error code itself is **28**.

If the input line had been typed interactively that is the end of the story. However, if the input line was part of a PML Macro or function the error may optionally be **handled**.

10.2 Handling Errors

An error arising during the processing of a PML Macro or function does not immediately give rise to an error message - this depends on the next line of input.

Provided the next command processed by PML is a matching handle command, the error is not output and the commands within the matching **handleblock** are processed instead.

`elsehandle` blocks may optionally also be present — if the `handle` block does not match the error one `elsehandle` will be processed — if it matches the error:

```
$* a command causes Error(46, 28)
    handle  (46, 27)
     $* handle block - not processed this time
    elsehandle (46, 28)
     $* The commands in this matching handle block are processed next
    elsehandle ANY
     $* An ANY Handle Block is processed for any errors. I
     $* n this position it would handle errors other than  (46, 27) and
    (46, 28)
    elsehandle NONE
     $* A NONE Handle Block is processed only if there were no errors
```

```
endhandle
$* This line is processed after the handle block.
```

If **(46,27)** matches the error, PML processes the commands in that handle block instead of outputting an error. Processing of the current PML Macro or function continues at the line after the `endhandle` command.

Note: The keywords `ANY` and `NONE` which can ***optionally*** be used in place of a specific error code.

If the line following a line which has caused an error is not a handle command, the outcome depends on the current setting of `onerror` (see *next section*). The default setting is `onerror RETURN` and the current macro or function is abandoned.

However, if in the ***calling*** PML Macro or function the next command to be processed is a `handle` command, it now gets an opportunity of handling the error (and so on up the calling PML).

10.3 Setting the ONERROR Behaviour

The default setting is:

```
onerror RETURN
```

which causes the current macro or function to be abandoned and gives the calling macro or function an opportunity to handle the error.

Another possibility is to jump to one section of code in the same macro or function after any unhandled error:

```
onerror GOLABEL /LabelName
$* The first command to be executed after the label
$* must be a handle command - it could for example be a handle ANY
command:
label /LabelName
handle ANY
 $* handle block
endhandle
```

To suppress errors, and **error alerts** the following may be used, for example during non-interactive testing:

```
onerror CONTINUE
```

Although the effect of errors is suppressed, any error messages generated will still be output. Beware of using this option which can have unpredictable effects. The option is applied to all subsequently nested PML Macro and function calls.

When debugging, you may interrupt normal PML processing after an error:

```
onerror PAUSE
```

The effect is the same as `$M-` and will allow you to type commands while the PML File is suspended. You must type `$M+` to resume the PML processing.

Do not leave this in working code.

10.4 Other Responses to an Error

After handling an error, you can still output the detail of the error message with the following commands:

```
$P $!!Error.Text
$P $!!Error.Command
$P $!!Error.Line
do !line VALUES !!Error.Callstack
 $P $!Line
enddo
```

To abandon the running PML Macro or function, but to re-instate the error so that the calling PML code has a chance to handle it, you can use the command:

```
return error
```

You can also re-instate the error but suppress the alert using the command:

```
return error noalert
```

To generate a new error (or replace an error with your own error) plus an optional message, use one of the following

```
return error 1
return error 1 'Your error message'
return error 1 NOALERT
```

To handle such an error there is a special form of the `handle` command:

```
handle  1
PML code to handle a user-defined error number
endhandle
```

11 Handling Files and Directories

Reading and writing files is greatly simplified by using the **FILE** object. This chapter describes the methods most frequently used for reading and writing.

Methods are also provided for moving, copying and deleting files and extracting information such as the pathname of a file, its size data and time last modified.

A **FILE** object may refer to a directory rather than a file and methods are provided for navigating around the directory structure.

For a complete list of the methods available, refer to the *Software Customisation Reference Manual.*

11.1 Creating a File Object

A file object is created by invoking the **FILE** constructor with the name of the file as its argument. For example:

```
!MyFile = object FILE (‘c:\users\bob\list.txt’)
!MyFile = object FILE (/net/usr/bob/list’)
!MyFile = object FILE (‘%PDMSUSER%\bob\list.txt’)
```

At this stage, the file may or may not exist — creating the **FILE** object does not open or access the file in any way.

11.1.1 Example

This example reads pairs of numbers from file data, adds them together and writes the answers to file RESULTS.

```
!Input = object FILE('DATA')
!Input.Open(‘READ’)
!Output = object FILE('RESULTS')
!Output.Open(‘WRITE’)
do
 !Line = !Input.ReadRecord()
 if (!Line.set()) then
  !array = !Line.Split()
  !Total = !Array[1].Real() + !Array[2].Real()
  !Output.WriteRecord( !Total.String() )
 else
```

```
    break
  endif
enddo
!Output.Close()
!Input.Close()
```

11.2 Reading from Files

When reading a file one line at a time using the `ReadRecord()` method you must open the file first with the `Open('READ')` method and close it afterwards with the `Close()` method. Each line read from the file will be returned as a **STRING** until the end of the file is reached, when you will get an **UNSET STRING** returned.

The **UNSET** string can be detected using the **STRING** object's `Set()` method (or `Unset()`) as in the example above (See *Example*).

11.3 Writing to Files

When you open a file for writing, the file will be created if it does not already exist.

If you use `Open('WRITE')` and the file already exists, the user will be shown an alert asking whether the file can be overwritten. Alternatively you may specify `OVERWRITE`, to force overwriting of a file if it already exists or `APPEND` if you want to add to the end of a file if it already exists.

11.4 Reading and Writing ARRAYS

You will obtain much quicker performance if you read or write an entire array in a single operation. In particular you are recommended to use the `ReadFile()` method which returns the whole file as an array of strings, opening and closing the file automatically:

```
!Input = object FILE('DATA')
!Output = object FILE('RESULTS')
!Lines = !Input.ReadFile()
!ResultArray = ARRAY()
do !Line VALUES !Lines
 !Numbers = !Line.Split()
 !Total = !Numbers[1].Real() + !Numbers[2].Real()
 !ResultArray.Append(  !Total.String() )
enddo
!Output.WriteFile('WRITE', !ResultArray)
```

Note: With the `ReadFile()` method you may optionally specify the maximum number of lines you are prepared to read and an error is raised if this number is exceeded. If not specified, a limit of 10000 is imposed.

11.5 Error Handling When Using a File Object

Errors will mainly be encountered when an attempt is made to open a file that does not exist, has been mistyped or to which you do not have access. Anticipated errors may be dealt with in a `handle` command following use of the `Open()` method.

The errors most commonly encountered include the following:

```
(160,7) Argument to method is incorrect
(160,9) File does not exist
(160,36) Unable to read file record, file is not open
(160,37) Unable to write file record, file is not open
(160,44) File exists, user does not want to overwrite it
(160,47) File length has exceeded N lines
(41,319) Cannot access file (Privileges insufficient)
```

12 Developing PML Code

Fully developed PML code should - if required to - interact with the user by means of forms, menus, and alert boxes. However, while you are still developing your PML code the facilities described here might be useful.

The commands described here should normally only be typed interactively or included in a PML File on a temporary basis. None of these commands should remain in fully developed PML code.

12.1 PML Tracing

If a PML File does not appear to be running in the way you expect the easiest thing to do is to turn on PML Tracing with the command:

```
PML TRACE ON
```

With PML Tracing switched on, each of the PML lines executed is output to the parent window together with its line number. Additional tracing messages tell you when you have begun to process a new PML File and when it has finished. To turn off tracing type:

```
PML TRACE OFF
```

These commands can also be put into a PML File on a temporary basis.

You can also turn PML tracing on by setting the Environment Variable `PMLTRACE` to `ON` before you start the program.

For more precise control of PML tracing you can use the `$R` command. `$R100` is the equivalent of `PML TRACE ON` and `$R0` is the same as `PML TRACE OFF`.

Type the command `$HR` for online help on how to use the `$R` command.

12.2 Diagnostic Messages From Within PML Files

It is often useful to output a line to the screen from within a running PML File to indicate that the execution of the program has reached a particular stage. Use the $P command:

```
$P !Total is: $!Total and !Maximum is: $!Maximum
```

Note: The use of $ to convert the value of a variable to a **STRING**.

12.3 Alpha Log

The **alpha-log** is a file containing a record of all the commands processed together with any text output and error messages. To start recording use one of the following commands:

`alpha log /filename`	$* to open a new file
`alpha log /filename OVERWRITE`	$* to replace an existing file
`alpha log /filename APPEND`	$* to add to an existing file

To finish recording and close the file use:

```
alpha log ENDl
```

12.4 Alpha Log and PML Tracing

The alpha-log does not include standard PML tracing from the command `PML TRACE ON`. For PML tracing to be included in the alpha-log, PML tracing should be directed to the **alpha-window**. For example, to obtain standard PML tracing in the alpha-log use the command:

```
$R102
```

Refer to the online help given by $HR for other options.

12.5 Suspending a Running PML Macro

If you cannot find out what is going wrong by means of PML trace output or writing out the values of PML Variables, you may suspend a running PML File at a particular point by including the command:

```
$M-
```

You can then query the values of any PML Variables of interest and even change the values of some PML Variables. To resume processing of the suspended PML File type into the **command line**:

```
$M+
```

Use these facilities only for debugging and do not leave these commands in a finished PML File.

Note: These facilities do not apply to PML Functions or methods.

12.6 Querying PML

A variety of queries are available to help diagnose problems. Typically you might type these queries into the **command line** whilst a PML File is suspended. Alternatively you could include any of these commands temporarily within a PML File.

12.6.1 Querying the Currently Running PML File Stack

If you are interested in the name of the currently running PML File and the other PML Files from which the current file was invoked use the command:

```
$QM
```

12.6.2 Querying the Values of PML Variables

The table below gives some useful commands for querying PML Variables.

Command	Effect
`q var !LocalName`	Queries the value of a ***specific*** local variable, use the command.
`q var LOCAL`	Queries the values of ***all*** local variables.
`q var !!GlobalName`	Queries the value of a ***specific*** global variable.
`q var GLOBAL`	Queries the values of ***all*** global variables
`q var !MyArray[1]`	Queries the value of a ***specific*** element of an array.
`q var !MyArray`	Queries the values of ***all*** elements of an array.
`q var !MyArray.Size()`	Queries the number of elements currently in an array.

12.6.3 Querying What Can Be Typed Next

When you are typing commands into a macro or into a command line, you may not remember exactly what arguments are available or what other commands are related.

You will find full details in the appropriate Reference Manuals to the module, but you can also ask the **command processor** what sort of command or commands it is expecting next by typing:

```
$Q
```

This will produce a list of ***every*** valid command word or argument type that you may enter next.

The list may be long and could include every valid command for the module that you are in.

The `$Q` facility is also useful to establish what is allowed as the next component of a command. Type the beginning of the command sequence followed by `$Q`.

13 Form Concepts: Getting Started

13.1 Overview

As a user of AVEVA products, you will already be familiar with forms, and the gadgets (buttons, textboxes, etc) on them.

In PML 2, forms are a type of object represented by a global variable — the form's name, for example **!!EntryForm**. This means that a form cannot have the same name as any other object type, global variable, or any other form.

The form object owns a set of predefined member variables and built-in methods. In addition, you can define your own members — form variables and form gadgets — and your own form methods. All these will determine the content and functionality of the form.

Gadget objects are user-defined members of the form object. Form members are always accessed using the ***dot notation***, for example

```
!!EntryForm.TextField
```

Gadgets own a set of predefined member variables and built-in methods which determine the content and functionality of the gadget. For example, the value of a text field gadget is accessed by:

```
!!EntryForm.TextField.val.
```

Note that gadgets do not support user-defined member variables or user-defined gadget methods.

Note: All the in-built members and methods of forms and gadget-types are listed in the *Software Customisation Reference Manual*.

Callbacks are user-defined actions assigned to a form and its gadgets and that are executed when the operator interacts with the form, for example, by clicking a mouse button on a gadget.

The callbacks are supplied as text strings and may be any valid PML expression, PML Function or method call, including any form, gadget or user defined object method. Effectively, these callbacks determine the intelligence of the form.

13.2 Naming Forms and their Members

The following are examples of the format of form and member names:

!!EntryForm	The name of a form

!!EntryForm.GadgetName	The name of a gadget on a form
!!EntryForm.GadgetName.val	The data value held by that gadget

Note: That total length of the name is limited to 256 characters even though the length each individual components is limited to 64 characters. Within the form definition, the members of the form should be referred to by using **!This** to replace the form name part of the gadget name. For example:

```
!This.GadgetName
!This.GadgetName.val
```

Note: The obsolete convention of using an underscore to replace the form name has been retained for compatibility with earlier versions.

13.3 Simple Form

You define a form using a command sequence that starts with:

```
setup form !!formname
```

and ending with:

```
exit
```

Between these two commands come a number of optional subcommands which define the gadgets on the form.

The following example defines a small form, containing the text `'Hello World'`, and a button labelled `'Goodbye'`, which removes the form when pressed:

```
setup form !!hello
 paragraph .Message text 'Hello world'
 button .bye 'Goodbye' OK
exit
```

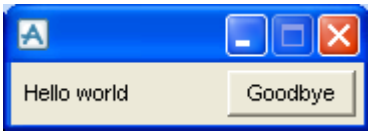


Figure 13:1. A Simple Form

Some points to note about the above definition and the form it creates are:

- there are no user-defined methods on this form, and so the setup `form . . . exit` sequence is the complete form definition;
- the paragraph command adds a paragraph gadget to the form (a paragraph gadget is just a piece of text displayed on the form). The name of the gadget is **Message**, and the dot before the name indicates that the gadget is a member of the form. The text itself is specified after the keyword **TEXT**.
- the `button` subcommand adds a button gadget named **.bye**. The text on the button will be `'Goodbye'`. The keyword **OK** is a **form control attribute** that specifies that the

action of this button is to remove the form from the screen. (For more about form control attributes, see *Form Attributes*.)

To display the form in this example, you can use the command:

```
show !!Hello
```

13.3.1 Adding a Gadget Callback

To perform intelligent actions a form or gadget must invoke a callback We will now add a simple gadget callback to the hello form.

We will add a second button, **change message**, which will execute a callback to modify the **Message** paragraph gadget.

```
setup form !!hello
 paragraph .Message text 'Hello world'
 button .change 'Change message' callback |!this.message.val = 'Modified'|
 button .bye 'Goodbye' OK
exit
```

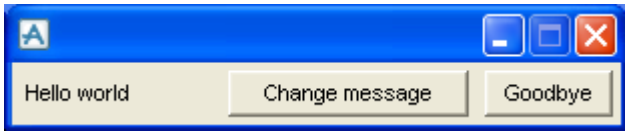


Figure 13:2. A Form with a Gadget Callback

A gadget callback is defined by the `callback` command followed by a command or PML Function enclosed in text delimiters. As we are giving a text string as part of the command which is itself supplied as a text string, we have had to use two kinds of delimiter: the apostrophe, (`'`) and the (`|`) vertical bar.

Note: The use of **This** to mean the current form. When the callback is executed,

```
!this.message.val = 'Modified'
```

will set the value member of the gadget **Message** on this form to read `Modified` rather than `Hello world`.

Note: The gadget callback could have been

```
|!!OtherForm.Para.val = 'Modified'|
```

to change the value of a paragraph gadget on another form named **!!OtherForm.**

Typically a callback will involve many commands and could be a complex piece of code in its own right. In practise, the recommended way of defining a complicated callback is to use a **form method**.

First though, we need to store our form definition in a file so that we can edit it at will.

13.4 Form Definition File

Form definitions must be held one per file. The file name must be the form's name in lowercase with the file extension `.pmlfrm`. For example, our form **!!Hello** would be captured in a file called `hello.pmlfrm`. This definition file should be stored in a directory pointed to by the `PMLLIB` environment variable. It will then be loaded automatically on execution of the `show !!Hello` command.

The form definition file contains:

- The form definition between `setup form` and `exit`. This includes the commands which create the form itself, and set its attributes, such as its size and title, the commands which create the gadgets on the form and specify how they are arranged, and the definitions of any variables which are to be members of the form.
- Any method definitions should follow the `exit` command, each method beginning with the `define method` command and ending with `endmethod`. Methods on forms are just like methods on any other kind of object - see *Methods on User-Defined Object Types*
- In particular, it will contain the form's **default constructor method**. This is a method with the same name as the form, and no arguments. It is the only method called automatically when the form is loaded, and so it can be used, among other things, to set default values for the gadgets on the form.
- The form may be given an **initialisation method**, which is run whenever the form is shown (as opposed to when it is loaded). See *Form Initialisation Callback*.
- No executable statements should appear in the file outside of the form definition or form methods. The effect of misplaced executable statements is indeterminate. You can put comments anywhere in the file.

13.5 How Forms are Loaded and Displayed

A form definition must be loaded before the form can be displayed. If you have stored the definition in a `.pmlfrm` file then loading will be automatic when the form is displayed for the first time. Normally, a form is displayed as a result of the operator making a menu selection or pressing a button on a form. This is achieved either by using the Form Directive in the menu or button definition (see next section) or by means of the command `show !!formname` used in the gadget's callback.

However, to display forms when you are developing them, you may find it convenient to type the command:

```
show !!formname
```

Sometimes it is useful to force the loading of a form's definition file before the form is actually displayed, so that you can edit the form or gadget attributes from another form's callbacks before the form is actually displayed. Using the command

```
loadform !!formname
```

from a callback will force load the definition if the form is unknown to PML, but do nothing if the form has already been loaded.

Once a form has been displayed you can remove it from the screen using the command

```
hide !!formname
```

Note that if you show it again it will appear on the screen, but its definition is already known to PML and so it will not be loaded.

It is possible to remove a form definition from PML using the command

```
kill !!formname
```

The form is then no longer known to PML until a new definition is loaded.

Note: Earlier AppWare used form definitions in macro files which had to be loaded explicitly via `$M path-name-to-file`. This mechanism still operates for backwards compatibility, but is strongly discouraged when writing new AppWare.

13.6 PML Directives

PML directives are commands used to control PML itself.

For example, you use a PML directive to instruct PML to re-make its index when you have added a new file. Some of these directives have been described in *Storing and Loading PML Files*: the information is repeated here, with additional directives for loading forms.

Note: Unlike the PML commands described in *How Forms are Loaded and Displayed*, PML directives should not be included in callbacks, but are generally for command line use.

You will need to use PML directives when you are developing new form definitions or modifying existing ones. PML directives are commands of the form `pml . . .`

The table below gives some useful PML directives.

Command	Effect
`pml rehash`	When you create a new PML Form while an AVEVA product is running, you must link in the file storing the form by giving this command. It causes PML to scan all the directories under the `PMLLIB` path, and to create a file `pml.index`, which contains a list of all the `.pmlfrm` files in the directories.
`pml index`	This command re-reads all the `pml.index` files in your search path without rebuilding them. If other users have added PML Files and updated the appropriate `pml.index` files, you can access the new files by giving this command.
`pml reload form !!formname`	When you edit an existing form while an AVEVA product is running, you must use this directive to reload the form definition file.

kill !!formname	If you experience problems of an edited form definition not being re-loaded, you can use this directive followed by the `pml reload` directive.
pmlscan directory_name	This command runs a utility supplied with AVEVA products. When you are not running an AVEVA product, you can use this command to update the `pml.index` file in a given directory. See the *Installation Guides* for more information.

See *Developing PML Code* for tracing commands that will be useful when you start to develop PML code.

13.7 Revisiting our Simple Form

Our simple form **!!Hello**, which we constructed earlier in this chapter, is not very intelligent. Once we have pressed the **Change** button the **.Message** paragraph will read 'Modified' for ever more, even if we hide the form and re-show it.

The extended version of form **!!Hello** below:

- illustrates the use of the form definition file;
- illustrates form methods as callbacks;
- introduces some important predefined form members.

First save the definition in a file called `hello.pmlfrm` and ensure that its directory is in your `PMLLIB` search-path.

```
setup form !!hello
 title 'Display Your Message'
 paragraph .Message width 15 height 1
 text .capture 'Enter message' width 15 is STRING
 button .bye 'Goodbye' OK
exit
define method .hello()
 -- default constructor - set gadget load-time default values
 !this.message.val = 'Hello world'
 !this.capture.callback = '!this.message.val = !this.capture.val'
 !this.Okcall = '!this.success()'
endmethod
define method .success()
```

```
    -- action when OK button is pressed
    !this.message.val = 'Hello again'
    !this.capture.val = ''
endmethod
```

Figure 13:3. A Smarter Form

In the form definition:

- `title` sets the **formtitle** member and hence displays a title;
- `para` adds a **PARAGRAPH** gadget size 15 chars by 1 line with no content;
- `text` adds a **TEXT** field gadget with tag `'Enter message'`, width 15 chars, to hold data of type **STRING**.

The constructor method `.hello()` does the following:

- initialises the paragraph's default value to `'Hello world'`;
- defines the callback on the text input field: to insert its value into the paragraph;
- sets the form member `Okcall` (in the line beginning `!this.Okcall`). This is a callback that gets executed when a button with **control-type** `OK` is pressed.

The definition of method `.success()` does the following:

- Sets the paragraph's value to `'Hello again'`.
- Resets the text field's value to empty.

Now load and show the form by typing:

```
PML rehash
show !!Hello
```

This will auto-load the form definition and execute the default constructor method (which will set the message paragraph to `'Hello world'` and define the gadget and form callbacks). It will also display the form.

Type your message into the **Enter message** field and press the **Enter** key. This will execute the field's callback, which will write your typed text into the message paragraph.

Type a new message and press **Enter**. The paragraph updates to reflect what you typed.

Click the **Goodbye** button. This executes the form's `Okcall` action which calls the `success()` method. The `success()` method sets the paragraph to `'Hello again'` and blanks out the text field. Finally, the `OK` control action hides the form.

Show the form again and observe that the paragraph reads `'Hello again'` and not `'Hello world'`. This demonstrates that when you re-show the form the form's constructor is ***not*** run, because the form is already loaded.

If you want to reset the form every time it is shown, you must define a form initialisation callback — see *Form Initialisation Callback*.

14 Form and Gadget Callbacks

Note: This chapter looks at the callback in more detail. If you are reading this manual for the first time then you may want to skip this chapter and return to it later.

Before you read this chapter you should have read and understood the *Form Concepts* chapter, in which we saw that callbacks are user-defined actions which are assigned to a form and its gadgets

They are supplied as text strings and may be any valid PML expression, PML Function or method call, including any form, gadget or user-defined object method.

14.1 Callbacks: Expressions

A callback may be any valid expression, including any AVEVA product commands. For example, the following is a PDMS command:

```
'new box xlen 10 ylen 20 zlen 50'
```

It can also include PML general commands like

```
'$m %pathname%/MyMacro'
```

to execute a given command macro, or

```
'q var !!form'
```

```
'q var !!form.gadget'
```

which will write out details of the form or gadget and its members. You might like to try it on the form **!!Hello**.

Typical expressions for a callback are

```
'!this.gadget.val = !MyObject.count'
```

```
'!MyObject.count = !this.gadget.val'
```

which get or set the value of a gadget on the form from or to a user-defined object.

14.2 Callbacks: Form Methods / PML Functions

Note: Before you read this section, make sure that you understand user-defined methods, as described in *Methods on User-Defined Object Types*.

Most callbacks require more than a single command, so invoking a method or function (or macro) is an essential requirement.

The advantage of using form methods as callbacks is that this keeps the whole form definition in a single file. Forms defined in early versions of PML 2 used PML Functions as callbacks. This is still valid and is sometimes even essential as you may need to manage a group of forms; but mostly the callbacks on a form are specific to that form.

You should note that a method or function invocation is just a valid PML expression, but such a powerful expression that it is worthy of its own section in the manual.

We have already used a form method as a callback in the revised **!Hello** form in *Form Concepts* chapter:

```
setup form !!hello
 title 'Display Your Message'
 paragraph .Message width 15 height 1
 text .capture 'Enter message' width 15 is STRING
 button .bye 'Goodbye' OK
exit
define method .hello()
 --default constructor - set gadget default values
 !this.message.val = 'Hello world'
 !this.capture.callback = '!this.message.val = !this.capture.val'
 !this.Okcall = '!this.success()'
endmethod
define method .success()
 !this.capture.val = ''
endmethod
```

The `.success()` method above could only deliver a fixed string `'Hello again'` to the **message PARAGRAPH** gadget. The great advantage of methods is that you can pass variables as arguments to the method, so it can be used more generally, for example as the callback to several gadgets.

```
define method .success( !output is GADGET, !message is STRING, !input
is GADGET )
 output.val = !message
 input.val = ''
endmethod
```

We have added three arguments, an output gadget, an input gadget and a message string variable. This has made the method very general. We can still use it as the **Okcall** callback:

```
!this.Okcall = |!this.success( !this.message, 'Hello again',
!this.capture )|
```

When the **OK** button is pressed the **Okcall** action will invoke the `success()` method, passing to it the **message** paragraph gadget as **!output**, the message text `'Hello again'` as **!message** and the text input field gadget **capture** as **!input**. The method will do just what it did before.

However, we could use it differently. We could add another button gadget, **Restart**, to the form:

```
button .restore 'Restart' callback |!this.success( !this.message,
'Hello world', !this.capture )|
```

When clicked, the **Restart** button's callback will execute and set the **message** paragraph gadget to read 'Hello world' and clear the **capture** text field, thus restoring the form to its state when it was displayed for the very first time.

If we invoked the success() method as:

```
!this.success( !this.capture, 'Hello world', !this.message )
```

it would set the value 'Hello world' into the **capture** text input field and clear the contents of the **message PARAGRAPH** gadget. Not what you need here perhaps, but you can see how versatile methods can be!

Note: The arguments to methods can be any valid PML object types, built in or user defined.

14.3 PML Open Callbacks

14.3.1 Events

When the operator interacts with a GUI, an event occurs. For example, when the operator:

- types something into a field on a form;
- moves the cursor into a window;
- presses down a mouse button;
- moves the mouse with button down;
- lets the button up.

  The events are queued in a time-ordered queue. The application software services this queue: it gets the next event, determines the object of the event (for example, form, gadget, menu) and the event type (for example, enter, leave, select, unselect, popup etc), deduces appropriate actions and carries them out. When one event is completed, the software looks for the next event.

14.3.2 Open Callbacks at Meta-events

There are a very large number of possible events, and most of them are very low level and have to be serviced very quickly to produce a usable GUI. It would be inappropriate to allow the (interpreted) PML AppWare access to them all.

However, the application software defines a set of **meta-events** for forms and gadgets. When a meta-event occurs, the application software checks for user-defined callbacks and executes them. Hence callbacks are the AppWare's way of providing actions to be carried out at these meta-events.

Callbacks provide a simple yet versatile mechanism for the AppWare to create and manage the GUI. Sometimes there is more than one meta-event associated with a gadget. In this case, the simple assigned callback is insufficient to fully exploit the gadget's possible behaviours. To overcome this shortcoming we can use **open callbacks** to allow the AppWare to be informed whenever a meta-event is encountered.

Open callbacks can be used wherever callbacks can be used. They always involve methods or PML Functions with a fixed argument list as follows:

```
define method .Control( !object is FormsAndMenusObject, !action is STRING)
```

- **!object** is a **Forms and Menus** object, for example, a form, gadget or menu.

- `!action` is the meta-event that occurred on the object and represents the action to be carried out by the method.

The open callback is a string of the form:

```
'!this.MethodName('
```

Note: The open bracket '(', no arguments and no closing bracket. The callback is to an ***open*** method or function.

We might assign an open callback to a multi-choice list gadget as follows:

```
setup form !!Open
 title 'Test Open Callbacks'
 list .choose  callback '!this.control('  multi width 15 height 8
exit
define method .open()
 do !i from 1 to 10
  !fields[!i] = 'list field $!i'
 enddo
 this.choose.dtext = !fields
endmethod
define method .Control( !object is GADGET, !action is
STRING)
 if ( !action eq 'SELECT' ) then
  --find out all about our gadget object
  !form = !object.owner()                    $*get object's owner
  !type = !object.type()                     $*get object type
  !name = !object.name()                     $*get object name
  !field = !object.PickedField               $*get picked field
                                             number
  !s = !object.DTEXT[!field]                 $*get DTEXT
  -- do something with the data
  $p selected $!form$n.$!name $!type field $!field, Dtext{$!s}
 elseif (!action eq 'UNSELECT' ) then
  !n = !object.PickedField                   $*get picked field
                                             number
   $p unselected field $!n                   $*do something with
                                             data
```

```
endif

endmethod
```

- Note in the constructor method `open()` we have initialised the list so that field **n** will display list field **n**. **DTEXT** is a shorthand for display-text, that is the text displayed in the list field.
- Control is the method which manages interaction with the list. Note the open callback defined in list choose.
- Note the use of `$*` for in-line comments
- Note the use of the printing command `$p` to print information to the system Request channel. `$!form` replaces the variable `!form` with its current value as a string - this only works for PML simple in-built scalars types **REAL**, **STRING**, **BOOLEAN**. `$n` in `$!form$n.$!name` is a separator needed to separate `$!form` from the following '`.`' which would otherwise be thought of as part of the variable name.

  When a list field is clicked, the list's callback will be examined and if it is an open method or function, then the Forms and Menus software will supply the arguments required to complete the function.

  Thus in this case the actual callback string executed will be

```
|!this.control( !!Open.choose, 'SELECT' )|
```

Inside the `control()` method we branch on the value of **!action** and access the data of **!object** (in this case our list). Finally we perform our application's resultant action - in this case just printing out all we know about the operator interaction using the `$p` command, which for a selection of an un-highlighted list field will write:

```
Selected OPEN.CHOOSE LIST field 5, Dtext{list field 5}
```

Notice that the same method could be used to manage several or even all gadgets on our form since the variable **!object** is a reference to a gadget object and is supplied as **!!Open.choose**, the full name of the gadget including its form. This allows us to find everything about that object from its in-built methods, including its gadget type (**LIST**, **TOGGLE**, etc) and its owning form:

```
!type = !object.type()

!form = !object.owner()
```

All the in-built members and methods of forms and gadget types are listed in the *Software Customisation Reference Manual.*

14.3.3 Using a PML Function in an Open Callback

All the examples so far have used form methods as open callbacks. The code would be essentially the same if we used PML Functions. The PML Function must be in a file of its own called `control.pmlfnc`. The body of the definition would be identical but must be bracketed by:

```
define function !!Control( !object is GADGET, !action is
STRING)

  . . .

endfunction
```

Note that the function has a global name `!!control`, that is, it is not a member of any form or object and thus cannot use the **!this** variable.

The open callback on the `choose` list gadget would become

```
list .choose callback '!!control(' multi width 15 height 8
```

The rest of the story remains the same.

14.3.4 Objects That Can Have Open Callbacks

Object	Object Classification	Callback
LIST multichoice	Gadget	**SELECT, UNSELECT, START, STOP**
LIST singlechoice	Gadget	**SELECT, UNSELECT**
OPTION	Gadget	**SELECT, UNSELECT**
ALPHA VIEW	Gadget	**SELECT**
BUTTON	Gadget	**SELECT, UNSELECT**
TOGGLE and **RTOGGLE**	Gadget	**SELECT, UNSELECT**
MENU (command fields)	Menu	**SELECT, INIT**
MENU (toggle fields)	Menu	**SELECT, UNSELECT, INIT**
FORM	Form	**INIT, QUIT, CANCEL, OK, KILLING, FIRSTSHOWN**
TEXT	Gadget	**SELECT, MODIFIED, VALIDATE**
SLIDER	Gadget	**START, STOP, MOVE**
FRAME	Gadget	**SELECT, UNSELECT, SHOWN, HIDDEN**
NUMERICINPUT	Gadget	**SELECT, MODIFIED**
COMBOBOX	Gadget	**SELECT, UNSELECT, VALIDATE**

Note: The **2D** and **3D VIEW** objects supplied in AVEVA products, in particular those in which **Event-Driven Graphics** are used, may have open PML Functions as callbacks. However, open callbacks on these gadgets are not supported as a user-definable facility at this release.

14.4 Undo/Redo Support for Callbacks

PDMS Undo/Redo is supported by certain callback actions for FORM, MENU and GADGET objects. Appware can decide, for any callback action, whether it is 'undoable', that is, supports the Undo/Redo buttons on the main toolbar. If so then a PDMS UNDOABLE object must be created and populated with the required Undo and Redo action strings, and then assigned to the appropriate F&M object callback.

For example:

In the form definition extract below:

```
Setup form !!myform . . .
    Title |MyForm with undoable callbacks|
    -- create form members as UNDOABLE objects for each F&M
    callback action,
    -- which is to be undoable, e.g. form OK callback, gadget
    callbacks etc.
    member !okUndo   is UNDOABLE
    member !doitUndo is UNDOABLE
    . . .
    button .doit  |Do it|  …
    . . .
    button .ok | OK |  … OK
exit
define method .myform()
    -- Constructor
    . . .
    -- Set up callbacks and assign any undo/redo actions to
    their undoable objects
    !this.okCall = !this.formOK()
    !this.okUndo.description(    'MyForm    OKcall    undo/redo
    actions' )
    !this.okUndo.undoAction( ' ' )
    !this.okUndo.redoAction( ' ' )
    -- assign undoable to form OK callback
    !this.setUndoable( 'OKcall', !okUndo )
    --
    !this.doit.callback = '!this.gadgetActions('
    !this.doitUndo.description( 'MyForm doit button undo/redo
    actions' )
    !this.doitUndo.undoAction( '!this.undoitAction( )' )
    !this.doitUndo.redoAction( 'this.doitAction( )' )
    -- assign undoable to do-it button callback
    !this.doit.setUndoable( !this.doitUndo )
    . . .
endmethod
define method .formOK()
    -- OKcall callback

    . . .
endmethod
define method .doitAction()
    -- doit button callback

    --  actions for doit/re-doit

    . . .
endmethod
```

```
define method .undoitAction( )
    -- undo method for doit button
    --  actions to undo the doit/re-doit action
    . . .
endmethod
```

Notes

1. **In most cases the Undoable object's undoAction and redoAction are not required!** This is because undo/redo is essentially linked to the state of the PDMS database rather than to the state of a form or its gadgets. Once a form is dismissed from the screen its state is not usually maintained, but is regenerated by the INIT callback next time it is displayed. Most forms which are sensitive to database changes, and remain displayed, are updated via their AUTOCALL callback (Tracker system), which again removes the need for explicit undo/redo support.
2. The Undoable object must be a form member (preferred) or a global variable or it cannot be held by F&M.

When a callback is executed, if an UNDOABLE object is present, then F&M modifies the undo/redo stack by means of the following actions:

```
undoable.Add( )
execute gadget callback
undoable.endUndoable( )
```

Methods to assign a PDMS UNDOABLE object to a callback, and to query a callback's UNDOABLE object are supported.

Not all callbacks are candidates for undo/redo. The following sections detail what is possible.

14.4.1 Form Callbacks

Forms currently support undo/redo for the following callbacks only:

Name	Purpose	Event/Action	Undo/Redo	Comments
INIT	Initialisation at show	INIT	yes	
OK	OK action	OK	yes	
CANCEL	Cancel action	CANCEL	yes	

Form methods:

```
!form.setUndoable( !callbackName is STRING, !undo is UNDOABLE )
!form.undoable( !callbackName is STRING ) is UNDOABLE
```

14.4.2 Gadget Callbacks

Gadgets support the following:

Name	Purpose	Events/Actions	Undo/Redo	Comments
CALLBACK	Standard gadget action	Gadget specific e.g. SELECT, UNSELECT, START, STOP	yes	
CALLBACK	View handler native actions	NATIVE	no	Supported by FEXVNC
CALLBACK	View handler interaction callbacks	Handler specific e.g. POPUP, ESCAPE	yes	Supported by FEXVCA

Gadget methods:

```
!gadget.setUndoable(!undo is UNDOABLE )
!gadget.undoable( ) is UNDOABLE
```

14.4.3 Menu and Menufield Callbacks

Menus and menu-fields support the following:

Callback name	Purpose	Events/Actions	Undo/Redo	Comments
CALLBACK	Initialisation at fill menu	INIT	no	
Menufield:				
CALLBACK	Field action	SELECT	yes	
CALLBACK	Toggle field action	SELECT		
UNSELECT	yes			

Menufield methods:

```
!menu.setUndoable( !menufieldName is STRING, !undo is UNDOABLE
)
!menu.undoable( !menufieldName is STRING ) is UNDOABLE
```

14.5 Core Managed Objects

AVEVA developers can define PML forms, menus, and gadgets that can be core-code controlled. See *Core Managed Objects*.

15 Forms

Before reading this section you should have read the Form Concepts chapter.

15.1 Modules and Applications

A PDMS GUI module consists of a set of co-operating applications running within the Application Window.

Each module has:

- A Main form to present the default application and to control access to the other applications.
- One or more document forms usually displaying the application model.
- Transient floating dialog forms that relate to specific tasks.
- A small number of docking dialog forms to provide frequently required services.

  The Main form has a **menu bar** that provides access to all the functionality of the current application of the module, from its pull-down menus (the *main* menu system), and a set of **tool bars**, each containing gadgets (usually icon buttons, icon toggles and pull-down lists) designed to give fast access to frequently used functions.

  The Main form is not displayed directly, but supplies all its menus and tool bars to be displayed in the Application Window.

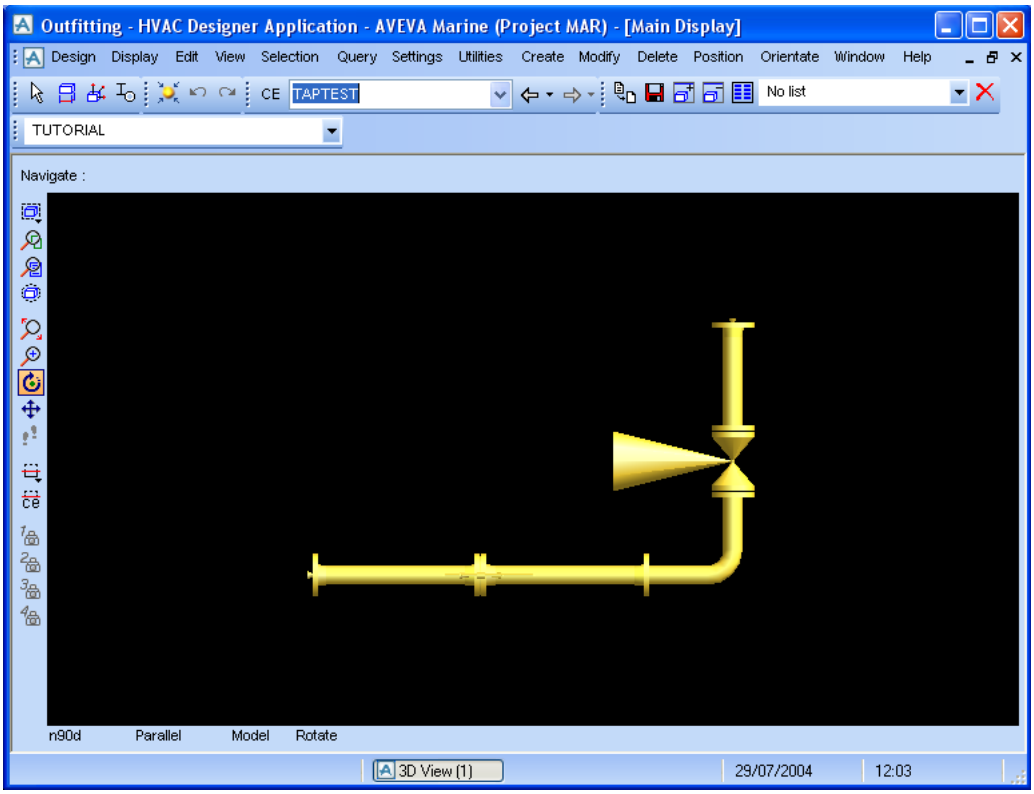
15.1.1 Application Window

The **Application Window(AppWin)** normally occupies most of the screen. When a Main form is assigned to it, the AppWin's components are replaced by corresponding components of the Main form.

From top to bottom the AppWin comprises:

- The AppWin Menu Bar, which contains the Main form's menu bar.
- The Dialog Bar, which contains the Main form's tool bars.
- The Document Area, which contains one or more document forms, which are normally constrained to this area.
- The Status Bar, which displays current status, messages and general information to the user.

The picture below shows a typical example of what the AppWin looks like:



15.1.2 Current Document

The AppWin supports the Windows™ Multi-Document Interface (MDI) to manage its forms of type Document. When a document is shown, or when the user selects a document from the menubar's Window menu or from the window management buttons on the status bar, the document becomes current, and may then be the target for actions selected from menubar, toolbars or from any docking dialogs currently displayed.

15.2 Defining a Form

The structure of a form definition file has already been discussed in *Form Definition File*. The form definition is a command sequence starting with:

```
setup form !!formname
```

and ending with:

```
exit
```

The sequence includes:

- The commands which create the form itself and set its attributes, such as its minimum size and title.
- The commands which create the gadgets on the form, and specify how they are arranged.
- The definitions of any variables which are to be members of the form.

15.2.1 Form Attributes

All the form attributes are optional and have sensible defaults. Those which can only be set once must be specified on the `setup form` line. These are:

- Form type.
- Minimum size.
- Resizability.
- Docking.
- Form position
- **NOQUIT**
- **NOALIGN**

  Other attributes are specified as sub-commands in the `setup form . . . exit` sequence. They can be edited after the form has been created by means of the form's in-built members and methods.

15.2.2 Form Type

The appearance and behaviour of a form is determined by its **Type** attribute:

Type of Form	Description
MAIN	The form that will be swapped to as a Main form. These forms are not usually displayed directly, but serve to provide gadgets for the application's toolbar and menus for the application's main menus.
DOCUMENT	Resizable form usually with a view gadget, but no menu bar. All document forms can be floated or un-floated using the right-mouse popup menu in the form's top border. When it is floating, you can drag the form away from the MDI frame and position it and resize it without constraint. This allows you to drag the document form away to another screen of a multi-screen configuration.
DIALOG	This is the default type the form will assume if you give no type when you set up the form. The default **DIALOG** form will be non-resizable, floating, and non-docking. You can specify the **DOCKING** attribute to allow the form to be docked within the application frame. By default, a docking dialog is displayed floating, and you can interactively dock it. When a dialog is docked it will be resized to match the application frame edge to which it is docked, and so is resizable by default. The qualifiers **LEFT**, **RIGHT**, **TOP**, and **BOTTOM**, specify the edge of the application frame to which the dialog form will be docked when first displayed.
BLOCKINGDIALOG	Normal form layout and content, but will block access to all other forms while it is displayed.

Here are some examples of ways you can set up forms of different types:

Setup Code	Description
`setup form !!myform dialog dock left`	Creates a **resizable docking dialog**;
`setup form !!myform dialog resizable`	Creates a **resizable floating dialog**;
`setup form !!myform dialog`	Creates a **non-resizable floating dialog**;
`setup form !!myform`	Creates a **non-resizable floating dialog**;
`setup form !!myform document`	Creates a **resizable MDI child document**;
`setup form !!myform document Float`	Creates a **floating resizable non-MDI document**.

15.2.3 Minimum Size and Resizability

A form will automatically stretch to fit the gadgets you add to it.

You can use the SIZE keyword to give a minimum size in multiples of the **character width** and **line height**. For example:

```
setup form !!New1 size 25.5 10
```

- **Character width** is the notional character width for the selected character font.
- **Line height** is the height of the tallest single line gadget, that is a **TOGGLE**, **BUTTON**, **RADIO BUTTON**, **OPTION** gadget or single-line **PARAGRAPH** for the selected character font.

The RESIZABLE command means that the form will be displayed with re-sizing controls at its corners. This will allow the user to change the size of the form. Docking forms and all document forms are resizable by default; but for other types, if you do not include the RESIZABLE command, the size of the form is fixed.

```
setup form !!New1 RESIZABLE

setup form !!New2 size 25 10 RESIZABLE
```

If you have a **View** gadget on the form it will resize itself with the form by moving its bottom right hand corner to maximum extent of the form.

15.2.4 Intelligent Resizable Forms

All gadgets except **ALPHA** and **VIEW** gadgets have **DOCK** and **ANCHOR** attributes that allow you to define gadgets that have intelligent positioning and resizing behaviour when their container gadget resizes.

This allows you to have more than one resizable gadget on a form and still have predictable and potentially complex resize behaviour. An example of an intelligent resizable form is given in *Complex Form Layout*.

However, the **DOCK** and **ANCHOR** attributes are mutually exclusive: setting the **DOCK** attribute resets the **ANCHOR** to the default; setting the **ANCHOR** attribute resets **DOCK** to none.

ALPHA and **VIEW** gadgets do not support **DOCK** or **ANCHOR** attributes. They do, however, expand to fill their containers, so you can put them in a frame and set the frame's **DOCK** or **ANCHOR** attributes to get the behaviour you desire.

15.2.5 Gadget Alignment Control

Certain gadgets, known as the ***linear*** gadgets, have their centres auto-aligned approximately to aid simple layout without the user having to know about **PATH** and **ALIGN** concepts.

This pseudo-alignment gives sensible layout for simple forms, but introduces small errors in all other circumstances and prevents accurate controlled layout because the user doesn't know the offsets applied. The **NOALIGN** keyword allows you to switch off this pseudo alignment to.

NOALIGN is now the recommended setting for forms of any complexity, and is explained in more detail in *Form Layout*.

15.3 Form Members

15.3.1 Form Title and Icon Title

The `title` sub-command is used to supply a string that is displayed in its banner at the top of the form's window. To set the title:

```
title 'Quite a Long Title for a Form'
```

You can modify the title at any time using the **FormTitle** member:

```
!!MyForm.FormTitle = 'Modified Title'
```

The `icontitle` sub-command is used to supply a string that is used when a form is iconised. To set the icon title:

```
icontitle 'Short Title'
```

You can also modify the icon title at any time using the **IconTitle** member:

```
!!MyForm.IconTitle = 'New Icon'
```

15.3.2 Form Initialisation Callback

The form's **initialisation callback** allows the form's gadgets to be initialised every time it is shown to reflect the current state of the application and possibly to validate whether the form can be displayed in the current context.

The callback will usually be a reference to a form method or possibly an open callback where several forms share a global initialisation function. See *PML Open Callbacks*.

You can set the callback by assigning to the form's **initcall** member. This can be done with the `INITCALL` command:

```
INITCALL '!This.InitCallBack()'
```

or directly by

```
!!MyForm.initcall = ‘This.InitCallBack()’
```

Note: The form initialisation callback must not attempt to display another form. You may invoke an **ALERT** object but not otherwise seek any input from the user.

If the callback discovers an error so serious that the form cannot be displayed it can abort the display of the form by returning an error. You can supply the text of an error message that is to be presented in an error alert in place of the form:

```
define method .initcallback()
 :
 return error 1 'You do not have write access to this database'
endmethod
```

If the initialisation callback has already caused an alert to be raised then you can prevent the raising of a new error alert by using the NOALERT keyword:

```
define method .initcallback()
 :
 return error 1 NOALERT
endmethod
```

15.3.3 Form OK and CANCEL Callbacks

The form supports the concepts of OK and CANCEL actions:

The OKCALL callback is executed when a form’s **OK** button is pressed or when the **OK** button of a form’s ancestor has been pressed (see *Free Forms and Form Families*).

It allows operators to approve the current gadget settings and carry out the function of the form. The form is then removed from the screen. Typically this callback will gather all the data from the form’s gadgets and perform the form’s major task. If you do anything of significance in callbacks on other gadgets you may have a hard time undoing everything if the user presses the **CANCEL** button.

You can assign to the form’s **OKCALL** member using the command

```
OKCALL ‘CallbackString’
```

You can modify the Okcallback at any time using

```
!this.Okcall = ‘CallbackString’
```

The CANCELCALL callback is executed when a form’s **CANCEL** button is pressed, when the **CANCEL** button of a form’s ancestor is pressed (see section on *Form Families*) or when the window’s **CLOSE** gadget is used. It allows the operator not to proceed with the functions of the form.

The form is then removed from the screen and all gadgets are automatically reset to the values they had when the form was displayed or when any **APPLY** button was last pressed. Typically this callback allows you, the PML programmer, to undo any actions already carried out on the form that ought to be undone in the light of the CANCEL request

You can assign to the form's **CANCELCALL** member using the command

```
CANCELCALL 'CallbackString'
```

You can modify the `CANCELcallback` at any time using

```
!this.Cancelcall = 'CallbackString'
```

See also *Button Gadgets*.

15.3.4 Quit/Close Callback

All Forms have a `QUITCALL` member that you can pass a standard callback string. This is executed whenever the user presses the **Quit/Close** icon (X) on the title bar of forms and the main application window.

If an open callback is used then it is called with the **FORM** object as its first parameter and '`QUIT`' as its action string.

QUITCALL for MAIN Forms

For forms of type **MAIN**, the `QUITCALL` callback is executed, if present. This permits the user to terminate the application, and so the associated PML callback should prompt the user for confirmation.

If the user confirms the quit, then the callback should close down the application, and ***not*** return. If the user decides not to quit, then the callback should return an error to indicate the decision to F&M.

Use `return error…noalert` if you want to avoid displaying an error alert. If the form has no `QUIT` callback, then the **QUIT** event will be ignored.

The following example shows a (global) PML function, that you could be use from all forms of type **MAIN**:

```
define function !!quitMain( )
 -- Sharable method Quit the application
 !str = !!Alert.Confirm('Are you sure you want to quit the
application?')
 if( !str eq 'YES' ) then
  -- execute application termination command, which should
not return
  finish
 else
  return error 3 |user chose not to QUIT| noalert
 endif
endfunction
```

This would be called from the form definition function body or from its constructor method as shown below:

```
Setup form !!myApplication MAIN
 . . .
```

```
 quitCall '!!quitMain( )'
 . . .
exit
define method .myApplication( )
 -- Constructor
 !this.quitCall = '!!quitMain( )'
 . . .
endmethod
```

QUITCALL for Other Forms

Essentially, if no QUIT callback is present, then the form is cancelled (hidden with reset of gadget values). If a QUIT callback is provided then you can prevent the default Cancel action by returning a PML error, but you must hide the form from your callback method (It is more efficient the use '!this.hide()', rather than 'hide !!myform' from your form methods).

Note: F&M does not display an alert for the returned error, it is merely for communication. You don't need a `QUIT` callback if you just want to allow the form to be hidden. For **DOCUMENT** forms (MDI children) only, the callback ***must not display an alert*** as this will cause some gadgets to malfunction afterwards.

15.3.5 FIRSTSHOWN callback

Typically assigned in the Constructor by

!this.FirstShownCall = '!this.<form_method>'

The purpose is to allow the user to carry out any form actions which can only be completed when the form is actually displayed. There are a variety of circumstances where this arises and it is often difficult to find a reliable solution. A couple of examples are given below.

Commands which manipulate form, menu or gadget visual properties, executed from a PML macro, function or callback may not happen until control is returned to the window manager's event loop. For example, in the application's start-up macro the command sequence show !!myForm … hide !!myform will result in the form not being displayed, but also not becoming known at all to the window manager. Attempts to communicate with this form via the External callback mechanism (possibly from another process) will not work. This can be rectified by doing the '!this.hide()' within the FIRSTSHOWN callback, because the form will be guaranteed to be actually displayed (and hence known to the window manager), before it is hidden.

It is sometimes difficult to achieve the correct gadget background colour setting the first time the form is displayed. This can be resolved by setting the required colour in the FIRSTSHOWN callback.

15.3.6 KILLING callback

Typically assigned in the Constructor by

!this.KillingCall = '!this.<form_method>'

The purpose is to notify the form that it is being destroyed and allow the assigned callback method to destroy any associated resources, e.g. global PML objects which would

otherwise not be destroyed. This may be necessary because PML global objects will survive an application module switch, but may not be valid in the new module.

Notes:

1. The callback method **MUST NOT** carry out any modifications to the Gadgets belonging to the form or to the Form itself (e.g. don't show or hide the form). Attempts to edit the form or its gadgets may cause unwanted side effects or possible system errors.
2. Form callbacks designed for other Form events (e.g. CANCEL, INIT) are rarely suitable as killing callbacks.
3. Restrict form and gadget operations to querying.

15.3.7 Form Variables: PML Variables within a Form

It is often convenient to store additional information on a form which will not be displayed to the user. This is achieved by defining **form variables**.

These are variables which can be any of the PML data types, including **ARRAYS** and **OBJECTS**. These variables have the same lifetime as the form and are deleted when the form itself is killed.

Form members are defined in just the same way as object members:

```
setup form !!MyForm...
 :
 member .MyNumber is REAL
 member .MyString is STRING
 member .MyArray is ARRAY
 :
exit
```

The value of a form member can be set and used in just the same way as an object member:

```
!!MyForm.MyNumber = 42
!Text = !This.MyNumber
!ThirdValue = !This.MyArray[3]
```

In a callback function you can use **!this.** to represent 'this form':

```
!ThirdValue = !This.MyArray[3]
```

15.3.8 Querying Form Members

You can query the members of a form using the command:

```
q var !!formname
```

This will list all the attributes of the form, and all the gadgets defined on it. This is a useful debugging aid.

To query all the gadgets of a form (excludes **USERDATA** gadget) use:

```
!!gadgetsarray = !!MyForm.gadgets()
```

Returns array of **GADGET**.

15.4 Loading, Showing, and Hiding Forms

15.4.1 Free Forms and Form Families

Forms are displayed on the screen either as **free-standing forms** or as a member of a **form family**.

A form can be displayed as a free standing form, for example by `show !!form free`. It then has no parent so it will not disappear when the form which caused it to be displayed is hidden.

When one form causes another form to be displayed, such as when a button with the **FORM** keyword is pressed or a gadget callback executes a `show !!form` command the result is a **child form**.

A form can have many child forms (and grand-children…) but a child form has only one parent - the form which caused the child form to be displayed. The nest of related forms is called a Form Family.

The Form Family exists just as long as the forms are displayed. If a form is already on the screen when it is shown, it is brought to the front of the display. If the child form is already in a Form Family it is transferred to the new parent.

If the user presses the **OK** button of a parent form, the system in effect presses the **OK** buttons on each of the child forms, 'youngest' first, invoking their `OKCALL` callbacks. The parent form and all child-forms are hidden and the Form Family then ceases to exist.

If the user presses the **CANCEL** button or uses the window's **CLOSE** controls, the system in effect presses the **CANCEL** buttons of each of the child forms, 'youngest' first, invoking their **CANCELALL** callbacks, and all the forms in the Form Family are hidden.

The action of **RESET** and **APPLY** buttons does not affect family members.

15.4.2 Loading and Showing Forms

A form definition must be loaded before the form can be displayed.

If you have saved the definition in a `.pmlfrm` file then loading will be automatic when the form is displayed for the first time.

Note: Earlier AppWare used form definitions in macro files which had to be loaded explicitly via `$m path-name`. This mechanism still operates for backwards compatibility, but is strongly discouraged when writing new AppWare.

Normally, a form is displayed as a result of the operator making a menu selection or pressing a button on a form. This is achieved either by using the `Form` directive in the menu or button definition or by means of the command:

`show !!formname`

used in the gadget's callback. In either case the form becomes a child of the menu's or gadget's owning form.

A form may be displayed free-standing, i.e. not as a child, by:

```
show !!formname free
```

Sometimes it is useful to force the loading of a form's definition file before the form is actually displayed, so that you can edit the form or gadget attributes from another form's callbacks before the form is actually displayed. Using the command:

```
loadform !!formname
```

from a callback will force load the definition if the form is unknown to PML, but do nothing if the form has already been loaded.

If you are sure that a form's definition has been loaded then you can show the form as a child or free-standing respectively using the form methods:

```
!!formname.show( )
!!formname.show( 'free' )
```

but note that this will ***not*** dynamically load the form definition.

15.4.3 Position of Forms on the Screen

Mostly forms are automatically positioned by the system according to their type and the way they are shown.

The **origin** of a form is its top left hand corner.

When a form is displayed as a child form then it is ***always*** positioned with respect to its parent.

For a form shown from a **MENU**, its origin is at the origin of the parent. If the form is displayed from a **BUTTON** or any other gadget, its origin is at the centre of the gadget.

When a form is shown as a **free** form for the first time then its default position is at the top left-hand corner of the screen.

We strongly recommend that you allow the system to position forms whenever possible.

You can force the screen position of free-standing forms using the following commands or methods:

```
show !!MyForm Free At xr 0.3 yr 0.5
show !!MyForm Free Centred xr 0.5 yr 0.5
!!MyForm.show( 'At', 0.3, 0.5 )
!!MyForm.show( 'Cen', 0.5, 0.5 )
```

The `At` option puts the origin of the form at the specified position; alternatively the `Cen` option puts the centre of the form at the given position. The co-ordinate values are fractions of the screen width or height respectively and are referred to as **screen co-ordinates**.

For example:

```
show !!MyForm free At xr 0.25 yr 0.1
```

positions the origin of **!!MyForm** one quarter of the way across from the left edge of the screen, and one tenth of the way down from the top.

```
!!MyForm.show( 'Cen', 0.5, 0.5 )
```

centres **!!MyForm** at the middle of the screen.

15.4.4 Hiding Forms

The recommended way for a form to be removed from the screen is for the user to press a button with the **OK** or **CANCEL** attribute.

Forms may also be cancelled by using the window's **close controls**. Both these mechanisms will remove the form and any of its children executing their `OK` or `CANCEL` callbacks appropriately.

Sometimes it is required to hide a form and other forms which are functionally associated but not part of the form family or as a result of a button press on a form you may want to hide other associated forms but not the form whose button was pressed. The `hide` command or method allows this:

```
hide !!MyForm

!!MyForm.hide()
```

Note: When you explicitly hide a form in this way its gadgets will be reset to their values at display or at the last `APPLY`, just like a `CANCEL` action, but the `CANCELCALL` callbacks for the form and its nest will not be applied. This means that before you execute the `hide` you should use any pertinent data from the forms to be hidden.

15.4.5 Killing Forms

You can destroy a loaded form definition using the command

```
kill !!MyForm
```

The form is hidden and then its definition is destroyed so that it is then no longer known to PML. You cannot then access the form or its gadgets, members or methods (including its `.show()` method.

Normally you will not need to include the kill command within your AppWare.

If you re-show the form using the `show` command then the system will attempt to demand load the definition again. This is slow and expensive if you haven't modified the definition so avoid it: use **loadform !!MyForm** and `!!MyForm.show()` instead.

If you execute a `setup form !!MyForm...` while the form **!!MyForm** already exists then it is killed and a new definition is started. This mechanism may be useful because it makes it possible to generate a form definition interactively from within your AppWare.

This can be very powerful but should be used with great care.

15.4.6 NOQUIT Form Status

You can stop forms from being hidden from the border close/quit pull-down menu by setting the **NOQUIT** attribute:

```
Setup form !!MyForm  . . .  NOQUIT
```

By default, you can quit from any user-defined form, except for the current system Main form.

15.5 CORE Managed Forms

Core managed forms are shown and hidden by core-code actions and cannot be 'killed' by PML AppWare. See *Form Core Support*

16 Menus

Menus are always members of forms but can be employed in various ways by the form and its gadgets.

Menus come in two types: main menus and popup menus. You determine what type a menu is when you create it. If you do not specify the type, then the system will try to infer its type from the context of its first use.

For example, if the first action in a form definition is to define the menubar, then the system will infer any menus referenced in the `bar…exit` sequence to be of type **MAIN**.

Or, as a second example, if a menu is assigned as the popup for the form or a gadget of the form, then the system will infer the menu to be of type **POPUP**.

16.1 Menu Types and Rules

Forms may have a bar menu gadget or main menu, which appears as a row of options across the top of the form. When you select one of the menu options, a **pull-down** menu is temporarily displayed. Fields on a menu may have pull-right arrows (>) that open a pull-down sub-menu when selected.

Forms and gadgets can have popup menus assigned to them. When you move the cursor onto them and press the mouse popup button, the menu pops-up at the cursor and you can then select from the displayed options.

The following rules determine how you can use menus:

- Each menu belongs ***either*** to the **Main menu system *or*** to the **Popup menu system**, but ***cannot*** belong to ***both***.
- A menu in the Main system can appear only once. i.e. it ***cannot*** be a sub-menu of several menus.
- A menu in the Popup system may appear ***only once*** in a given popup tree, but may be used in any number of popup trees.
- A menu cannot reference ***itself***, either directly as a pullright of one of its own fields or be a pullright of another menu in its own menu tree.
- Any pullright field of a menu references a sub-menu that will be inferred to be of the same type as the referencing menu.

16.1.1 Hints and Tips for Using Menu Types

In general:

- It is not necessary to specify the menu usage type on a form that has only main menus (and this includes most Main forms).
- It ***is always*** necessary to specify the usage-type for menus that are part of the **POPUP** menu system.

For forms that contain a mixture of main and popup menus:

- First define the bar before defining the menus.
- Next define the menus of the main system.
- Next define menus of the popup system, declaring them all as type `POPUP`.
- If you are dynamically creating new menus with the `NewMenu()` form methods, then always specify the menu type. This will maximise the system's chance of alerting you to any errors.

16.1.2 Core-Code Based Menus

AVEVA developers can define core managed PML menu fields. See *Form Core Support*.

16.2 Defining a Bar Menu Gadget

A bar menu is defined within a form definition. The menu bar is created with the `bar` subcommand. Note that its name is 'bar': there can be only one bar menu on a form. Then you can use the bar's `Add()` method to add the options. For example:

```
setup form !!MyForm Dialog size 30 5
 bar
 !this.bar.add ( 'Choose', 'Menu1')
 !this.bar.add ( 'Window', ' ' )
 !this.bar.add ( 'Help', ' ' )
exit
```

This code specifies the text of three options labelled **Choose**, **Window**, and **Help**.

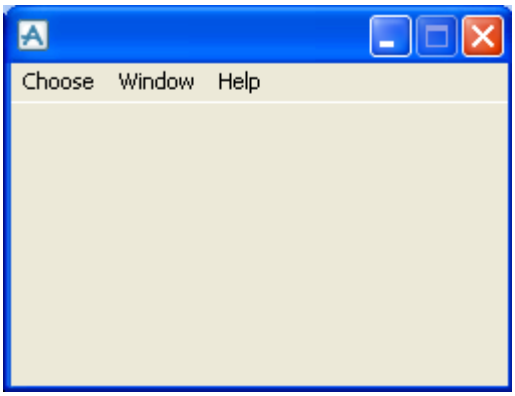


Figure 16:1. A Simple Menu

- The **Choose** option when picked will open **Menu1** as a pull-down (assuming you have defined **Menu1**).

Note: That **Menu1** need not exist when the bar is defined, it can be defined later in the form definition, but it must exist before the **Choose** option is selected or an error alert will be raised.

- The **Window** and **Help** options will open the Window and Help system menus (described later in this chapter).

16.2.1 Defining a Menu Object

A menu is a set of menu fields, each representing an action that is invoked when the field is selected. The fields' **display text** indicates to the user what the actions are, and the fields' **replacement text** defines what the actions are.

Within the form definition a menu object can be created using the form's `NewMenu` method or the `menu` sub-command. You can then use the menu's `Add()`, `InsertAfter()`, and `InsertBefore()` methods to add or insert named menu fields. A menu field can do one of three things:

- Execute a callback.
- Display a form.
- Display a sub-menu.

  You can also add a visual separator between fields.

Below is an example of a complete menu definition:

```
!menu = !this.newmenu( 'file', 'main' )
!menu.add( 'MENU', 'Send to', 'SendList', 'SendTo' )
!menu.add( 'SEPARATOR', 'saveGroup' )
!menu.add( 'CALLBACK', 'Save', '!this.SaveFile()', 'Save' )
!menu.add( 'FORM', 'Save as...', 'SaveFile', 'SaveAs' )
!menu.add( 'SEPARATOR' )
 --core-code managed field for Explorer Addin, ticked.
 --Note no Rtext needed
!menu.add( 'CORETOGGLE', 'Explorer', '', 'Expl' )
!menu.add( 'MENU', 'Pull-right1', 'Pull1')
 --initialise toggle field as ticked (typically in the
constructor)
!menu.SetField( 'Expl', 'Selected', true )
```

This creates a new main menu called **Menu** with six fields and two separators between them. For example:

- The **SAVE** field when picked will execute the callback command `this.SaveFile()`.
- The **Save as...** field when picked will load and display the form **!!SaveFile**. By convention, the text on a menu field leading to a form ends with **three dots**, which you must include with the text displayed for the field.
- The `SEPARATOR`, usually a line, will appear after the previous field.
- The **Pull-right1** field when picked will display the sub-menu **!this.Pull1** to its right. A menu field leading to a **sub-menu** ends with a **> symbol**: this is added automatically.

Named Menu Fields

You can add menu fields with an optional **fieldname** that you can later refer to when editing the menufield or modifying its attributes**'**. If you do not specify a field name then you will not be able to refer to the field again. You can also assign a name to separator fields, which allows separator group editing.

The general syntax is:

```
!menu.Add( '<FieldType>',' <Dtext>', '<Rtext>', { '<FieldName>' } )
!menu.Add( 'SEPARATOR', { '<FieldName>' })
```

Where the fields have the following meanings:

Field	Description
<FieldType>	has allowable values: **'CALLBACK'**, **'TOGGLE'**, **'MENU'**, and **'FORM'**.
<Dtext>	is the display-text for the field (cannot be null or blank). May contain multi-byte characters.
<Rtext>	is the replacement-text for the field. A null string indicates no replacement-text. The allowable values for **RTEXT** for the different field types are: **'CALLBACK'** - callback string **'TOGGLE'** - callback string **'MENU'** - menu name string (without preceding '.'). It cannot be blank. **'FORM'** - form name string (without preceding '!!'). It cannot be blank.
<FieldName>	is an optional argument, which, if present, is the unique field name within the menu.

16.2.2 Window Menu

You can add the system **Window** menu to a bar menu using:

```
!this.bar.add ('<Dtext>', 'window')
```

This menu is dynamically created on use with a list of the titles of the windows currently displayed as its fields. Selecting a field will pop that window to the front. This can be very useful on a cluttered screen.

16.2.3 Online Help Menu

You can add the system **Help** menu with the specified display text to a bar menu using

```
!this.bar.add ('Dtext', 'Help')
!this.bar.InsertAfter('window', '<Dtext>', 'Help')
```

When selected, this **Help** option displays a system-help pull-down menu that gives access to the application help system. The fields are:

Field	Description
Contents	This displays the **Help** window so that you can find the required topic from the hierarchical contents list.
Index	This displays the **Help** window with the **Index** tab selected, so that you can browse for the topic you want to read about from the alphabetically-arranged list. You can locate topics quickly by typing in the first few letters of their title.
Search	This displays the **Help** window with the **Search** tab at the front so that you can find all topics containing the keywords you specify.
About	To see the product version information.

You can access **On Window help** by pressing the **F1** key while the form has keyboard focus, or by including a **Help** button in the form's definition.

Note: By convention, the help menu should be the last one defined for the menu bar, which will ensure that it appears at the right-hand end of the menu bar.

16.2.4 Popup Menus

You can use any of your defined menus as popup menus for most interactive gadgets and for the form background as long as you have specified them as belonging to the popup menu system.

When the cursor is moved over it with the popup mouse button pressed down, and then released, the menu will be displayed, and you can select from it in the normal way.

A popup is added to a gadget or form using its `Setpopup()` method, with the popup menu as the argument to the method. Note that the menu pop1 must exist when the `Setpopup()` method is executed.

For example:

```
setup form !!MyForm resizable
 menu .pop1 popup
 !this.pop1.add( 'MENU', 'Options', 'optionmenu' )
 !this.pop1.add( 'MENU', 'More', 'moremenu' )
 !this.pop1.add( 'MENU', 'Last', 'Lastmenu'
 button .b1 ...
 . . .
 !this.b1.setpopup( !this.pop1 )
 . . .
exit
```

16.2.5 Finding Who Popped up a Menu

You can find out whether a menu was popped up from a gadget, and if so, the gadget's name. The method is:

```
!menu.popupGadget() is GADGET
```

If the menu was a popup on a gadget then the returned **GADGET** variable is a reference to the gadget. If the menu was popped up from a pulldown-menu or from a popup on the form itself, the value is **UNSET**.

Example:

```
!g = !menu.popupGadget()
if !g.set() then
 !n = !g.name()
 $p menu popped up by gadget $!n
else
 !n = menu.owner().name()
 $p menu popped up by form $!n
endif
```

16.2.6 Toggle Menus

A menu **TOGGLE** field is a menu field with a callback action and a tick-box to show that the field has been selected or unselected.

By default the field will be unselected so the box will not be ticked. When picked the fields callback action will be executed and the tick-box ticked.

If you pick the field again the callback action will again be executed and the tick removed.

Note that the open callback is an obvious candidate for toggle menus as the `SELECT` or `UNSELECT` action is returned as the second argument to the callback method. See *PML Functions and Methods*

For example, in your form definition you can add a toggle field as follows:

```
setup form !!Myform Dialog size 30 5
 . . .
 !menu = !this.newmenu('Test', 'popup')
 !menu.add(   'Toggle'   ,'Active/Inactive',   '!this.toggle(',
'OnOff' )
 . . .
exit
 . . .
define method .toggle( !menu IS MENU, !action IS STRING )
 !name = !menu.fullname()
 !field = !menu.PickedFieldName
 $P menu $!name $!action field: $!field
```

```
endmethod
```

Note: How we use the **PickedFieldName** member of the menu object to obtain the last picked field.

If you pick this menu field the callback method will print:

```
menu !!MyForm.Menu1 SELECT field: OnOff
```

16.3 Editing Bars and Menus

The contents of menu bars and menus can be modified at any time using the members and methods of the bar menu gadget object and the menu object.

16.3.1 Inserting Menus into a Bar

You can insert new fields into a menu bar using the `InsertBefore()` and `InsertAfter()` methods, which insert the new fields relative to existing named menus. The methods use named menus to determine the point where they should insert the new menu.

The general syntax is:

```
InsertBefore(<TargetMenuName>, <Dtext>, <MenuName>)
InsertAfter(<TargetMenuName>, <Dtext>, <MenuName>)
```

Where the fields have the following meanings:

Field	Description
<TargetMenuName>	is the name of the menu immediately before or after where you want the new menu to go.
<Dtext>	is the display-text for the menu.
<FieldName>	is the unique name for the menu within the bar.

For example:

```
setup form !!MyForm Dialog size 30 5
 bar
 -- adds a pulldown for menu1 labelled with <dtext>
 !this.bar.Add( '<dtext>', 'menu1' )
 -- adds a window pulldown labelled with <dtext>
 !this.bar.Add( '<dtext>', 'Window' )
 -- adds a help pulldown labelled with <dtext>
 !bar.InsertAfter( 'Window', '<dtext>', 'Help' )
 ...
exit
```

If you use the identifier 'Window' or 'Help' as the name of the menu, the system will interpret them as system **Window** and **Help** menus, although they will still be displayed with the string given in **<dtext>**.

Named menus and the methods that create them are discussed in more detail in the rest of this section.

16.3.2 Inserting New Menu Fields

You can insert new fields into a menu using the `InsertBefore()` and `InsertAfter()` methods, which insert the new fields relative to existing named menu fields.

The methods use named menu fields to determine the point where they should insert the new field.

The general syntax is:

```
InsertBefore(<TargetFieldName>,<FieldType>,<Dtext>,<Rtext>,{<FieldName>})
InsertBefore('SEPARATOR',{<FieldName>})
InsertAfter(<TargetFieldName>,<FieldType>,<Dtext>,<Rtext>,{<FieldName>})
InsertAfter('SEPARATOR',{<FieldName>})
```

Where the fields have the following meanings:

Field	Description
<TargetFieldName>	is the name of the field immediately before or after where you want the new field to go.
<FieldType>	has allowable values: **'CALLBACK'**, **'TOGGLE'**, **'MENU'**, and **'FORM'**.
<Dtext>	is the display-text for the field (cannot be null or blank). May contain multi-byte characters.
<Rtext>	is the replacement-text for the field. A null string indicates no replacement-text. The allowable values for **RTEXT** for the different field types are: **'CALLBACK'** - callback string '**TOGGLE'** - callback string **'MENU'** - menu name string (without preceding '.'). It cannot be blank. **'FORM'** - form name string (without preceding '!!'). It cannot be blank.
<FieldName>	is an optional argument, which, if present, is the unique field name within the menu.

16.3.3 Changing the State of Menufields

There are two methods you can use to set and read the status of menu- and menu-field properties.

Setting Menu-Options' Status

You can de-activate a menufield on a menu bar and a field in a menu with the `SetFieldProperty()` so that it cannot be selected. Similarly, you can make them invisible so the user cannot see them.

You can also use `SetFieldProperty()` to hide a menufield and to select or unselect toggle-type fields.

The general syntax is:

```
!menu.SetFieldProperty (<FieldName>, <PropertyName>, Boolean)
```

Where the fields have the following meanings:

Field	Description
<FieldName>	The name of the field you want to change.
<PropertyName>	The name of the property you want to change in the named field. The allowed values are: **'ACTIVE'** - greyed in or out **'VISIBLE'** - visible or invisible **'SELECTED'** - selected or unselected (toggle type fields, only. Specifically, this value cannot be used with bars).
Boolean	The value, **TRUE** or **FALSE**, for the property.

Note: The property names may optionally be truncated to the first three characters '**ACT'**, '**VIS'**, and '**SEL'**.

For example:

```
!bar = !!MyForm.bar
!menu = !!MyForm.Menu1
```

sets local variables **!bar** and **!menu** to be references to the bar gadget and **Menu1** of form **!!MyForm**.

Then

```
!bar.SetFieldProperty( 'Foo', 'ACTive', false)
```

will grey-out the menufield on **bar** that has the field-name "`Foo`". And

```
!menu.SetFieldProperty ( 'Bar', 'ACTive', true)
```

will activate the menufield on **Menu1** that has the field-name "`Bar`".

You can use the same method to change the selected status of a toggle menu field.

Reading the Status of Menus and Menus' Fields

To read the status of a menu or of a menu-field property, you can use the `FieldProperty()` method.

The general syntax is:

```
Boolean = !menu.FieldProperty (<FieldName>, <PropertyName>)
```

Where the fields have the following meanings:

Field	Description
<FieldName>	is the name of the field you want to change.
<PropertyName>	is the name of the property you want to change in the named field. The allowed values are: **'ACTIVE'** - greyed in or out **VISIBLE'** - visible or invisible **'SELECTED'** - selected or unselected (toggle type fields, only. Specifically, this value cannot be used with bars)

Note: The property names may optionally be truncated to the first three characters '**ACT'**, '**VIS'**, and '**SEL'**.

For example:

```
!bar = !!MyForm.bar
```

sets local variable **!bar** to be a reference to the bar gadget of form **!!MyForm**.

Then

```
!isSet = !bar.FieldProperty( 'Foo', 'ACT')
```

will get the greyed-out status of the menufield on **bar** that has the field-name "`Foo`".

You can use the same method to change the selected status of a toggle menu field.

16.3.4 Implied Menu-field Groups

A separator field and all following fields up to but not including the next separator field implies a **menu-field group**. You can modify the **ACTIVE** and **VISIBLE** properties of all fields in the group, by reference to its separator name.

For example, for the menu:

```
!menu = !this.newmenu( 'file', 'Main' )
!menu.add( 'MENU', 'Send to', 'SendList', 'SendTo' )
 !menu.add( 'SEPARATOR', 'SaveGroup' )
  !menu.add( 'CALLBACK', 'Save', '!this.SaveFile()', 'Save' )
 menu.add( 'FORM', 'Save as...', 'SaveFile', 'SaveAs' )
 !menu.add( 'SEPARATOR', 'explGroup' )
  !menu.add( 'FORM', 'Explorer...', 'ExplFile', 'Expl' )
```

Executing the method

```
!menu.SetField( 'saveGroup', 'visible', false )
```

will make the all the fields invisible for the group currently implied by the separator field '**SaveGroup'**, i.e. the fields **SaveGroup**, **Save** and **SaveAs**.

The combination of named **SEPARATOR** fields, insertion and field group visibility will be useful for managing the sharing of menus between co-operating sub-applications. This facility should be used with great care.

16.3.5 Creating Menus Dynamically

You can create new menus from within your appware dynamically using the form method `NewMenu()`.

For example, you could equip your form with the methods `popupCreate()` and `popupAction()` which would allow you create and service a popup menu from an array of strings.

Executing `!this.popupCreate('NewPopup', !fieldArray)` will create a new popup menu and assign it to the form.

```
define method .popupCreate( !name is STRING, !fields is ARRAY )
 --!fields is an array of field name strings
 !menu = !this.newmenu( !name, 'popup' )
 --add all the fields with same open callback
 do !n from 1 to !fields.size()
  !menu.add( 'Callback', !fields[!n], '!this.menuAction(' )
 enddo
 -- assign the new menu as the form's popup menu
 !this.setpopup( !menu )
endmethod
define method .popupAction( !menu is MENU, !action is STRING )
 -- General popup menu action routine
 if ( !action eq 'SELECT' ) then
  !name = !menu.fullname()
  !field = !menu.pickedField
  -- execute application actions according to the field
selected
  $P selected field $!field of menu $!name
 ...
 else
  -- execute applications for unselected field (toggle)
  ...
 endif
endmethod
```

17 Form Layout

Typically gadgets are laid out onto the form from left to right and from top to bottom. It is unnecessary and undesirable to specify absolute positions for gadgets; we recommend defining the layout of each gadget relative to a predecessor. This allows the layout to be edited simply, without having to calculate new positions.

17.1 Form Coordinate System

The form can be thought of as an imaginary **grid**:

- The origin (0,0) of the grid is at the top left-hand corner of the form.
- The horizontal pitch is the **notional character width** for the currently selected font.
- The vertical pitch is the **notional line-height**, which is the height of the tallest single line gadget for the currently selected font, i.e. the maximum height of a textual **TOGGLE**, **BUTTON**, **OPTION** or **TEXT** gadget.

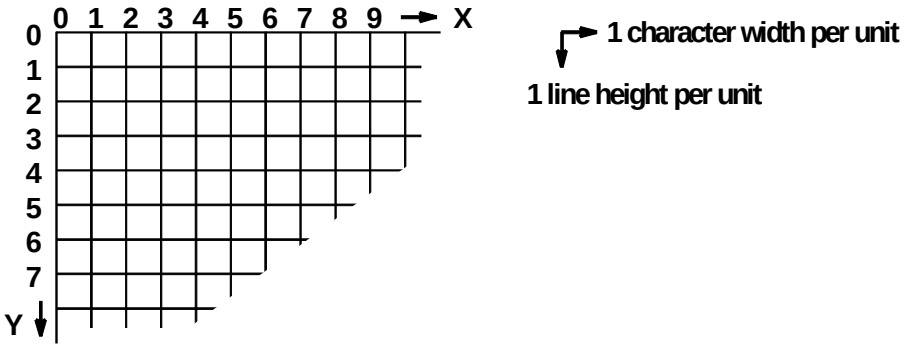


Figure 17:1. Conceptual X and Y Coordinates

At any time the form has maximum and minimum extremities **XMIN**, **YMIN**, **XMAX**, **YMAX**. As new gadgets are added to it the form **XMAX** and **YMAX** extents grow to include the gadget boxes.

Note: The grid width is the ***notional*** character width because fonts usually have proportional spacing so different characters in the font typically have different widths. Thus ***n*** grid units does not equate to ***n*** characters. Generally, you can get more than ***n*** characters in ***n*** grid units unless the string has a high proportion of wide characters e.g. capitals, numerals, w's, m's. It is important that you understand this concept when you specify the width and height of gadgets.

17.1.1 Form Setup NOALIGN Property

Forms have the notion of a set of ***linear*** gadgets, namely **BUTTON**, **TOGGLE**, **TEXT**, **OPTION** and single line **PARAGRAPH**. These gadgets fit within 1 vertical grid unit. By default, a form's linear gadgets are drawn (whatever the specified position, or current **PATH**), with their Y-coordinate adjusted so that they would approximately centre-align with an adjacent **BUTTON**.

This pseudo-alignment gives sensible layout for simple forms for **PATH RIGHT** (the default) and **PATH LEFT**, but introduces small errors in all other circumstances and prevents accurate controlled layout because the user doesn't know the offsets applied.

Also the system applies these offsets at form build-time, rather than at form definition, which can lead to gadgets overlapping other gadgets or the form edges very slightly.

The keyword **NOALIGN** is available in the form setup command and prevents this gadget auto-alignment, e.g.

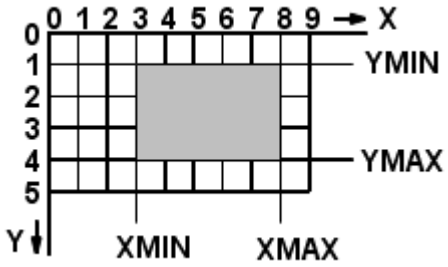
```
Setup Form !!myForm dialog ... noAlign
```

We recommend you use **NOALIGN** in conjunction with **PATH RIGHT** (the default path) and **HALIGN CENTRE**, as it gives a better layout, with fewer surprises.

17.1.2 Gadget Box

A gadget can be thought of as having an enclosing box with an origin at the top left hand corner. This box will enclose the geometry of the gadget, including its name-tag if specified.

The extremities of a gadget box on the form can be referenced by the gadget pseudo-variables:



In their definition gadgets may be positioned at specific form coordinates form, using the **AT** keyword followed by the X and Y grid coordinates.

17.2 Positioning Gadgets

It is recommended that you layout a form using one or both of the following schemes. Both allow the form design to have gadgets added, removed, and repositioned, without having to adjust the geometry of all other gadgets on the form.

Auto-placement uses the **PATH**, **DISTANCE** and **ALIGNMENT** keywords. These keywords have the following meaning:

Keyword	Description
PATH	is the direction in which the next gadget origin will be placed relative to the previous gadget.
DISTANCE	is the spacing between gadgets along the current path.
ALIGNMENT	specifies how the next gadget aligns to the previous one for the current path.

Relative placement means specifying the new gadget position by explicit reference to the **extremities** of a previous gadget.

For special cases it is also possible to position gadgets using explicit form coordinates. See *Absolute Gadget Positioning*.

17.3 Auto-placement

17.3.1 Positioning Gadgets on a Defined Path

To specify the **path** along which gadgets are to be added, use one of the following `PATH` commands:

```
PATH right
PATH left
PATH up
PATH down
```

The **default** direction is `PATH right`.

The current path direction is set until you give a different `PATH` command.

17.3.2 Setting the Distance Between Gadgets

To specify the horizontal and vertical **distance** between adjacent gadgets, use the `HDISTANCE` and `VDISTANCE` commands, respectively. For example:

```
HDIST 4
VDIST 2
```

where the **clearance distances** are specified in grid units.

The defaults are `HDIST 0.25, VDIST 1.0`

Note that these specify ***clearance distances*** between gadgets (for example, the distance between the **XMAX** of one gadget and the **XMIN** of the next), ***not*** the distance between gadget origins, thus:

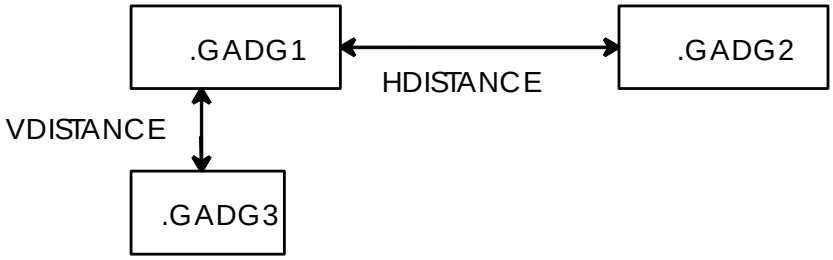


Figure 17:2. Distances Between Gadgets

17.3.3 Gadget Alignment

To specify horizontal and vertical **alignments**" of adjacent gadgets, use the `HALIGN` and `VALIGN` commands, respectively. The options are:

```
HALIGN left
HALIGN centre
HALIGN right
VALIGN top
VALIGN centre
VALIGN bottom
```

The defaults are `HALIGN left` and `VALIGN top`.

17.3.4 How It All Works

The precise position of a new gadget relative to the preceding one is determined by a combination of the current path, clearance distances and alignments.

For horizontal paths, **LEFT** and **RIGHT**, the **horizontal distance** and **vertical alignment** apply.

For vertical paths, **UP** and **DOWN**, the **vertical distance** and **horizontal alignment** apply.

The following diagram shows the effects of all possible combinations of path and alignment settings (with a fixed clearance distance, D, in each direction):

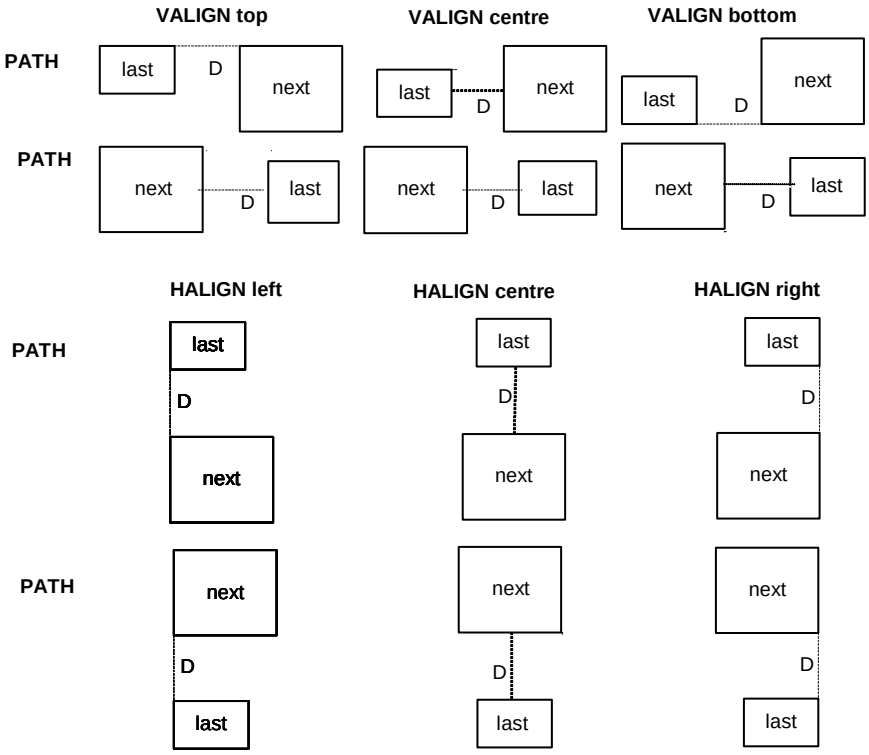


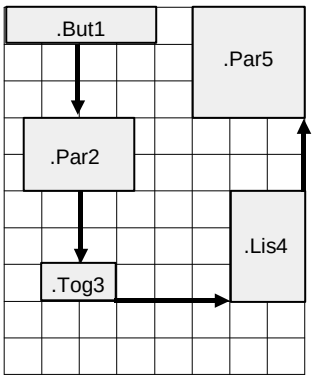
Figure 17:3. Effect of Path Alignment Settings

Combinations of path, distance and alignment commands allow you to set up correctly aligned rows and columns of gadgets without the need to calculate any grid coordinates.

For example, the command sequence:

```
button  .But1                              $* default placement
PATH down
HALIGN centre
VDIST 2
paragraph  .Par2  width 3 height 2         $* auto-placed
toggle  .Tog3                              $* auto-placed
PATH right
HDIST 3.0
VALIGN bottom
list  .Lis4 width 2 height 3               $* auto-placed
PATH up
HALIGN right
paragraph .Par5  width 3 height 3          $* auto-placed
```

gives the following layout of toggle gadgets:



The gadgets are created in numeric order with the arrow direction showing the current path when each new gadget was placed (down, down, right, up).

The arrow positions show the current alignment when each gadget was placed (centre, centre, bottom, right).

Figure 17:4. Example Layout of Gadgets

17.3.5 Positioning Gadgets and Frames

The default position for the origin of the first gadget on a form is at the form origin (0,0).

A **frame** is a gadget that can act as a container for gadgets, including other frames.

Frames themselves position just like simple gadgets in that a frame and other gadgets around it can be positioned relative to one another.

Inside a frame a gadget's position is relative to the top-left-hand-corner of the frame, rather than the form as whole.

Gadgets inside a frame can be positioned relative to each other.

The default positioning of gadgets within a frame depends on the current settings of **PATH**, **HDIST** and **VDIST**. The frame's tag name is positioned on the top border **HDIST** from the top-left-hand corner. Gadgets are positioned and the frame is sized so that the side of a gadget nearest to the side of a frame is **HDIST/2** away from it. Gadgets close to the top or bottom border are arranged so that they are **VDIST/2** away from it. The frame border itself is just a few pixels wide - certainly less than one character height or width.

17.4 Relative Placement

Relative Placement uses the AT keyword, followed by keywords and values that define the offsets from another gadget. The complete AT syntax is given in *AT Syntax*.

17.4.1 Positioning Relative to the Last Gadget

To specify the origin of a new gadget relative to an extremity of the last placed gadget, use the **AT** keyword as follows:

```
toggle .OnOff AT XMIN YMAX+1
```

Positions the origin of the new toggle gadget with respect to the extremities of the ***last-created*** gadget (indicated by the absence of an explicit gadget name in the AT command). The new gadget's origin is at the coordinates (**XMIN**, **YMAX+1**) of the last gadget:

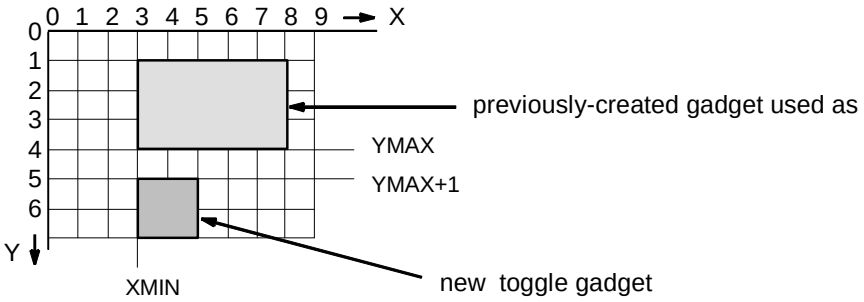


Figure 17:5. Positioning Relative to Specific Previous Gadgets

To position the origin of a new gadget relative to an extremity of **.Gadget1**, a specific previous gadget, use:

```
toggle .OnOff AT XMIN .Gadget1 YMAX .Gadget1+1
```

The effect is similar to that illustrated above.

```
toggle .OnOff AT XMIN .Gadget1-2 YMAX .Gadget1+1
```

Positions the new toggle to the left of **.Gadget1** (since the specified X offset is negative):

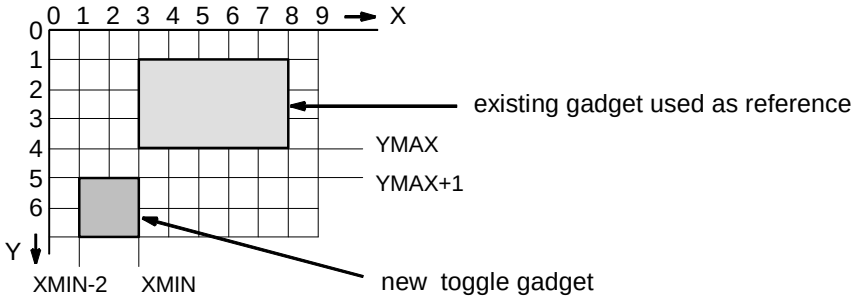


Figure 17:6. Newly Placed Gadget

Note: The new gadget need not be adjacent to the referenced gadget(s). The same rules apply even if there are other gadgets positioned between them. You will probably find form design easier, however, if you add gadgets in two directions only (right and down, say), rather than in a more random order

17.4.2 Positioning Relative to the Form Extremities

You can also position a new gadget relative to the ***current*** size of the form, and the ***current*** size of the gadget.

For example, if you want to place an **OK** button at the extreme right-hand bottom corner of the form, you could use the commands:

```
button .OK AT XMAX FORM-SIZE YMAX FORM OK
```

`XMAX FORM` and `YMAX FORM` refer to the maximum **X** and **Y** coordinates for the ***entire form so far***.

`XMAX FORM - SIZE` subtracts the current gadget's size so that the form will not expand. The net result of the above command is to add the **OK** button so that in the X-direction it just finishes at the form's current maximum extent without increasing the form width, and in the

Y-direction the gadget's origin is placed at the form's current maximum depth and extends it to include the gadget's height. The result is:

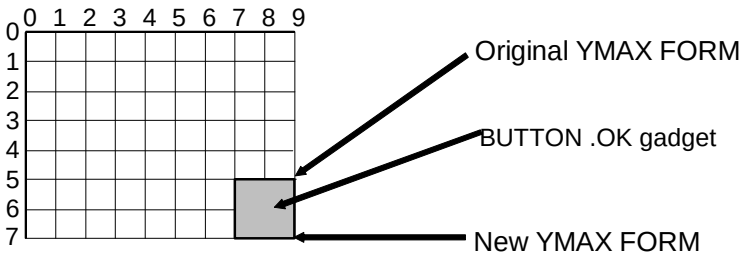


Figure 17:7. Positioning Relative to Form's Extremities

Typical code to add **CANCEL** and **OK** gadgets to a form would be:

```
button .CANCEL at xmin form ymax form CANCEL
button .OK at xmax form - size OK
```

17.5 Mixing Auto and Relative Placement

Note that each gadget coordinate is independent with respect to auto and explicit placement. All the following are legal constructs:

```
toggle .t1 AT XMIN.Gadget1 YMAX.Gadget2
```

places .**t1** with respect to **XMIN** of one gadget and **YMAX** of a different gadget.

```
toggle .t2 AT XMAX YMAX.Gadget2 + 0.25
```

places **.t2** with respect to **XMAX** of the last placed gadget and **YMAX** of a specific gadget **Gadget2**.

```
PATH down
. . .
toggle .t3 AT xmin.Gadget1
```

places .**t2** with respect to **XMIN** of gadget **.Gadget2**, whilst the **Y** coordinate for **.t2** is auto-placed at current **VDIST** value below the last placed gadget.

17.6 Absolute Gadget Positioning

Note: Absolute positioning is not the recommended way to define your forms: use relative positioning.

For example, any of the commands:

```
toggle .OnOff at 3  3.5
TOGGLE .ONOFF AT X 3  Y 3.5
TOGGLE !This.TOGGLENAME AT Y 3.5  X 3
```

positions a new toggle gadget with its origin at grid coordinates (3, 3.5), thus:

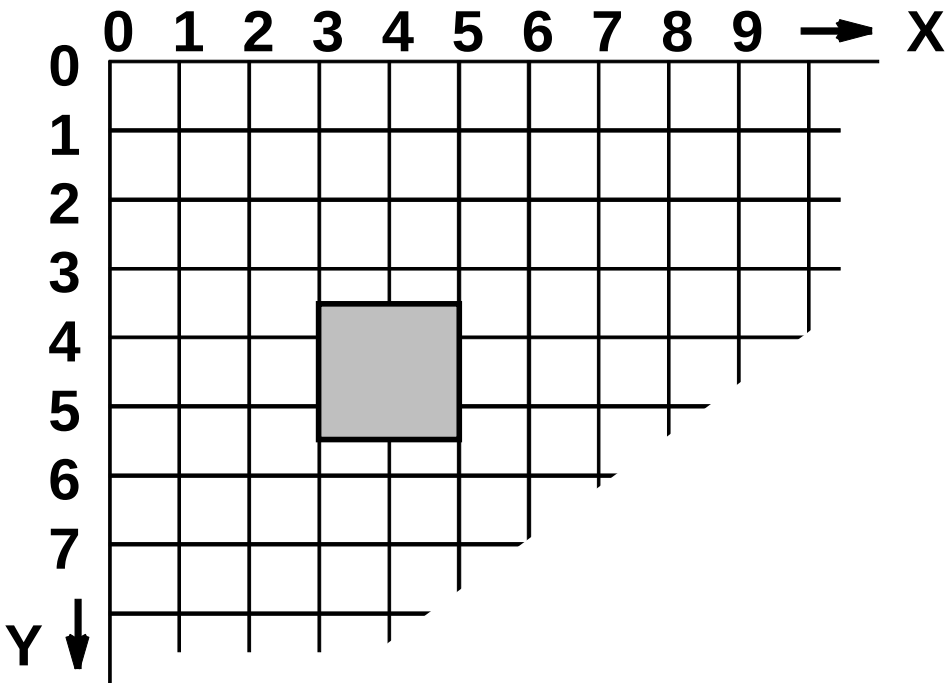


Figure 17:8. Positioning at a Known Location

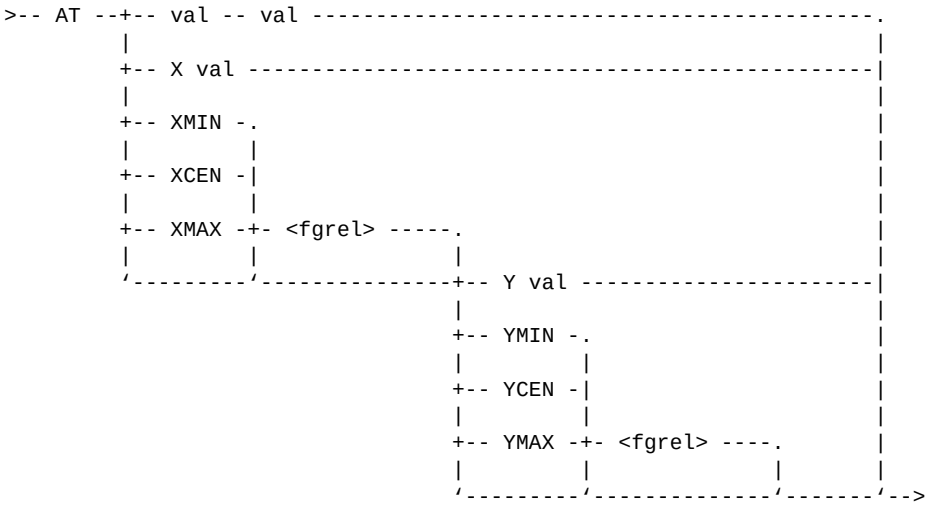
Note: You can position gadgets ***anywhere*** on the grid, not only at the grid intersection points.

17.7 AT Syntax

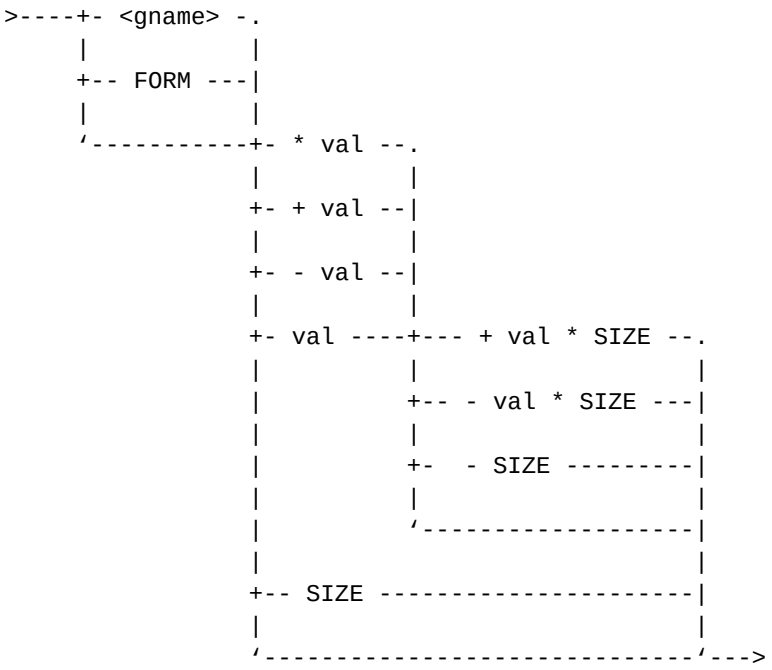
The **AT** syntax is used to define the position of a gadget's origin within a form. The position may be specified absolutely (in form layout grid units) or relative to the extremities of existing gadgets, or relative to the size of the form and the gadget.

The rest of the Forms and Menus syntax is described in ***Software Customisation Reference Manual***: refer to that manual for more information, in particular about conventions used in the syntax graphs. The **AT** syntax graph is the easiest way of illustrating all the possible options, and so it is included here, as well as in the *Software Customisation Reference Manual*.

Some examples follow the syntax graph.



where **<FGREL>**, shown below, sets the gadget position relative to another gadget or the form's extent. For example, it can be used to position a gadget half-way across the width of a form:



Below are some examples of how you can use the syntax:

Syntax	Effect
AT 5 7.5	Puts gadget origin at form grid coordinates (5, 7.5).
AT X 5.5	Puts gadget origin at form grid coordinates (5.5, y) where y is calculated automatically from the **y** extremity of the last placed gadget and the current VDISTANCE setting.
AT YMAX+1	Positions new gadget at (x, y) where x is calculated automatically from the **x** extremity of the last placed gadget and the current HDISTANCE setting, and y is at YMAX+1 of the last gadget.
AT XMIN.GAD1-2 YMAX.GAD2+1	Positions new gadget with respect to two existing gadgets. The gadget is offset by 2 grid units to the left of .GAD1(X=XMIN-2) and 1 unit below .GAD2 (Y=YMAX+1).
AT XMAX FORM-SIZE YMAX FORM-SIZE	XMAX FORM refers to the current right hand size of the form at its current stage of definition (not its final maximum extent). YMAX FORM refers to the form's current bottom extent. The -SIZE option subtracts the size of the gadget being positioned in the form. This example positions the gadget at the extreme right-hand bottom edge of the form.

17.8 Intelligent Positioning and Resizing

So far we have considered the static layout of the form. But often our forms need to be resized by the user at run-time.

Most gadgets also have **DOCK** and **ANCHOR** attributes that allow you to define intelligent position and resize behaviour when the gadget's own container-gadget resizes.

This allows you to have more than one resizable gadget on the same form and to have predictable and potentially complex resize behaviour.

The **DOCK** and **ANCHOR** attributes are mutually exclusive. Setting the **DOCK** attribute resets the **ANCHOR** to the default; setting the **ANCHOR** attribute resets **DOCK** to none.

You can set these attributes only when you define the gadget: you cannot change it after the exit from form setup. Thus you are not allowed to change the resize behaviour at run-time.

At present **ALPHA** and **VIEW** gadgets support neither the **DOCK** nor **ANCHOR** attributes. However, as these gadgets expand fully to fit their containers; you can place them in their own frame, with no other gadgets, and set the frame's **DOCK** or **ANCHOR** attributes to get the behaviour you desire.

17.8.1 The ANCHOR Attribute

The **ANCHOR** attribute allows you to control the position of an edge of the gadget relative to the corresponding edge of its container.

For example **ANCHOR RIGHT** specifies that the right hand edge of the gadget will maintain a fixed distance from the right hand edge of its owning container.

The **ANCHOR** attribute may have any combination of the values **LEFT**, **RIGHT**, **TOP**, or **BOTTOM**; or it may have the values **NONE** or **ALL**. When the container is resized, the fixed distance is maintained for any anchored edges.

17.8.2 The DOCK Attribute

The **DOCK** attribute allows you to dock a gadget to the left, right, top, or bottom edge of its container, typically a form or a frame; or you can cause the gadget to dock to all edges, or to no edges.

When the gadget's container is resized, the docked gadget's edges will remain 'stuck' to the corresponding container edge or edges. Any docked edge will be resized to be the same size as its container's edge.

18 Frames

Frames are very special and versatile gadgets that are crucial for form definition, layout, and run-time behaviour. This section gives an overview of frame gadgets. Their definition and details are discussed in *Gadgets and their Attributes*.

18.1 Types of Frame

There are five different types of frame, and each type has different constraints on what gadgets it can contain and what it can be used for.

The five types of frame are **normal**, **tabset**, **toolbar, Panel** and **Foldup Panel**.

18.1.1 Normal Frames

A frame of type **NORMAL** is a container with a visible border that can contain any type of gadget, including other frames. This simple frame displays its tag as a title to describe the gadget group as a whole.

Normal frames can be **nested** and as an inner frame expands, an outer frame will also expand to accommodate it.

The FRAME has a Radio Group property which operates on the set of RTOGGLE gadgets (radio buttons) owned directly by it.

18.1.2 Tabset Frames

The **TABSET** frame type defines a container for a set of tabbed page frames. The container has no visible border and no tagtext i.e. it has no displayed title.

Any normal frame defined directly within a **TABSET** frame will become a **tabbed page frame**, with its tag text displayed in the tab; selecting a tab will bring the tabbed page frame to the front, and hide the previously shown tabbed page frame. A page **SHOWN** event is raised for the tabbed page frame whenever the user interactively selects a new page tab.

Handling this event allows the AppWare to modify the content of gadgets on the selected page before it is displayed to the user.

Only one page can be visible at one time, and one page is always visible. To change the visible page programmatically, you have to set the new page visible — setting the current page invisible has no effect. It's not possible to remove a tabbed-page from the set, but you can deactivate it, so it is inaccessible to the user.

The tabbed page frame may contain any form gadgets including normal frame gadgets.

To create a **multi-paged tabbed form**, you can define a dialog or document form with a single **TABSET** frame that contains a set of tabbed page frames, with each page containing the gadgets for the desired page of the form.

The form will automatically resize to take account of the largest tabbed page frame. There is an example showing the definition of tabbed frames in *Complex Form Layout*

You cannot nest **TABSET** frames.

18.1.3 Toolbar Frames

Main forms support the notion of user-defined toolbars (see *Modules and Applications*). You can create these using toolbar frames.

The frame type **TOOLBAR** allows you to define formal toolbars that contain all the gadgets within the frame's scope.

A toolbar frame can contain only a subset of gadget types: **BUTTON**, **TOGGLE**, **OPTION**, **TEXT, COMBOBOX**, **SLIDER** and **NUMERICINPUT**. It must have a name and can appear only on main forms; moreover, a toolbar frame is the only kind of frame a main form can contain.

The frame gadget's visibility attribute allows you to show and hide toolbars, as well as individual gadgets within the toolbar.

Note that any gadgets belonging to a main form and defined outside of a formal toolbar frame definition are interpreted as a default toolbar with the name '`Default`'.

18.1.4 PANEL Frames

This is like a Normal frame but with no enclosing box (by default). You can formally follow the PANEL keyword by INDENT to get a 3D indented surround. The Panel also never displays its tag text.

18.1.5 Fold Up Panel Frames

This is like a Panel but has a formal title bar, which displays the tag text and provides an icon which can be clicked to fold-up and hide, or unfold and show the contained gadgets.

19 Gadgets and their Attributes

As a user, you will already be familiar with forms and their gadgets. The types of gadgets that you can define on the body of the form are summarised below.

Refer to *Figure 20:1.: Examples of different types of gadgets.* for examples of the following:

Gadget	Purpose
FRAME	A container that groups other gadgets visually and logically. It also acts as a radio group for all directly included RTOGGLE gadgets.
PARAGRAPH	Display-text, for information.
BUTTON	Act as visual buttons and are pushed to achieve some action. The buttons like **OK** and **Apply**, which appear on many forms, are examples of a special type of button, which use **form control attributes** to control the display of the form and whether it is actioned.
COMBOBOX	Similar to a Window Combobox. A combination of a dropdown list and a single line textbox.
LINE	Allows horizontal and vertical lines to be drawn on a form to assist visual grouping of gadgets.
NUMERIC INPUT	Allows numeric input within a specific range.
TOGGLE	These gadgets have just two settings: on or off. You can have several **TOGGLE** gadgets on a form, and they will normally all be independent.
OPTION	Has a pull-down list of options associated with it, from which the user can choose one only.
LIST	Displays one or more lines of data from which the user can select one or several. The list can be scrollable if there are too many options to be displayed at once.
CONTAINER	Allows the hosting of an external control inside a PML defined form.
DATABASE SELECTOR	Used to display a list of database elements, from which the user can choose.
TEXT	A text-box where the user can type input.

Gadget	Purpose
TEXTPANE	An area where the user can type and edit multiple lines of text, or cut and paste text from elsewhere on the screen.
VIEW	Used to display alphanumeric or graphical views. There are several different types, used for displaying different types of information.
SLIDER	The **SLIDER** gadget allows you interactively to generate values in a specified range, at specified steps. PML supports both vertical and horizontal **SLIDERS.**
RTOGGLE	A FRAME may have a set of RTOGGLE gadgets defined directly within it which act in concert as a radio group.

19.1 Gadget Definition Commands

You can define gadgets only within the form definition sequence,

```
setup form... exit.
```

A gadget definition (except for menu bars) has the format:

type *name* *{common properties}* *{specific properties}*

For example:

paragraph **.mytext** **at X... Y...** **width 10 lines 3**

Common properties can appear in any order. These include:

position	***All*** gadgets have this
tag	***Most*** but ***not all*** gadgets have these
tooltip	
callback	
anchor	
docking	

Specific properties: the order of these commands generally matters. See the syntax graphs in the *Software Customisation Reference Manual* for definitive information about order.

Most gadget properties can be changed using the members and methods supported by each gadget object.

Note: You cannot change a gadget's geometry except for a **VIEW** gadget on a resizable form.

The *Software Customisation Reference Manual* contains detailed information on the methods available for editing each type of gadget. This chapter describes some of the most frequently used editing operations.

19.2 Some Generic Gadget Members and Methods

For full list see Customisation Reference Manual.

The following members are shared by all gadgets regardless of gadget type:

Member Name	Type	Purpose
`type`	STRING Read only	Gadget type e.g.'BUTTON, LIST etc.
`name`	STRING Read only	User's defined gadget name

The following methods are shared by all gadgets regardless of gadget type:

Method Name	Result	Purpose
subtype()	STRING	Gadget subtype e.g. for Frame gadget returns one of NORMAL, TABSET, FOLDUPPANEL etc.
container( )	FMO GADGET or FORM	Return reference to the Forms & Menus object (FMO) which directly contains the gadget.

19.3 Gadget Size Management

19.3.1 The Effect of the Tag on Gadget Size

All gadgets have a **TAG** attribute that you can set and modify by gadget syntax or methods.

- **SLIDER**, **PARAGRAPH**, **G3D** and **G2D** do not display their tag at all and so it doesn't affect the gadget size.
- **FRAME** gadgets display their tag as an integral part of their surrounding box, and so the tag contributes to the gadget width and height.
- **LIST**, **SELECTOR** and **TEXTPANE** gadgets display their tag above the gadget's box. It affects the gadget height but not the width.
- **OPTION**, **TEXT**, **TOGGLE** and **RADIO_BUTTON** gadgets display their tag horizontally next to the gadget glyph and/or value indication and it affects the width but not the height.
- **BUTTON** gadgets display their tag inside the button glyph whose width and height can now be specified independently of the actual tag string.
- **ALPHA** gadgets (not in **FRAME** gadgets) display their tag above the bounding box, so it affects the gadget height but not the width. **ALPHA** views within **FRAMES** do not display their tag at all, so it has no effect on the size.

19.3.2 User Specifiable Tagwidth for TEXT, TOGGLE and OPTION Gadgets

TEXT, **TOGGLE** and **OPTION** gadgets support the `Tagwidth` syntax.

For example, the **OPTION** definition below:

```
option .ELLIST tagwid 7 |Element Id| width 4
```

Yields a form that looks like this:

The `Tagwidth` specifies the size of the gadget's tag field in grid width units including any padding space, regardless of the actual tag string.

The actual tag may have more characters than the declared `Tagwidth` and still fit in the tag field (typically a mostly-lower-case string of 10 characters occupies only about 60-70% of the 10 X notional character spaces). Similarly the actual strings may have fewer characters than the declared `Tagwidth` and the extra space will be blank padded.

If the tag width is not explicitly given then it is assumed to be the number of characters in the 'tagtext' string multiplied by the horizontal grid size (the notional character width for the font).

Note: You can specify the tag width without specifying any tagtext at definition time; you can add this at run time.

The `Tagwidth` is not needed for gadgets with an explicit area specification (width and height, lines or length). **FRAME**, **LIST**, **SELECTOR**, **TEXTPANE** and **PARAGRAPH** can always force an explicit width (excluding any padding).

For example, the following **PARAGRAPH** gadget definition displays the given string, which has 14 characters without lots of unused space:

```
paragraph .oneline text 'a single line!'  width 9
```

19.3.3 Specifying Gadget Size Relative to Other Gadgets

During form layout, a gadget's position can already be specified relative to the position of previously placed gadgets, for example

```
frame f1 |Frame 1| At Xmin.gad6 Ymax.gad8 + 0.6 ... Width 30 Height 10
```

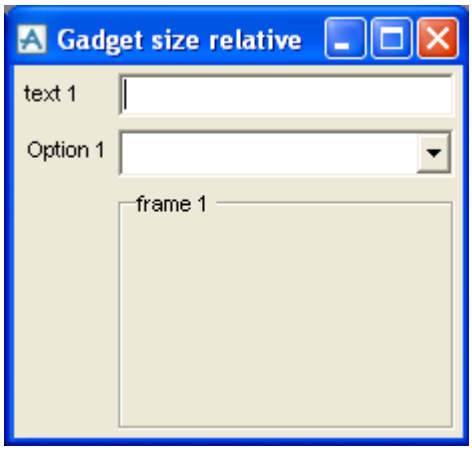
Form layout syntax has been extended to allow a gadget's width and/or height to be specified relative to the size of previously placed gadgets:

```
frame f1 |Frame 1| At Xmin.gad6 Ymax.gad8 + 0.6 ... Width.gad6 Height
```

The frame's width is set to the width of gadget gad6 and its height is set to the height of the last (most recently defined) gadget.

19.3.4 An Example Form

The following example illustrates the combination of gadget relative size and tag width.



And this is the PML that produced it:

```
setup form !!relGadSize dialog resi NoAlign
 title |Gadget size relative|
 path down
 -- define desired tag width value
 !tw = 6
 text .text1 tagWid $!tw |text 1| at 0 0 anchor L+T+R  width 20 is
STRING
 option .option1 tagWid $!tw |Option 1| anchor L+T+R  width.text1
 frame .frame1 |frame 1| anchor all at xmin+$!tw  width.option1 height
5
 . . .
 exit
exit
```

The text field gadget text1 reserves 6 grid units for its tag and 20 grid units for its text display, but the actual gadget length is greater than this because of padding space in the strings and space for the visible box.

The option gadget option1 also reserves 6 grid units for its tag (though the tag text has 8 characters, rather than 6) and specifies the text display (including the drop-down glyph) must be the same width as the text1's text display (excluding the tag).

The two gadgets have the same X co-ordinate and their display fields will align at their start, because they have the same tag width, and at their end because they have the same length.

The frame gadget's X co-ordinate is incremented with the desired tag width and so will also align at its start with option1. The frame's width is set to be the same as the width of option1 (excluding the tag) and so aligns with the end of option1 also.

All the gadgets have their anchor attributes set to allow the form to be resizable. When the form is interactively resized the gadgets will adjust to maintain their initial alignments. The frame will expand to take up any spare space on the form.

19.3.5 The Meaning of Size for the PML Gadget Types

PML often specifies gadgets in terms of the displayed data rather than gadget size, so it is usually difficult to predict the actual gadget size from this definition.

The new tag width specification has overcome this for a gadget's tag .the Relative Gadget Extent specification aims to overcome the difficulty for the gadget's data display part also.

The usual literal **WIDTH** and **HEIGHT** settings mean the following for the PML gadget set:

There are currently three categories of gadget:

Category	Gadget	Effect of WIDTH and HEIGHT
Gadgets with an optional in-line tag.	**TEXT, OPTION** **TOGGLE RTOGGLE**	**WIDTH** and **HEIGHT** define the data display space in grid units. These settings exclude the tag and any gadget glyph or decoration.
Boxed gadgets with optional tag above.	**LIST**, **SELECTOR**, **TEXTPANE**	**WIDTH** defines the width of a data display line in grid units and **HEIGHT** defines the number of visible data lines. The box may include scroll bars to expose data lines that do not fit into the display space defined, and ***excludes any tag***.
Gadgets with no tag or with an integral (optional) tag.	Gadgets that never display a tag: **FRAME**, **SLIDER**, **G2D**, **G3D**, **GM3D**, **PARA**.	**WIDTH** and **HEIGHT** define the (possibly implicit) enclosing box in grid units.
	Gadgets with integral (optional) tag: **FRAME**, **BUTTON**.	**WIDTH** and **HEIGHT** define the implicit enclosing box in grid units.
	ALPHA: a boxed gadget with optional tag above.	**WIDTH** defines the width of a data display line in grid units and **HEIGHT** defines the number of visible data lines. The box may include scroll bars to expose data lines that do not fit into the display space defined, and may also include a data input line with a prompt above.

The relative gadget extent settings define the data display part of the gadget being defined in terms of the data display part of the specified or implied gadget.

For the three categories defined above**, WIDTH.GAD** and **HEIGHT.GAD** have the following meanings:

Category	Gadget	Effect of WIDTH and HEIGHT
Gadgets with an optional in-line tag.	**TEXT, OPTION** **TOGGLE RTOGGLE**	**WIDTH** and **HEIGHT** define the (implicit) enclosing box ***excluding any tag.***
Boxed gadgets with optional tag above.	**LIST**, **SELECTOR**, **TEXTPANE**	**WIDTH** and **HEIGHT** define the enclosing box ***excluding any tag.***
Gadgets with no tag or with an integral (optional) ta**G**.	Gadgets that never display a tag: **FRAME**, **SLIDER**, **G2D**, **G3D**, **GM3D**, **PARA**.	**WIDTH** and **HEIGHT** define the (possibly implicit) enclosing box ***including any tag*.**
	Gadgets with integral (optional) tag: **FRAME**, **BUTTON**, **ALPHA**.	**WIDTH** and **HEIGHT** define the (possibly implicit) enclosing box ***including any tag*.**

19.4 Gadgets that Support Pixmaps

Some gadgets support pixmaps as content as an alternative to text. e.g. Buttons, Toggles, and Paragraphs.

Pixmaps are pixelated pictures held in files of type `.png`.

When pixmaps are required you will need to specify pathnames to the pixmap file and the maximum required size of the image in width and height, both measured in pixels

The default size for pixmaps is assumed to be 32x32 pixels.

For example:

```
button .ButtonName pixmap /buttonpix WIDTH 26 HEIGHT 26
```

19.4.1 Selected and Unselected States

BUTTON and **TOGGLE** gadgets may have two associated pixmaps for the states **SELECTED** and **UNSELECTED**, in that order.

```
button .B1 pixmap /pix1_sel /pix1_unsel WIDTH 26 HEIGHT 26
```

- If only one pixmap file is supplied, it will be used for all states.
- If the Selected pixmap is unset, then it reverts to the Unselected one.

19.4.2 AddPixmap Method

The AddPixmap method is the best way of setting or changing a gadget's associated pixmaps.

AddPixmap(!pixmap is STRING)

!pixmap is a string holding the file pathname of the required .png file.

AddPixmap(!pixmap1 is STRING, !pixmap2 is STRING)

!pixmap1 corresponds to the Un-selected state of the gadget, and pixmap2 corresponds to the Selected state. Specifying !pixmap1 as the null string ' ', will leave the current Selected pixmap unchanged.

PARAGRAPH gadgets only have one pixmap which is represented by the .VAL member, and so can be directly set or changed using **!this.myPara.val = '<pixmap-pathname>**.

Notes:

1. The PML function !!PML.GetPathname('<myPixmap>.png') returns the required pixmap pathname for pixmaps held in the standard PDMS Appware pixmap directories.
2. It is recommended that when you define the gadget you set its size to encompass the largest pixmap which you will later add. Failure to do this may give rise to unexpected behaviour.
3. Historically you could add a third pixmap which was used when the gadget was de-activated. This practice is no longer necessary as the gadget pixmapped is automatically greyed-out on de-activation.

19.5 De-activating gadgets: Greying Out

You may de-activate a gadget by setting its *active status* to **FALSE**:

```
!!MyForm.List.Active = FALSE
```

You may re-activate a gadget by setting its *active status* to **TRUE**:

```
!!MyForm.List.Active = TRUE
```

When a gadget is de-activated, it has a **greyed-out** appearance and the user is prevented from interacting with it.

19.5.1 Greying Out Gadgets on Forms

You can de-activate or re-activate all the gadgets on a form using the form method:

```
!MyForm.setactive( TRUE )
!MyForm.setactive( FALSE )
```

If you want most of the gadgets on the form to be de-activated, you can de-activate all gadgets on the form (and then activate the ones you want, individually) using the following form methods:

```
SetGadgetsActive( active is BOOLEAN )
SetActive( active is BOOLEAN )
```

`SetActive(false)` greys out all gadgets on the form, but doesn't set their Active status, so that `SetActive(true)` restores the form to the precise state it was in before greying-out, that is, any inactive gadgets will still be inactive.

The command:

```
SetGadgetsActive( false )
```

Greys-out all gadgets on the form and sets their *active status* to *inactive* i.e. their previous active state is lost. Similarly

```
SetGadgetsActive( true )
```

greys-in all gadgets and sets their *active status* to *active*.

19.6 Making Gadgets Visible and Invisible

All gadgets have a **visibility member attribute** that you can access with PML. For example, to make a gadget invisible:

```
!!myform.mygadget.visible = false
```

And to check a gadget's visibility:

```
!bool = !!myform.mygadget.visible
```

19.6.1 Making Gadgets Visible and Invisible in Frames

To make invisible the gadgets owned by a frame, you can set the **visible status** of the frame to *false*. Conversely, to make them visible again you can set the *visible status* to true.

The frame's **visible** property will automatically apply to all of its children, but will not overwrite their corresponding property values.

So, for example, if frame **.f1** contains button .**b1** and **f1** and **b1** both have `Visible = true`.

The command:

```
!!form.f1.visible = false
```

will cause the frame and all its gadgets to be hidden, but the query:

```
!value = !!form.b1.visible
```

will still return true.

19.7 Setting Keyboard Focus

The **keyboard focus** defines which gadget will receive keystrokes. The **FORM** object has as one of its properties the name of the gadget which will initially have the keyboard focus. This default keyboard focus can be changed by setting the **keyboardfocus** member of the form:

```
!!MyForm.keyboardfocus = !!Myform.gadget
```

The keyboard focus can be moved at any time to one of the following types of gadget by calling the `setfocus()` method on the target gadget:

- **TEXT**
- **TEXTPANE**
- **BUTTON**
- **TOGGLE**
- **RTOGGLE**
- **SLIDER**
- **LIST**
- **SELECTOR**
- **OPTION**
- **ALPHA VIEW**

For example:

```
!!MyForm.Textfield.Setfocus()
```

19.8 Refreshing Gadgets

In general, when a gadget is edited inside a PML callback function or macro the visual change is delayed until the entire callback has been executed or interactive input is sought. If you want to see the gadget's appearance change immediately it is edited, use the method

```
!Gadget.Refresh()
```

You can also use the command **REFRESH** to refresh all gadgets of all displayed forms.

Warning: Refreshing gadgets should be used sparingly, especially for View gadgets, as it will cause their content to be regenerated.

19.9 Gadget Background Colour

The gadget method `Background()`, which returns the gadget's background colour as a colourname string, is available to all gadgets (except **BAR**).

The following restrictions apply:

- Some gadgets do not support this property in all circumstances, e.g. gadgets showing a pixmap.
- Gadgets whose colour has not been set explicitly, may not have a colour with a known colourname. In this case an error is raised.

20 Gadget Set

This chapter describes all the gadgets supported by PML. It covers their definition and use.

Note: In some of the examples the relative gadget positioning commands are indicated by the **AT** keyword followed by three dots . . . See *Relative Placement* for details.

20.1 Examples

This section contains two examples, which illustrate much of the functionality described in this manual.

20.1.1 Simple Form Layout

Figure 20:1.: Examples of different types of gadgets., illustrates **PARAGRAPH** gadgets, **FRAMES**, **TOGGLES** and **BUTTONS**.

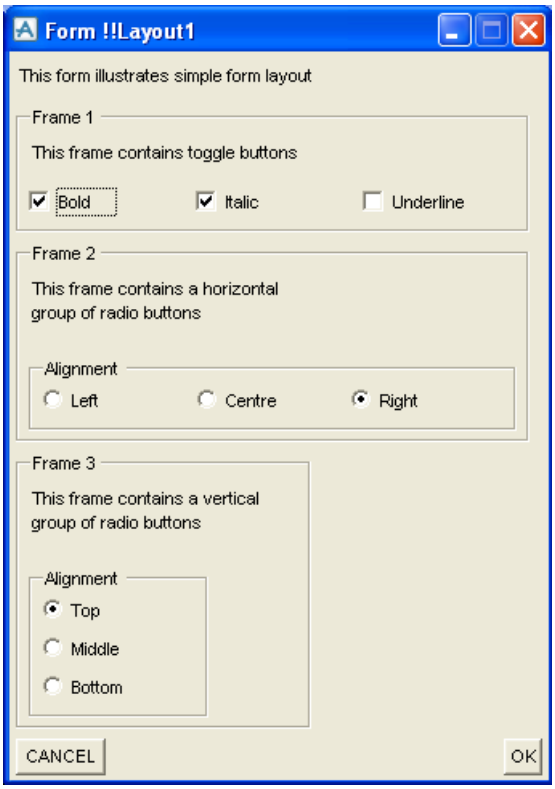


Figure 20:1. Examples of different types of gadgets.

This file `layout1.pmlfrm`, shown below, defines the form shown in *Figure 20:1.: Examples of different types of gadgets.* Note that all the gadgets on the form are dummies: there are no callbacks. The PATH commands control the layout: they are described in *Positioning Gadgets on a Defined Path*.

```
-- PDMS Customisation User Guide
-- Form layout1 - Demonstrate simple form layout
setup form !!layout1 DIALOG
 title 'Form !!Layout1'
 -- defaults: path right, hdist 1.0, vdist 0.2
 paragraph .Message text 'This form illustrates simple form
layout'
 path down
 frame .frame1 'Frame 1'
  paragraph .Message1 text 'This frame contains toggle buttons'
  toggle .Bold      'Bold     '
  path RIGHT
  toggle .Italic    'Italic   '
  toggle .Underline 'Underline'
 exit
 PATH DOWN
 frame .frame2 'Frame 2' at XMIN FORM
  paragraph .Message2 text 'This frame contains a horizontal
group of radio buttons' width 20 lines 2
  .Horizontal 'Alignment' HORIZONTAL
   add tag 'Left     ' select 'LEFT'
   add tag 'Centre   ' select 'CENTRE'
   add tag 'Right    ' select 'RIGHT'
 exit
 frame .frame3 'Frame 3' at XMIN FORM
  paragraph .Message3 width 20 lines 2
  .Vertical 'Alignment'
   add tag 'Top      ' select 'TOP'
   add tag 'Middle   ' select 'MIDDLE'
   add tag 'Bottom   ' select 'BOTTOM'
 exit
 button .CANCEL CANCEL
 button .OK at XMAX FORM-SIZE YMAX FORM-SIZE OK
exit
define method .layout1()
```

```
 -- Constructor

 !this.Message3.val = 'This frame contains a vertical group of
radio buttons'

endmethod
```

20.1.2 Complex Form Layout

The figure below illustrates Tabset frames, Text gadgets, Options, Lists, Text Panes, Paragraphs and Buttons. Note that this is a fully functional, docking form which illustrates the use of form methods, particularly the Constructor method, and the use of open callbacks. It also demonstrates the use of the gadget's dock and anchor attributes to achieve forms with intelligent resize behaviour.

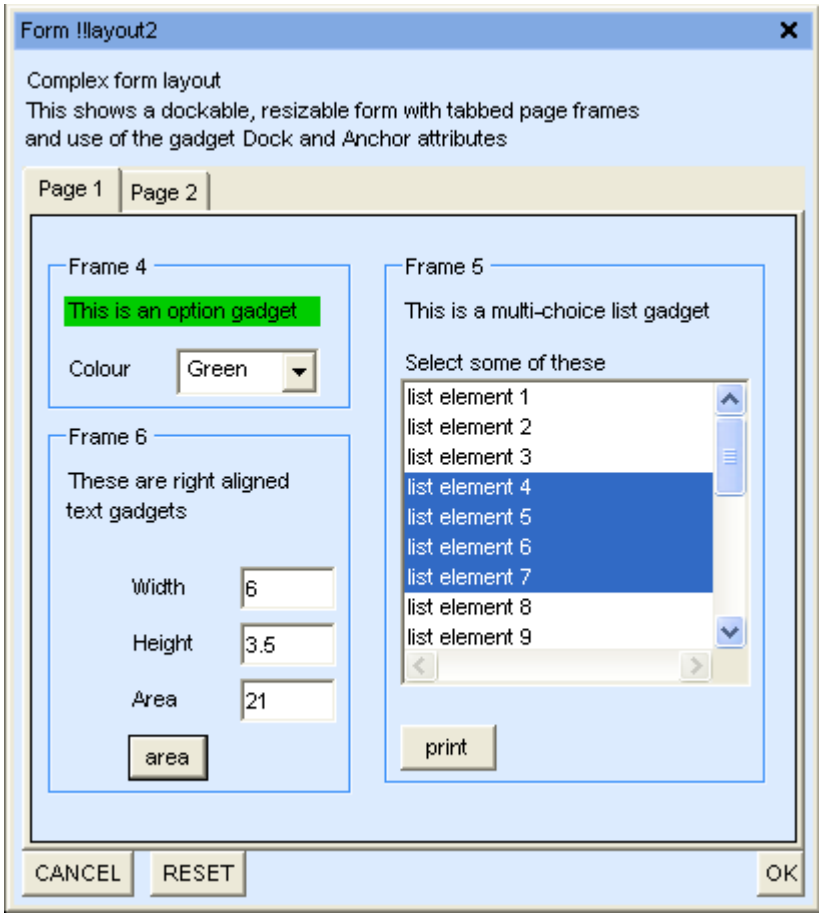


Figure 20:2. Complex Form Layout

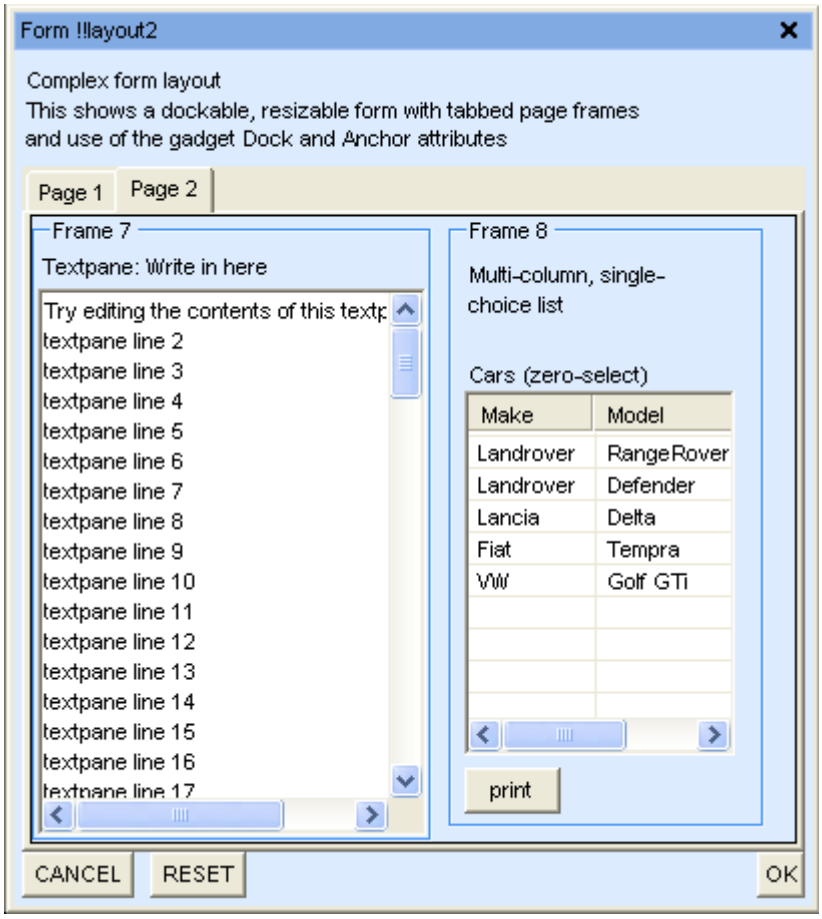


Figure 20:3. The tabbed pages of a complex form

The file `layout2.pmlfrm`, shown below, defines the form shown in *Figure 20:3.: The tabbed pages of a complex form*. Within the form definition the **TABSET** frame is defined and directly contains a frame gadget for each tabbed page. Note its **ANCHOR ALL** setting which maintains the distance between each edge of the frame and the corresponding edge of the form, when the form is resized by the user. This allows the frame to grow and shrink without overwriting gadgets outside of it.

Each tabbed page frame contains the definition of all the gadgets belonging to the page. Note the use of the **DOCK FILL** setting which allows each edge of the tabbed page frame to stick to the corresponding edge of the **TABSET** frame so that they grow and shrink in unison.

Alternatively, when you define a **TABSET FRAME**, you can specify its **DOCK** or **ANCHOR** attributes to determine its resize behaviour within your form.

For each tabbed-page frame within the **TABSET**, it is no longer necessary to specify any **DOCK** or **ANCHOR** attribute settings, and any that are specified will be ignored. Each tabbed-page frame will always fill the available space in its **TABSET** parent (it exhibits **DOCK FILL** behaviour).

The gadget **ANCHOR** attribute is used extensively to allow resizable gadgets to expand in specific directions and not others for. It is also used by non-resizable gadgets, e.g.

BUTTONs, to allow them to move with edges of their containers and so avoid being overlaid by resizing gadgets.

Note also, the extensive use of form methods to provide the intelligence of the form as gadget callbacks. In particular the method `listCallback(!list is GADGET, !event is STRING)`, which just reports details of select and unselect events on list fields, has the standard form of an open callback, and is used as such on the list gadgets **LI1** and **LI2**, i.e. `!this.Li1.callback = |!this.listCallback(|.` Open callbacks are described in *PML Open Callbacks*.

```
-- PDMS Customisation User Guide
-- Form layout2 - Demonstrate complex form layout
setup form !!layout2 dialog docking
  title 'Form !!layout2'
  paragraph .Message width 40 lines 3
  path DOWN
  frame .Tabset TABSET 'tabset' anchor All
    frame .page1 |Page 1| dock Fill
      frame .frame4 'Frame 4'
           paragraph .Message4 text 'This is an option
gadget' width 16
          option .Colour 'Colour'width 5 tooltip'select
colour for paragraph'
      exit
      frame .frame6 'Frame 6'
        halign right
        paragraph .Message6 text 'These are right aligned
text gadgets' width 16 lines 2
        text .Width  'Width ' width 5 is REAL
        text .Height 'Height' width 5 is REAL
        text .Area   'Area  ' width 5 is REAL
        halign left
        button .b3 |area| tooltip'calculate the area'
      exit
     frame .frame5 'Frame 5' at Xmax.frame4+2 Ymin.frame4
anchor All
        paragraph .Message5 text 'This is a multi-choice
list gadget' wid 20
          list .Li1 'Select some of these' anchor all
MULTIPLE width 20 height 10
```

```
        button .b1 |print| Anchor L + B tooltip'print list
selections'
      exit
    exit
    frame .page2 |Page 2| at 0 0 dock Fill
      frame .frame7 'Frame 7' dock left width 20 height 10
        textpane .text 'Textpane: Write in here' dock Fill
width 1 height 1
      exit
       frame .frame8 'Frame 8' at Xmax.frame7 y 0 anchor
All
        path down
          paragraph .Message8 text 'Multi-column, single-
choice list' width 15 lines 2
           list .li2 |Cars (zero-select)| anchor T+B+L+R
columns single zerosel width 15 height 11
        button .b2 |print| Anchor L + B tooltip'print list
selection'
      exit
    exit
  exit
  path right
   button .CANCEL at XMIN form YMAX form anchor L + B
CANCEL
  button .RESET anchor L + B RESET
  button .OK at XMAX form-size anchor R+B OK
exit
define method .layout2()
  -- Constructor - initialise gadget values
  -- main form gadgets
  -- paragraph
  !this.message.val = |Complex form layout
This shows a dockable, resizable form with tabbed page
frames and use of the gadget Dock and Anchor attributes|
  -- tooltips
   !this.CANCEL.setTooltip('discard values and hide the
form')
```

```
  !this.RESET.setTooltip('reset to initial values')
   !this.CANCEL.setTooltip('accept values and hide the
form')
  --Page 1
  -- frame 4
    -- option
    !ColourArray[1]='Black'
    !ColourArray[2]='White'
    !ColourArray[3]='Red'
    !ColourArray[4]='Green'
    !ColourArray[5]='Blue'
    !This.Colour.Dtext=!ColourArray
    -- set callback - make paragraph the selected colour
     !this.colour.callback = |!this.message4.background =
!this.colour.selection('dtext')|
  -- frame 5
    -- multi-choice list
    do !i from 1 to 20
      !Elements[!i] = 'list element $!i'
    enddo
    !This.Li1.Dtext= !Elements
    -- set callbacks
    !this.b1.callback =
|!this.printListSelections(!this.li1)|
    !this.li1.callback = |!this.listCallback(|
  -- frame 6
    -- make Area read-only
    !this.Area.seteditable( false )
    !this.Width.val = 6.0
    !this.Height.val = 3.5
    !this.b3.callback = '!this.calcArea()'
--Page 2
  -- frame 7
    -- textpane - add data
```

```
     !s[1] = 'Try editing the contents of this textpane
gadget'
    do !i from 2 to 100
      !s[!i] = 'textpane line $!i'
    enddo
    !this.text.val = !s
  -- frame 8
    -- multi-column list
    -- Define headings
    !a[1] = 'Make'
    !a[2] = 'Model'
    !a[3] = '  Price'
    !this.li2.setHeadings(!a)
    -- set up dtext rows as array of array
    !Row1[1] = 'Landrover'
    !Row1[2] = 'RangeRover'
    !Row1[3] = '£62000'
    !Row2[1] = 'Landrover'
    !Row2[2] = 'Defender'
    !Row2[3] = '£23999'
    !Row3[1] = 'Lancia'
    !Row3[2] = 'Delta'
    !Row3[3] = 'not for sale'
    !Row4[1] = 'Fiat'
    !Row4[2] = 'Tempra'
    !Row4[3] = 'offers'
    !Row5[1] = 'VW'
    !Row5[2] = 'Golf GTi'
    !Row5[3] = 'p.o.a.'
    do !i from 1 to 5
      !dtext[!i] = !Row$!i
    enddo
    !this.li2.setRows( !dtext )
    -- Add data
```

```
    do !i from 1 to !dtext.size()
      !rtext[!i] = 'row $!i'
    enddo
    !this.li2.rtext = !rtext
    -- set callbacks
    !this.b2.callback =
|!this.printListSelection(!this.li2)|
    !this.li2.callback = |!this.listCallback(|
endmethod
define method .listCallback(!list is GADGET, !event is
STRING)
  -- open callback to report on list events
  -- can be used for any list gadget
  !n = !list.pickedField
  !sel = !list.dtext[!n]
  !name = !list.fullname()
  $P $!event callback on field $!n<$!sel> for list $!name
endmethod
define method .calcArea()
  -- calculate the area
  !area = !this.width.val * !this.height.val
  !this.Area.val = !area
endmethod
define method .printListSelection(!list is GADGET)
  -- report single-selection list gadget selection
  -- can be used for any single-choice list
  !sel = !list.selection('Dtext')
  !num = !list.val
  !name = !list.fullname()
  $P ------------------------------------------------
  $P selected field for list $!name
  $P Field $!num: <$!sel>
  $P ------------------------------------------------
endmethod
define method .printListSelections(!list is GADGET)
```

```
    -- report multi-selection list gadget selections
    -- can be used for any multi-choice list
    !sels = !list.selection('Dtext')
    !nums = !list.val
    !name = !list.fullname()
    !nvals = !sels.size()
    $P ------------------------------------------------
    $P $!nvals selected fields for list $!name
    do !n from 1 to $!nvals
      $P Field $!nums[$!n]: <$!sels[$!n]>
    enddo
    $P ------------------------------------------------
endmethod
```

20.2 Frame Gadgets

Frame gadgets provide visual grouping of gadgets on a form, and aid form layout.

The grouping is more than visual: a frame is a genuine container and manages the gadgets it contains, so gadgets are positioned relative to the frame's origin rather than the form's origin.

When a container resizes it will adjust the layout of all its children, taking account of their **anchor** or **dock** properties.

The frame gadget's properties **visible** and **active** will automatically apply to all of its children, but will not overwrite their corresponding property values. So, for example, **frame .f1** contains **button .b1** and **f1** and **b1** both have **visible = true**.

The command:

```
!!form.f1.visible = false
```

will cause the frame and all its contained gadgets to be hidden, but the query:

```
!value = !!form.b1.visible
```

will still return true.

20.2.1 Defining a Frame

You define a frame using a command sequence beginning with the command `frame` and ending in `exit`.

All the gadgets defined after the `frame` command and before `exit` will be included in and contained by the frame.

The following example defines Frame1 and its gadget collection. The frame sub-type <frame-type> is either one of the supported types TABSET, TOOLBAR, PANEL, FOLDUPPANEL, or omitted for a 'normal' or 'tabbed page frame'.

```
frame .frame1 <frame-type> '<frame-tag>'
 paragraph .Message1 text 'This frame contains toggle buttons'
 PATH DOWN
 toggle .Bold 'Bold'
 PATH RIGHT
 toggle .Italic 'Italic'
 toggle .Underline 'Underline'
exit
```

Frame Size

During form definition, once a frame is positioned on the form the origin remains fixed but the frame automatically expands downwards and to the right to contain any gadgets added to it. You cannot add gadgets to it at negative coordinates relative to the frame's origin. You can optionally declare a minimum size for the frame. For example:

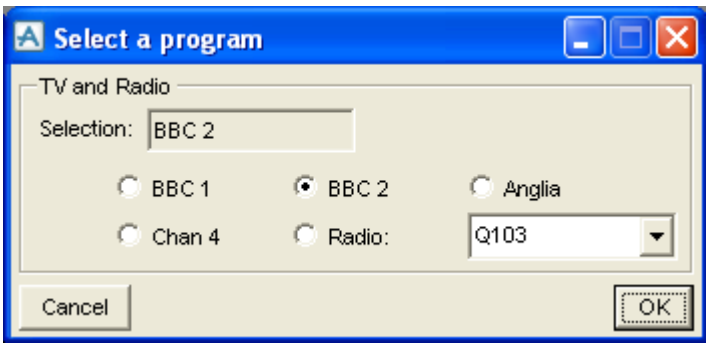
```
Frame frame1 'frame 1' SIZE 10 20
```

This is relevant only for **NORMAL** and **TABSET** frames; for **TOOLBAR** frames, the gadgets appear as a single row in the order defined i.e. gadget positions are ignored.

20.2.2 Frame Radio Groups

A FRAME may have a set of RTOGGLE gadgets defined directly within it which act in concert as a radio group.

An example of a **FRAME** with a directly defined radio group looks like this:



The radio group action only applies to **FRAME** gadgets of type **NORMAL, PANEL, FOLDUPPANEL**.

You can add **RTOGGLE** to a **FRAME** with the usual positioning and layout commands.

The **FRAME** has a value member, **VAL**, which is the index of currently selected **RTOGGLE** for the radio group. You can use this to change the selected **RTOGGLE**.

Similarly, you change the value of the **FRAME** by setting the **VAL** member of any of the group's **RTOGGLE**s to true.

Note that the **FRAME** group value may be set to zero, indicating that there is no selected **RTOGGLE**. Similarly if the selected **RTOGGLE** value is set to false, then it becomes deselected and the **FRAME** value will then be zero.

The default value for an **RTOGGLE** gadget is **FALSE**, and the default value for a **FRAME** gadget is zero, i.e. no selected **RTOGGLE**.

Frame Callbacks

The **FRAME** gadget can have an assigned callback, which is executed when the radio group selection is changed, i.e. whenever the user selects an unselected radio-toggle. As there is only a SELECT action supported, it can be either a simple callback or an open callback.

The form definition below is a simple TV and radio station selector, shown above.

```
setup form !!FRGTest dialog noAlign
  title |Select a program|
  Frame .rg1 |TV and Radio|
   path down
   text .choice tagwid 6 |Selection:| width 12 is STRING
   rToggle .rad1 tagwid 7 |BBC 1| States '' 'BBC 1'
   path right
   valign centre
   rToggle .rad2 tagwid 7 |BBC 2| States '' 'BBC 2'
   rToggle .rad3 tagwid 7 |Anglia| States '' 'Anglia'
   rToggle .rad4 tagwid 7 |Chan 4| at xmin.rad1 ymax.rad1
States '' 'Chan 4'
   rToggle .rad5 tagwid 7 |Radio:| States '' 'radio'
   option .Radio width 10
   exit
  button .cancel |Cancel| at xmin form ymax form + 0.2 CANCEL
  button .ok | OK | at xmax form - size OK
 - set focus to button to ensure to ensure Windows does not
set it to first Rtoggle
 !this.keyboardFocus = !this.ok
exit
```

Note: The form's keyboard focus is initially placed on the **OK** button to prevent it being assigned (by Windows) to the first **RTOGGLE rad1** (effectively the first interactive gadget on the form as the text field **Selection** is read-only)

The form constructor method assigns a simple callback, the form method `RGroupSelectionChanged()`, to the frame **rg1** (radio group). It then initialises the gadget values and selects the second **RTOGGLE** as the default selection.

```
define method .FRGTest()
 -- Constructor
 -- Frame radiogroup with simple callback
 !this.rg1.callback  = '!this.RGroupSelectionChanged( )'
```

```
 -- set result field read-only
 !this.choice.setEditable(false)
 -- Radio choices
 !this.rad5.setTooltip(|select your Radio option|)
 !radio[1] = 'Q103'
 !radio[2] = 'Hereward'
 !radio[3] = 'Cambridge'
 !radio[4] = 'ClassicFM'
 !radio[5] = 'Caroline'
 !this.Radio.dtext = !radio
 !this.Radio.setTooltip(|change your Radio option|)
 !this.Radio.callback = '!this.selectProgram(!this.rad5)'
 -- set initial value
 !this.rg1.val = 2
 !this.RGroupSelectionChanged( )
Endmethod
```

The group callback uses the **FRAME**'s **VAL** member to get the current selected index and hence the current **RTOGGLE** and its **OnValue** member value. If the selected **RTOGGLE'S** value is 'radio' then the selected program is read from the **RADIO** option gadget. Finally the selected program string is displayed in the **Selection** (read-only) text gadget.

```
define method .RGroupSelectionChanged( )
 -- Service radiogroup select event
 !Frame = !this.rg1
 !index = !Frame.val
 !rTog  = !Frame.RToggle(!index)
 !value = !rTog.onValue
 -- Do some application actions
 if( !value eq 'radio' ) then
  !value = !this.Radio.selection('dtext')
 endif
 !this.choice.val = !value
endmethod
```

The callback on the **RADIO** option gadget, detects if the 'Radio:' **RTOGGLE rad5** is current and if so it deselects it leaving no current selection, and clears the **Selection** text field.

```
define method .selectProgram( !rtog is GADGET )
 -- Select new program from option list
```

```
 if( !this.rg1.val eq !rtog.index ) then
  -- rtog is current, so deselect it within group
  !this.rg1.val = 0
  !this.choice.clear()
 endif
endmethod
```

20.2.3 Managing Pages in Tabset Frames

Within a Tabset frame, whenever the user interactively selects a new tab a HIDDEN event is raised for the previous tabbed page frame and then a SHOWN event is raised for the new one, which pops to the front. The HIDDEN and SHOWN callbacks are only executed for tabbed page frames which provide an Open callback.

If you want to manage tabbed pages that are also radio groups, then you must supply an open callback so you can differentiate the **SELECT** (**RTOGGLE**) event and the (page) **SHOWN** event.

Setting a tabbed page frame's VISIBLE property, e.g. **!this.TabbedPage.visible = true**, selects it and gives it focus, but does not raise HIDDEN or SHOWN events.

The example below shows a typical form method you could use as a PML open callback to handle frame events for any frame gadgets defined on a form:

```
define method .frameEvents(!frame is GADGET, !event is
STRING)
 -- Frame events open callback handler
 if( !event eq 'SELECT' ) then
  --Handle radio button selection
  !selection = !frame.val
  ...
 elseif( !event eq 'SHOWN' ) then
  -- tabbed page selected
  -- modify page content
  ...
 endif
endmethod
```

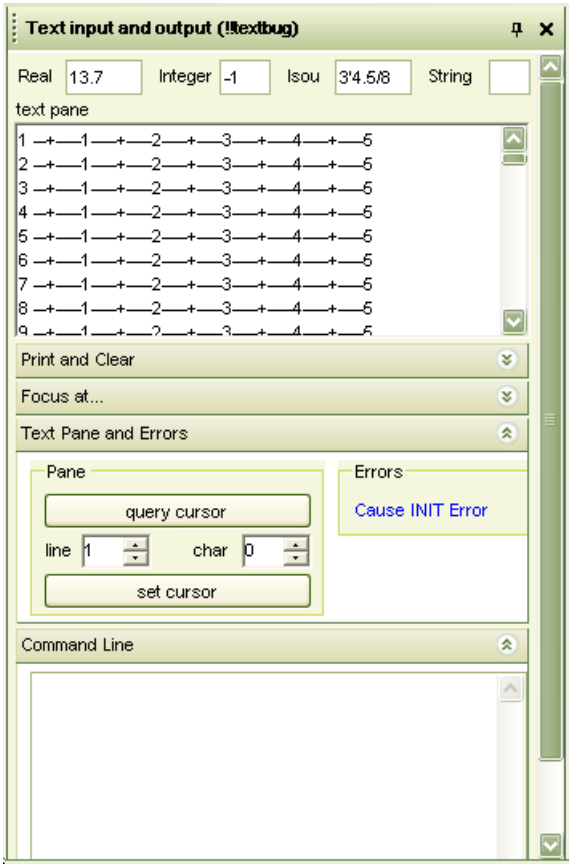
20.2.4 Managing the Fold Up Panel

The form shown below is a docking dialog which has four FOLDUPPANEL gadgets, the first two are collapsed (hidden) and the second two are expanded (shown). Each one has a title bar which displays the panel's tag text, and an icon which allows the panel to fold-up or fold-down when clicked.

The default state is 'unfolded' and the EXPANDED property allows the user to initialise a panel to unfolded (true) or folded (false).

When the panel expands or collapses, any gadgets which lie below the panel and between (or partially between) the panel's horizontal limits will be moved down or up the form.

If the form's AutoScroll attribute is selected, then a scroll bar will automatically appear whenever gadgets have been moved off the bottom of the form, so that the entire form is always accessible.



Fold Up Panel Events

HIDDEN and SHOWN events are raised whenever the user interactively folds or unfolds the panel. These events are only fired if an Open callback is defined for the foldup frame. To manage FoldUpPanels which are also radio groups, then you must supply an open callback so that you can differentiate the panel's SELECT, HIDDEN and SHOWN events.

20.3 CONTAINER Gadget

The Container gadget allows the hosting of an external Control, e.g. a PMLNet, control inside a PML defined form. It allows the user to add an external .Net control, which may raise events that can be handled by PML. In order to customise the context menus of the .Net control, the Container may have a PML popup menu assigned to it. This is shown when the .Net control raises a 'popup' event.

A CONTAINER gadget can be created by

container .NET indent PMLNETControl dock fill

The following section contains the PML code for an example form !!MyNetForm.

By default the Container will be will have a drawn surrounding box, but you can select NOBOX or INDENT which produces a 3-D indented surround.

The Dock and Anchor attributes are supported to allow intelligent resize behaviour. The enclosed Control must support resizing and is usually set as Dock fill, so that it follows size changes of the Container.

The following restrictions apply:

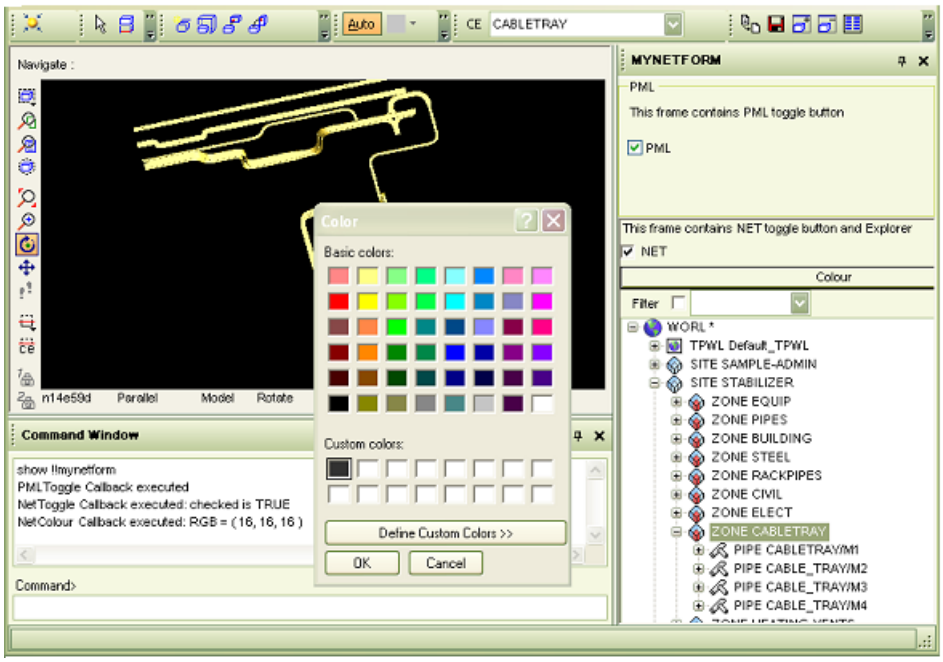
- Tagtext can be specified but is never displayed.
- Positioning must be specified before size (<vshape>).
- Currently only Controls of type PMLNet are supported.

20.3.1 Example of Container Gadget

This section creates a form, MYNETFORM, which hosts a PMLNet control.

First you will need to have created your PMLNet control - See .Net Customisation Reference Manual.

In this example the control is called MyNetControl, and comprises a text label, a toggle (NET), a button (Colour) and a PDMS explorer, as shown in the picture.



MyNetControl supports three events, namely OnCheck, OnPopup and OnColour.

- OnCheck - raised when the NET toggle is clicked. Callbacks keep the PML toggle and the .NET toggle in step.
- OnPopup - raised when the right mouse button is pressed while the cursor is over the background of the control. Callbacks edit and show the PML popup menu.

- OnColour – raised when the user selects a colour from the standard Winforms colour picker (as pictured), which is shown when the colour button is pressed. Callback prints the RGB components of the selected colour.

The PML code to create and manage the form is given below. Purple is used to highlight the PMLNet commands, the rest is just standard PML2. It includes rudimentary callback methods to service the PMLNet controls events.

In your Appware you need an import command (to load the dll containing your PMLNetControls)

```
import 'PMLNetTest'
```

This is usually done once only, probably in application start-up macro. You can have it in your form, but you will then need to handle the error which will occur if the import is executed more than once:

```
import 'PMLNetTest'

handle (1000,0)
-- PMLNetTest dll already loaded
endhandle
-- MyNetForm.pmlfrm: Test form hosting a PMLNet Control
setup form !!MyNetForm size 25 20 dialog dock right
using namespace 'Aveva.PDMS.PMLNetTest'
member .MyCtrl is MyNetControl
!this.FormTitle = 'My PMLNet Form'
path down
-- define PML Container to host .Net control
container .NET  indent PMLNETControl  dock fill
-- define PML Frame
frame .PMLFrame 'PML' dock top
   paragraph .PMLMessage text 'This frame contains a PML toggle button' wid 25
   toggle .PMLToggle 'PML '
exit
-- define PML popup menu
menu .PMLPopup popup
   !this.PMLPopup.add( 'CALLBACK', 'Attributes', '!this.attributesMenu()' )
   !this.PMLPopup.add( 'CALLBACK', 'More', '!this.moreMenu()' )
   !this.PMLPopup.add( 'CALLBACK', 'Last', '!this.lastMenu()')
exit
----------
-- MYNETFORM Constructor
----------
define method .MyNetForm()
   using namespace 'Aveva.PDMS.PMLNetTest'
   -- create instance of .Net control
   !this.MyCtrl = object MyNetControl()
```

```
-- add .Net control to PML container
!this.NET.control = !this.MyCtrl.handle()

-- add PML event handlers to .Net control, to service (some of) the events it raises

!this.MyCtrl.addeventhandler('OnCheck',  !this, 'NETToggleCallback')
!this.MyCtrl.addeventhandler('OnPopup',  !this, 'NETPopupCallback')
!this.MyCtrl.addeventhandler('OnColour', !this, 'NETColourCallback')

-- add callback to the context menu, so we can edit the menu before it gets popped up
!this.PMLPopup.callback = '!this.editMenu()'

-- add PML callback for PML Toggle
!this.PMLToggle.callback = '!this.PMLToggleCallback(!this.PMLToggle.val)'
endmethod

----------------------

-- Callback methods

----------------------

define method .NETPopupCallback(!x is REAL, !y is REAL)
   -- service the .Net control's popup event
   !this.NET.popup = !this.PMLPopup
   !this.NET.showPopup( !x, !y )
endmethod

define method .editMenu()
   -- Edit the popup menu which is about to be shown
   !this.PMLPopup.add( 'CALLBACK', 'New field', '$p this is a new field' )
endmethod

define method .attributesMenu()
   -- service the menu field
   $P attributesmenu callback executed
endmethod

define method .moreMenu()
   -- service the menu field
   $P moremenu callback executed
endmethod

define method .lastMenu()
   -- service the menu field
   $P this.Lastmenu callback executed
endmethod

define method .NETToggleCallback(!checked is BOOLEAN)
   -- service the 'checked' event for the toggle in the .Net control
   -- keep PMLtoggle in step
   !this.PMLToggle.val = !checked
endmethod

define method .NETColourCallback(!red is REAL, !green is REAL, !blue is REAL)
   -- service the 'colour selected' event for the button in the .Net control
   $P NetColour Callback executed: RGB = ( $!red, $!green, $!blue )
endmethod

define method .PMLToggleCallback(!checked is BOOLEAN)
   -- service the PML toggle callback
```

```
    -- keep .NETtoggle in step
    !this.MyCtrl.val(!checked)
endmethod
```

20.4 Paragraph Gadgets

Paragraph gadgets allow a **text** or a **pixmap** to be displayed on a form. This gadget has no associated name-tag and no call-back command: it is passive so cannot be selected by the user.

A paragraph gadget can contain text or a pixmap. Once it has been defined, a textual **paragraph** cannot be changed to a pixmap **paragraph** or vice versa.

Paragraph gadgets support the **DOCK** and **ANCHOR** attributes.

20.4.1 Textual Paragraph Gadgets

A **textual paragraph** gadget is defined by the `paragraph` command.

A paragraph gadget's size may be specified explicitly in terms of **width** and **height**, or defined implicitly by the initial value, or content. See *Gadget Size Management*. Once it has been defined, the size of the gadget cannot be changed.

An initial value of a paragraph gadget can be set using the **TEXT** keyword.

You do not need to set the value in the `setup form` sequence: you can leave it until a value is assigned to its **val** member, but you must give a textual paragraph gadget an initial size if you do not give it an initial text value.

```
paragraph .message text 'Text string'
paragraph .message AT . . . text 'Text string' width 16 lines 7
paragraph .message AT . . . background 2 width 20 lines 2
```

For multi-line paragraphs the text is line-wrapped and formatted into the available space. It can contain explicit newlines to achieve the desired format.

20.4.2 Pixmap Paragraph Gadgets

A **pixmap paragraph** gadget has a fixed width and length that may be specified explicitly in terms of width and height in pixels of the largest pixmap that is to be displayed, or defined implicitly by the initial value. Once it has been defined, the size of the gadget cannot be changed. The default size for a pixmap is 32x32 pixels.

```
paragraph .picture AT . . .
pixmap /filename
paragraph .picture AT . . .
pixmap /filename width 256 height 200
```

The pixmap may be changed at any time by assigning a new value to the **.val** member:

```
!!MyForm.picture.val = /newfilename
```

20.4.3 Textual Paragraph Gadgets

The background colour may optionally be set using the **BACKGROUND** keyword and a colour specification.

20.5 Button Gadgets

When the user presses a **button gadget** (control button) it will usually display a child form or invoke a call-back - typically a PML Form method.

Buttons have a tag-name or pixmap which is shown within the button rectangle. The tag, pixmap, call-back, and child form are all optional.

(See *Gadgets that Support Pixmaps* for more about pixmaps).

For example:

```
button .SubForm 'More . . .' FORM !!ChildForm

button .SubForm pixmap /filename FORM !!ChildForm

button .Calculate 'Calculate' CALLBACK
'!!this.CallbackFunction()'
```

You can specify the width of the **BUTTON** independently of any tag text string it contains using the **WIDTH** syntax. You can also define its height with the **HEIGHT** syntax, allowing you to define textual **BUTTON**s taller than a single character in height.

For example:

```
Button .btn1 |reject selection| at … width 10 height 1.5
```

Note: The **BUTTON**'s tag is always centre-aligned within the define area.

20.5.1 Buttons of Type Toggle

Buttons can optionally be used in toggle mode, where they show visually differentiated **pressed** and **unpressed** states, similar to **TOGGLE** gadgets.

Buttons with Pixmaps

For these buttons, the ***Unselected*** and ***Selected*** pixmaps swap whenever the button is pressed, so alternating between the two images in the pressed and un-pressed states.

Textual Buttons

Toggle buttons will highlight when pressed. For example on toolbars they will turn from blue to orange when pressed, and go back to blue again when un-pressed.

The syntax to select the new mode is `toggle`**,** which can be anywhere after the button name and before the button control type or associated form, e.g.

```
Button .B1 TOGGLE pixmap /Unselected.png /Selected.png /
Inactive.png width 16 height 16 tooltip…
```

The button's value-member **!button.val** is a **BOOLEAN** and reflects the button's state, having the value **TRUE** when the button is pressed and **FALSE** when it is not.

20.5.2 Buttons of type LINKLABEL

The Linklabel, provides a purely textual button presentation, i.e. it has no enclosing box. It is often used to indicate a link to some application item, e.g. a hyperlink to a file, a link to an associated form. They do cause validation of any modified text fields of the form whenever they are pressed.

The tag text is shown in a different colour to all other gadget's tag text. The link label gadget highlights by underlining when the mouse cursor passes over it. Pressing it causes a SELECT event to be raised and runs any associated call back.

Linklabels have the following restrictions:

- They don't support change of background colour.
- They don't support 'pressed' and 'not pressed' value.
- They can have popup menus, though this is not recommended.
- They don't have Control Types e.g. OK, CANCEL etc.

The sub-type of any Button gadget can be queried using the it's Subtype method.

20.5.3 Form Control Attributes

A button may optionally have a **form control attribute**, such as **OK**, which takes effect ***after*** the callback command is invoked.

It is convenient, but not essential, to give a button the same PML name and displayed tag name as its control attribute.

If no form control attribute is specified, the effect of the button depends entirely on the callback or the showing of a child form.

You can only have one of each type of control attribute on any form, apart form APPLY which may be used on several buttons.

Control Attribute	Purpose
OK	Allows the user to approve the current gadget settings and action the form. The form nest's OKCALL callbacks are run (see *Form OK and CANCEL Callbacks*) and the nest is hidden. Any callback on the **OK** button is ignored.
APPLY	Similar to **OK** in that the gadget settings are approved and the form is actioned but not removed from the screen. There may in fact be several **APPLY** buttons for different sections of form (ideally each section within its own Frame). A form with one or more **APPLY** buttons should also be given a **DISMISS** button for removing it from the screen.
CANCEL	Allows the user to decides not to proceed with the form. The form nest's CANCELCALL callbacks are run and the nest is hidden. All gadget values are reset to their initial settings or to the values established at the last **APPLY**.

Control Attribute	Purpose
RESET	Returns the values of all gadgets on the form to the values they had when the form was displayed. If the user has since pressed an **APPLY** button, the form gadgets are reset to the values they had when the **APPLY** button was last pressed. The callback is then invoked in which your PML code should ensure that anything that needs undoing is indeed undone.
HELP	Invokes online help.

The effect of **OK** and **CANCEL** on gadgets is more extensive if a form family is involved, as described in *Free Forms and Form Families*.

Examples:

```
button .Ok AT . . . 'OK' CALLBACK '!!MyOkFunction()' OK
button .Apply 'Apply' CALLBACK '!!MyApplyFunction()'  APPLY
button .Cancel 'Cancel' CALLBACK '!!MyCancelFunction()'  CANCEL
button .reset AT . . . 'Reset' RESET
button .help AT . . . 'Help' HELP
```

20.5.4 Defining a Dismiss Button

To define a **dismiss button**, use a command like this:

```
button .Dismiss 'Dismiss' CANCEL
```

Note that this button deliberately does ***not*** have a callback. When this button is pressed, the form nest is removed from the screen and its CANCELCALL callbacks executed.

20.6 Toggle Gadgets

TOGGLE gadgets are used for independent on/off settings as opposed to a radio group. A **TOGGLE** should be given a tag name or pixmap, which is displayed to the right of the **TOGGLE** button.

In *Figure 20:1.: Examples of different types of gadgets.*, the three **TOGGLES** are defined by the following lines.

```
toggle .Bold 'Bold'
toggle .Italic 'Italic'
toggle .Underline 'Underline'
```

They are named **Bold**, **Italic** and **Underline**, and they are tagged with corresponding text, enclosed in quotes.

More examples:

```
toggle .Italic 'Italic' AT . . .
toggle .GridOn pixmap /filename callback '!!MyFunction()'
```

The value of a toggle gadget is set and used via the **.val** member which is a **BOOLEAN** value:

```
!!MyForm. Italic.val = TRUE

if ( !!MyForm.GridOn.val ) then

...

else

...

endif
```

The default value for a toggle is **FALSE**.

20.7 RToggle Gadgets

The **RTOGGLE** gadget is very similar to the **TOGGLE** gadget, but is allowed only in **FRAME**s, where they operate together to form a set of radio buttons, only one of which can be selected at any one time.

You can add **RTOGGLE** gadgets to a **FRAME** with the usual layout and positioning commands. The **RTOGGLE** gadgets are implicitly numbered 1, 2, 3, … ***n*** as they are added.

RToggle Callbacks

The **RTOGGLE** gadget can have an assigned callback, which is executed whenever its selected status is changed. When the group selection is changed, by the user clicking an unselected radio button, the current button (if any) is unselected and the new button is selected. An open callback (PML function or form method) is necessary as the events **UNSELECT** and **SELECT** need to be reported.

The PML code below shows a modification to our example form, which makes use of open callbacks on the **RTOGGLE**s instead of a simple callback on the **FRAME** radio group. The `Constructor` and the `RgroupSelectionChanged` methods are modified accordingly.

Note: The behaviour of the two versions is identical. Both mechanisms are equally valid, and are provided to minimise the work required in replacing the **RGROUP** and (deprecated) **RADIO** gadgets.

```
define method .FRGTest()
 -- Constructor
 -- Frame radiogroup
 -- set result field read-only
 !this.choice.setEditable(false)
 --TV and Radio with open callbacks
 !this.rad1.callback = '!this.RGroupSelectionChanged('
 !this.rad2.callback = '!this.RGroupSelectionChanged('
 this.rad3.callback = '!this.RGroupSelectionChanged('
 this.rad4.callback = '!this.RGroupSelectionChanged('
```

```
 this.rad5.callback = '!this.RGroupSelectionChanged('
 -- Radio choices
 !this.rad5.setTooltip(|select your Radio option|)
 !radio[1] = 'Q103'
 !radio[2] = 'Hereward'
 !radio[3] = 'Cambridge'
 !radio[4] = 'ClassicFM'
 !radio[5] = 'Caroline'
 !this.Radio.dtext = !radio
 !this.Radio.setTooltip(|change your Radio option|)
 !this.Radio.callback = '!this.selectProgram(!this.rad5)'
 -- set initial value
 !this.rad2.val = true
 !this.RGroupSelectionChanged( !this.rad2,'SELECT' )
endmethod
define method .RGroupSelectionChanged( !rtog is GADGET,
!event is STRING )
 -- Service specified radio-toggle events
 if( !event eq 'UNSELECT' ) then
  -- Do some application actions
  !this.choice.clear()
 elseif( !event eq 'SELECT' ) then
  !value = !rtog.onValue
  -- Do some application actions
  if( !value eq 'radio' ) then
   !value = !this.Radio.selection('dtext')
  endif
  !this.choice.val = !value
 endif
endmethod
```

Order of Event Generation

Events for the radio group **FRAME** and its radio-toggles happen in the following order, when an **RTOGGLE** is selected:

UNSELECT on previously selected **RTOGGLE** (if any)

SELECT on new **RTOGGLE**

SELECT on **FRAME**

20.8 Option and Combobox Gadgets

The **OPTION** and **COMBOBOX** gadgets offer a single choice from a list of items. Clicking the down-arrow icon opens the drop-down list to allow a selection of a field from the choices. The currently selected field is highlighted.

The two gadget types are similar, the main differences being:

- Option gadget's display text field cannot be edited.
- Combobox gadget's display text field is editable, just like a TEXT gadget.
- Combobox does not support the display of pixmaps.

The drop-down list is very similar to a Single Choice List Gadget and supports DTEXT, RTEXT, ZEROSELECT and NORESELECT properties. The same methods are used to manage the list's content. In just the same way it also allows UNSELECT and SELECT events.

When the user presses the option gadget, the entire set of items is shown as a drop-down list and the user can then select a new item by clicking the option required.

There is always a selected value unless the option list is empty.

You can use the Add methods to add a single new entry to be appended to **OPTION** gadgets:

```
Add( !Dtext is STRING )
Add( !Dtext is STRING, !Rtext is STRING )
```

Where **Dtext** is the text to be displayed in the option list, and **Rtext** is the replacement text for the new field.

If **Rtext** is not specified, it will be set to the **Dtext** string by default.

20.8.1 Textual Option Gadgets

The width of a textual option gadget, in grid units, must be specified. A tag name is optional and is displayed to the left of the gadget.

```
option .Colour 'Colour'  width 10
```

The current value in a textual option gadget is scrollable using the left- and right-arrow keys. This means you can specify a gadget that’s narrower than the options displayed in the drop-down list.

The **OPTION** gadget actually contains two parallel lists of the same length, the **display values** (or **Dtext**) and the **replacement values** (or **Rtext**). The list of display values ***must*** be supplied by assigning an array of values to the gadget's **Dtext** member. This is the list of choices displayed to the user.

In *Figure 20:1.: Examples of different types of gadgets.*, *Examples*, the lines in the default constructor method define the **Colour** option gadget values as follows:

```
!ColourArray[1]='Black'
!ColourArray[2]='White'
!ColourArray[3]='Red'
```

```
!ColourArray[4]='Green'
!ColourArray[5]='Blue'
!This.Layout2.Colour.Dtext=!ColourArray
```

Other examples:

```
option .Colour 'Colour:' AT . . .  callback '!!MyFunc()'
width 20
```

20.8.2 Combobox Gadgets

A COMBObox is a combination of an option gadget and text field. It can be defined by the command

combo .Colour tagwid 5 'Colour' scroll 20 width 5

When the ComboBox is editable(default), with the drop-down list closed, the user can search for a required option by typing the first few letters into the display field and clicking the down-arrow. The list will open with the first matching option highlighted. This is useful for large lists.

The display field is accessible to the user, who can edit the contents by typing or pasting text into the field. If the user clicks ENTER while the gadget's text field has focus and has been modified, a VALIDATE event is raised. You can trap this event by assigning a PML Open callback to the gadget. This allows you to give meaning to the action of typing text into the display field. The Open callback is necessary to differentiate the VALIDATE event from the SELECT and UNSELECT events.

On receipt of the VALIDATE event, your callback method can retrieve the displayed text by means of the **DisplayText** method and decide what action is associated. Additionally you can assign a popup menu to the gadget, which gives the user the choice of several actions.

For example, you might append the current display text to the drop-down list as a new choice, possibly ensuring that there are no duplicate entries. An assigned popup menu could allows options to be removed from the drop-down list and the editable status of the combobox to be toggled.

(PML example code is available as User Manual example Layout2.pmlfrm which can be obtained from AVEVA's support web site.)

20.8.3 Pixmap Option Gadgets

The gadget-shape must be specified, using the **WIDTH** keyword and either **HEIGHT** or **ASPECT**. A tag name is optional and is displayed to the left of the gadget.

The display text should be set to pixmap's filename and assigned to the **Dtext** member:

```
!CircleDtextsArray[1] = '/directory/circle/cenrad'
!CircleDtextsArray[2] = '/directory/circle/3pts'
!!MyForm.Circle.Dtext = !CircleArray
option .Circle1 AT . . .  callback '!!MyFunc()'  PIXMAP
width 256 height 128
option .Circle2 AT . . .  callback '!!MyFunc()'  PIXMAP
width 256 aspect 2.0
```

The replacement-texts, if needed, are set by assigning an array of values to the **Rtext** member.

```
!CircleRtextsArray[1] = 'Specify centre and radius'
!CircleRtextsArray[2] = 'Pick three points on the circumference'
!!MyForm.Circle.Rtext = !CircleArray
```

20.8.4 Setting and Getting the Current Selection

The default selection is the first value in the list. You can explicitly set the currently selected value by means of the option gadget's `select()` method:

```
!!Layout2.Colour.select('Dtext','Orange')
!!MyForm.Circle.select('directory/circle/cenrad')
```

You can **read** the current selection using the `selection()` method. This will return the replacement-text (or the display-text if replacement-texts were not defined):

```
!SelectedText = !This.List.Selection()
```

The **.val** member reads the index number of the currently selected value:

```
!ChosenNumber = !!Form.List.Val
```

The `clear()` method will discard both display- and replacement-text lists:

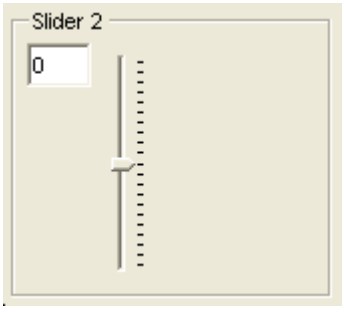
```
!!MyForm.Colours.clear()
```

20.9 Slider Gadgets

The **SLIDER** gadget allows you interactively to generate values in a specified range, at specified steps. PML supports both vertical and horizontal **SLIDERS**.

An example **SLIDER** definition is:

```
Frame .fr2 |Slider 2| at xmin form ymax anchor All width 20
 text .t2  wid 3 is REAL
 slider .sl2  vertical  anchor All  range -50 +50 step 5 val 0 height 5
exit
```



20.9.1 Event Callbacks

The **SLIDER** gadget responds to left-mouse **SLIDER START**, **MOVE**, and stop events at which it executes the gadget's callback if one is defined.

We recommend you use an open-callback (PML function or form method) as it includes the action 'START', 'MOVE' or 'STOP'. For example the form method .serviceSlider would have the signature

```
define method .serviceSlider( slider is GADGET, action is STRING )
```

The callbacks for action **START** and **MOVE** are not followed by an automatic update, for efficiency reasons, so you may need to follow some gadget modifications carried out in the callback with a !gadget.refresh(), e.g. modifying a **TEXT** gadget to track the **SLIDER**'s current value.

The MOVE callback is generated at each step increment in the **SLIDER**'s range.

20.10 Line Gadgets

The LINE gadget gives the ability to display horizontal or vertical lines to separate groups of gadgets on a form, for increased clarity of intent. The line's presentation reflects the colour of the current Windows scheme.

A Line gadget can be defined by **line .horiz 'H-Line' Horiz width.f3 height.t1**

The line's width and height can be set either specifically or in terms of the width of other gadgets on the form.

Setting the height for a Horizontal separator or the width for a Vertical separator causes the line to be drawn across the middle of the implied area. This allows for equal spacing on each side of the separator line. If the width or height is omitted then a default value is assumed.

The line's Dock and Anchor attributes allow it to be dynamic and respond to interactive changes in form size.

The Line has the following restrictions:

- The tag text is never displayed.
- It cannot not appear in toolbar frames.
- It is not interactive and has no associated value.
- It supports the standard default gadget members and methods only.

20.11 Numeric Input Gadget

The NUMERICINPUT gadget allows numeric input within a specified range, with given granularity. It has Up/Down arrow icons which control incrementing and decrementing the displayed value by the specified increment, within the range. The tag text is always displayed.

Additionally it is possible to type in the required value, which is adjusted to the nearest valid value in the range. The default initial value is the minimum value of the range and the maximum value is adjusted so that the range is an integral number of steps.

It is not possible to provide user formatting of the values displayed by the gadget.

It has the following properties and methods:

val is REAL -	value of the numeric input.
range is array of REAL -	a real array with members Start, End and step (>0).
ndp is REAL -	(read only) the number of decimal places. If zero then all values will be integer.
editable is BOOLEAN -	enable/disable ability to edit the displayed value.
modified is BOOLEAN -	enable/disable modified events.
setRange	(!range is array of REAL, !ndp is REAL)

The NumericInput gadget supports SELECT and MODIFIED events, and users may provide a callback method to service these events. Note that often no callback is required, and the numeric input value is merely read and used by other gadgets of the form.

A SELECT event is raised whenever the user clicks ENTER while the numeric input display field has focus. Typically this happens after the user has typed in a required value, but will also apply if the user enters the field after modifying the values using the up/down arrows. The callback can be a simple or an Open callback.

A MODIFIED event is raised for each modification of the displayed value using the up/down arrows. Modified events are only reported if they are enabled and the user has provided an Open callback, as this allows differentiation from the SELECT events. The default state is modified events disabled.

20.12 List Gadgets

A **list gadget** allows the user to make single or multiple selections from many alternatives. The list of choices is scrollable if the list is too long to be displayed all at once.

A **LIST** gadget is thus appropriate if you have a long list of choices, or if you are constructing the list dynamically (at run time), especially if you cannot predict how many choices there will be.

You must specify whether the gadget is a single or multiple-choice list, and give the width and height of the displayed portion.

The length defines the number of choices that will be visible within the gadget. You may optionally specify a text tag to be displayed at the top-left above the gadget, the position on the form and a callback command.

Typically you enter text values into the list using an array of strings and assigning to its **Dtext** member.

```
list .List 'Select some of these' MULTIPLE width 15 height
8
 !Elements[1]= 'Element 1'
 !Elements[2]= 'Element 2'
 !Elements[3]= 'Element 3'
 !Elements[4]= 'Element 4'
```

```
!Elements[5]= 'Element 5'
!Elements[6]= 'Element 6'
!This.List.Dtext= !Elements
```

More examples:

```
list .Components SINGLE  width 10 height 15
list .Components 'Colour:' AT . . .  MULTIPLE width 10 height 15
list  .Elements  'Elements  to  be  deleted'  callback '!this.Delete' MULTIPLE  width 10 length 15
```

As with the option gadget, the list gadget actually contains two parallel lists, of the same length, one containing display values (**Dtext**) and the other containing replacement values (**Rtext**).

The **Dtext** values ***must*** be supplied, but the replacement values are optional.

If you don't supply the **Rtext** values they will default to the **Dtext** values. The **Rtext** values are often used to assign callback strings to each field of the list.

Resetting a list gadget's display-texts automatically deletes the existing display and replacement-texts and clears any current selections. For example, the contents of gadget **List** of *Figure 20:3.: The tabbed pages of a complex form* could be replaced by:

```
!Choices[ 1 ] = 'Tea'
!Choices[ 2 ] = 'Coffee'
!Choices[ 3 ] = 'Chocolate'
!This.List.Dtext = !Choices
```

You can replace the list's **Rtext** with a new array of the same length without disturbing the current **Dtext**s:

```
!newRtext[1] = ‘drink6’
!newRtext[2] = ‘drink4’
!newRtext[3] = ‘drink12’
!This.List.Rtext = !newRtext
```

You can use the new **Add** methods to add a single new entry to be appended to **LIST** and **SELECTOR** gadgets:

```
Add( !Dtext is STRING )
Add( !Dtext is STRING, !Rtext is STRING )
```

Where **Dtext** is the text to be displayed in the option list, and **Rtext** is the replacement text for the new field.

If **Rtext** is not specified, it will be set to the **Dtext** string by default.

20.12.1 Single Choice List Gadgets

You can **set** and **get** the current selection in a single-choice list using the display-text, replacement text, or **.val** member. For example:

```
!This.List.val =2 - selects second list field
!This.List.Select( 'Dtext', 'Coffee' )
!This.List.Select( 'Rtext', 'drink4' )
!fieldNumber = !This.List.val
!Rtext = !This.List.selection()
!Dtext = !This.List.selection('Dtext')
```

Zero-Selections and Single Choice List Gadgets

You can also define a single-choice list-gadget to allow zero-selections..

The list syntax allows you to define the gadget with the **SINGLE ZEROSELECTIONS** keyword to indicate that the list is single choice with no mandatory selected field, e.g.

```
list .List |Cars| Anchor all single zerosel width 25 length 10
```

The val member now allows programmatic de-selection of the current field.

For Single choice lists the keyword NORESELECT disables UnSelect and Select events when the currently selected field is clicked with the mouse, for example:

```
list .l1 |List gadget| zeroSel noReselect width 15 length 5 tooltip 'single choice list'
```

For ZeroSelection lists it is now possible to interactively deselect the selected field by clicking in unused rows or after the last column.

Single choice List gadgets support UNSELECT events. Typically when a field is selected, an UNSELECT event is raised for the previously selected field (if any) and then a SELECT event is raised for the new field. An UNSELECT event is raised whenever a selected field is interactively deselected.

Notes:

1. UNSELECT events are not notified to PML unless an open callback has been specified (so that SELECT and UNSELECT events can be differentiated).
2. Typically the UNSELECT action allows Appware to manage consequences of deselection for any dependent gadgets or forms.
3. It is recommend that you do not change the List's selection programmatically in an UNSELECT event.

See the *Software Customisation Reference Manual* for more information.

20.12.2 Multiple Choice List Gadgets

You can **read** the current selections in a multiple-choice list using the `selection()` methods. This will return the replacement-texts:

```
!Xarray = !This.List.selection() - returns selected replacement texts by default
!Xarray = !This.List.selection('Dtext') - returns selected display texts
```

To read the index numbers of the currently selected fields of a multi-choice list gadget:

```
!ChosenNumbersArray = !!Form.List.Val
```

You can read back the current **Dtext**s or **Rtext**s as an array or simply as strings:

```
!array = !This.List.Dtext - get all the Dtexts
!string = !This.List.Rtext[3] - get Rtext of the third list field
```

You can select fields of this list either singly (additive) or by array of **Dtext** or **Rtext**, using its `select()` methods:

```
!This.List.select('Rtext', 'Tea')
!This.List.select('Dtext', !array)
```

Callbacks on Multi-Choice List Gadgets

At PDMS11.6 we introduced support for Extended Selection Mode for multi-selection lists, whereby **CTRL** and **SHFT** keys can qualify the list selection. As a result a whole set of **UNSELECT** events followed by a whole set of **SELECT** events can result from a single (atomic) user action. These events are presented in sequence, but AppWare cannot tell where the sequence starts or ends.

At PDMS11.6, problems may arise if a multi-selection list is programmatically changed during the list callback. Modifying the list content or its selection during the sequence can cause unexpected results or even software crashes.

At PDMS11.6Sp1 we nave introduced new **START** and **STOP** batch actions to bracket the sequence of **UNSELECT** and **SELECT** event actions.

For maximum upwards compatibility, the **START** and **STOP** batch actions are only notified to PML if the user has assigned an open callback, since this is the only way that different event types (actions) can be differentiated.

AppWare which used simple callbacks and worked at PDMS11.6 will continue to work because **START** and **STOP** events will not be notified to it.

AppWare which used open callbacks and worked at PDMS11.6 will continue to work if the **SELECT** and **UNSELECT** meta-events were explicitly differentiated in the callback, as shown below, because the new **START** and **STOP** events will be ignored

```
Define method .listSelectionChanged( !list is GADGET, !action is STRING )
 . . .
 if( !action eq 'SELECT' ) then
 . . .
 elseif( !action eq 'UNSELECT' then
 . . .
 endif
 . . .
endmethod
```

If you experience a change in behaviour, then you may need to rewrite your callback to explicitly handle the new **START** and **STOP** events and modify the list content or its selection only on receiving the **STOP** event action.

For newly written AppWare we strongly advise you use open callbacks to service multi-selection list gadgets.

20.12.3 Multi-Column List Gadgets

You can simulate a multi-column list using spaces and tabs if you are using a fixed-width font. But if you are using a proportionally spaced font, then you cannot do this and guarantee the columns will always line up.

The list gadget may have multiple columns as shown in the following example, which is also a single-choice, zero-selection list. The column widths can be interactively modified using the mouse cursor in the headings row.

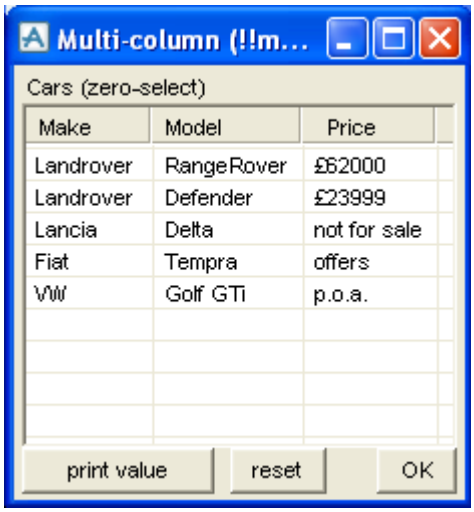


Figure 20:4. Multi-Column List Gadget

The number of columns is deduced from the List's data. If the user specifies a set of (1 or more) column headings before the list is populated, then this will determine the number of columns. If no headings are pre-specified then the number of columns is deduced from the display text of the List's first row. This provides upwards compatibility for existing Appware using single column lists.

A List gadget's headings can be replaced after the list has been populated. If the new headings specify the same number of columns then the headings are replaced but the List's data fields and selection remain unchanged. If the number of columns is different, then the list is replaced by an empty list with the new headings. Invoking the Clear() method will clear the list's data fields and rebuild the current headings.

The column headings, and hence the number of columns, for the list are assigned as an array of **STRING**:

```
!This.List.SetHeadings( !headings )
```

The **Dtexts** for each column of each row of the list can be provided as a PML array, where each element is an array of **STRING**. This can be row oriented or column oriented.

```
!This.List.SetRows( !rows )
```

!rows is an array of 'row arrays', and its size determines the number of rows in the list.

```
!This.List.SetColumns( !columns )
```

!columns is an array of 'column arrays', and its size must match the number of columns of the list. The size of all the column arrays must be the same and determines the no of rows in the list.

The default width of each column is determined by the longest header or **Dtext** value supplied. **Rtext** values for each row are supplied as for single column lists.

Selection within the list applies to the entire row not individual cells, but rows of the list can be selected on the basis of a column's **Dtext**:

```
Select( !column is REAL, !dtext is STRING )
```

This selects the first list row whose column has the given **Dtext**. If the list is a multi-choice list then repeated use of this method will add selected rows to the list.

Note: For a multi-column list gadget each row's Dtext string is held as a 'horizontal tab' character separated string of column data, matching the column headings (as opposed to a simple string for single column list). The standard list members **val, Dtext, Dtext[n]** and methods **Select('Dtext',** …**), Selection('Dtext')** work on multi-column lists with the following limitations:

- Querying Dtexts will return the tab separated strings.
- Supplying Dtext to populate the list or invoke selections will require tab separated strings as input.

See the *Software Customisation Reference Manual* for more information.

Popup-Menus on Multi-Column List Gadgets

List gadgets with a popup menu assigned to them will pop up the menu when the right-mouse button is released when the cursor is over the list title or any list selection field.

For multi-column lists right-mouse in a list field will first select that field, and unselect any other selected fields, before the popup appears. This selection behaviour does not occur for single column lists.

The use of popup menus on list gadgets can be confusing so use them with care.

20.13 Database Selector Gadgets

A **database selector** is a special kind of list gadget. It provides a mechanism for displaying the current database element along with its owner and members. The user can also interact with a selector to change the current element.

A single-choice selector permits only one selected value at a time. A typical example is the PDMS **Members List**.

A multiple-selector could be used to display the results of querying database attributes or to select a group of elements for modification.

Its definition is similar to list gadgets.

```
selector .Mem 'members:' SINGLE width 12 height 8 DATABASE
selector .Mem SINGLE width 12 height 8 DATABASE OWNERS
selector .Mem SINGLE width 12 height 8 DATABASE MEMBERS
```

```
selector .Mem AT . . .  SINGLE width 12 height 8 DATABASE AUTO

selector .Mem  AT . . .  MULTIPLE width 12 height 8 DATABASE

selector .Mem callback '!this.MyList(' MULTIPLE width 12 height
8 DATABASE
```

The DATABASE keyword is mandatory.

The `owners` and `members` keywords are optional. By default, a database selector displays the current element together with elements both above it and below it in the database hierarchy. If present, the `owners` keyword specifies that only elements above the current element are shown. If present the `members` keyword specifies that only elements below the current element are shown.

If `auto` is specified, the selector automatically refreshes its displayed contents whenever the current element of the database changes. If absent, the contents of a selector remain as they were when the gadget was initially displayed until the gadget is explicitly refreshed by your PML code. The `auto` keyword and the `multiple` keyword cannot be used together.

To **set** and **get** selected values for a `selector` use the `select()` and `selection()` methods, and the **.val** members, as you do for lists. For example, to access the currently selected value of a single-selector gadget:

```
!element = !This.Members.selection()
```

Note that for a selector gadget, the **Rtext** and **Dtext** are always the same as one another.

20.14 Text Gadgets

A text gadget is a box that can display a value and into which the user may type a value, or edit an existing value.

To type into a text gadget on a form it must have the **keyboard focus** to receive the keystrokes typed by the user. The user can move the focus to another text gadget by selecting it with the mouse or stepping from one gadget to the next using the **TAB** key.

```
text .number AT . . . width 10 is REAL

text .name 'Name:' callback '!!MyFunction' width 10 scroll 100
is STRING

text .active 'Active:' callback '!!MyFunction' width 10 is
BOOLEAN

text .bore 'Bore:' width 10  is BORE  format !!FormatBore

text .password 'Password:' AT . . . width 10 NOECHO is STRING
```

You must:

- Specify the **WIDTH**, which determines the maximum number of character spaces visible. Optionally you may specify a larger **SCROLL** width, which is the maximum number of characters that the gadget can hold, and scroll through. The default scroll-width is 132 characters. The maximum is 256 characters.
- Specify a data type using **IS** which determines how a value assigned to the gadget will be displayed and how a value typed by the user will be interpreted by your PML code. You may also supply a **FORMAT** object which specifies in detail how a value is to be displayed or interpreted (see below).

You may optionally

- Specify a tag name to be displayed to the left of the gadget.

- Specify a callback command.
- Specify a position on the form.
- Specify a **NOECHO** keyword that indicates any characters typed should be displayed as stars: a typical use would be a gadget for the user to enter a password.
- Give the text gadget an initial value, which will be the value accessed by your PML code if it is not modified by the user. To set an initial value for a text input gadget, use its **.val** member:

  ```
  This.name.val = 'PIPE-1'
  ```

- Specify whether the text displayed is editable interactively or is read only.

  When you type into a text gadget its background color changes from white to beige to indicate that has been modified. Its content is actioned when you press **Enter** while the text gadget has focus, or if you press a button on the form.

When a field is actioned its content is read and validated according to the field's type (see *Validating Input to Text Fields*). If an error is detected, then the field's background colour changes to gold to indicate an error and focus is returned to it so that the user can correct the error. If no error was detected then the text gadget's callback (if any) is executed

To get the current value of a text input gadget:

```
!Value = !This.name.val
```

The data type of the variable **!Value** will be the same as the type of the text gadget. To get the textual value use:

```
!string = !Value.String()
```

To set the keyboard focus so that keystrokes come to this gadget:

```
!This.name.SetFocus()
```

20.14.1 Controlling Text Gadgets' Editing

Text fields have a member Editable (read/write), which controls the user's ability to edit the displayed text interactively, e.g.

```
!this.myTextField.editable = false
```

Makes the field read only.

Modified Events for Text Gadgets

When the user finishes modifying the field by clicking **ENTER** while the field has focus, or pressing any button on the form, the field content is validated. If valid, then the field's callback (if any) is notified (executed) of a **SELECT** event, and the field returns to 'white' again indicating that it is no longer modified. Otherwise an error is detected, the field's background colour becomes 'gold' and focus is returned to it for further modification.

The setEditable method allows PML to be notified when the displayed text is modified by user interaction

```
setEditable( !attribute is STRING, !value is REAL )
```

Currently the only attribute supported is `HANDLEMOIFY` which may have the integer values:

- 0 **MODIFIED** events off (default).
- 1 Generate **MODIFIED** event at first user modification only.

Note: **MODIFIED** events are not notified to PML unless the field is editable, modified events are enabled and an open callback has been specified (so that **MODIFIED** and **SELECT** events cannot be differentiated)

Typically, the first **MODIFIED** event is used to allow AppWare to gain control and modify the properties (e.g. **ACTIVE** status) of dependent gadgets, which possibly the user should not have access to until the text field's **VALIDATE** and **SELECT** events have been handled.

The code fragment below defines an **RTOGGLE** that allows a user specified TV program to be typed into an associated **TEXT** gadget.

```
rToggle .rad6 tagwid 7 |TV:|                    States '' 'TV'
text .TV  width 10 is STRING
!this.rad6.callback = '!this.RGroupSelectionChanged('
-- set open callback on text field and option list
!this.TV.callback = '!this.selectProgram('
!this.Radio.callback = '!this.selectProgram('
- handle first Modified event only
!this.TV.setModified( 'handleModify', 1 )
```

The extended (open) callback `selectProgram(`, shown below, intelligently handles the **TEXT** gadget and **OPTION** list. The open callback `RGroupSelectionChanged` sets the value of the 'TV' **RTOGGLE** from the **TEXT** gadget.

```
define method .selectProgram( !gad is GADGET, !event is
STRING )
 -- Select new program from option list or text input
field
 if( !gad.type( ) eq 'TEXT' ) then
  -- Control active state of R-toggle according to
modified state of textfield
  !rtog = !this.rad6
  if( !event eq 'MODIFIED'  ) then
   -- deactivate R-toggle
   !rtog.active = false
  elseif( !event eq 'SELECT' ) then
   -- reactivate R-toggle
   !rtog.active = true
  endif
 else
  -- select radio program from option list
  !rtog = !this.rad5
 endif
```

```
 if( !this.rg1.val eq !rtog.index ) then
  -- deselect current selection
  !this.rg1.val = 0
  !this.choice.clear()
 endif
endmethod
define method .RGroupSelectionChanged( !rtog is GADGET,
!event is STRING )
 -- Service specified radio-toggle events
 if( !event eq 'UNSELECT' ) then
  -- Do some application actions
  !this.choice.clear()
 elseif( !event eq 'SELECT' ) then
  !value = !rtog.onValue
  -- Do some application actions
  if( !value eq 'radio' ) then
   !value = !this.Radio.selection('dtext')
  elseif( !value eq 'TV' ) then
   !value = !this.TV.val
  endif
  !this.choice.val = !value
 endif
endmethod
```

20.14.2 Copying and Pasting into Text Fields

The text gadget supports the standard copy and paste mechanisms:

- Use the mouse **Select button** to copy and the mouse **Adjust button** to paste.
- Use shortcut keys, for example: **Ctrl C** to copy, **Ctrl V** to paste.
- Use the system **Edit** popup menu
  Pasting into a field acts like typing in, and will set the text field's **Modified** appearance.

Note that user defined Popup menus are not supported on Text gadgets because they clash with the system-supplied Edit menu.

20.14.3 Formatting in Text Input Gadgets: Imperial Units

The **FORMAT** object manages the information needed to convert the value of an object or type to a **STRING**, and vice versa.

The **FORMAT** object for a specific data type is normally a global PML Variable used for all text gadgets of that type.

For example, you can create the **FORMAT** object for a **REAL** object, so that a distance is displayed in ***feet-and-inches*** with fractions (rather than decimals) with a maximum denominator of 4:

```
!!RealFMT = object  FORMAT()
 !!RealFMT.DIMENSION =  'L'
 !!RealFMT.UNITS =  'FINCH'
 !!RealFMT.FRACTION = TRUE
 !!RealFMT.DENOMINATOR = 4
 !!RealFMT.ZEROS =  'FALSE'
```

See the *PDMS Software Customisation Reference Manual* for more information on the **FORMAT** object.

The **TEXT** gadget is created with type **REAL** and assigned a **FORMAT** object for converting values to and from text:

```
text  .Dist  'Distance:'  width 10  is REAL  format
!!RealFMT
```

When we assign a value in ***millimetres*** to the text gadget:

```
!!Form.Dist.val = 3505.2
```

The display-text of `11' 6` will be shown in the text gadget.

To switch from displaying ***feet-and-inches*** to displaying ***mm*** all that is necessary is to change the setting of the **units** member of the format object **RealFMT** from **FINCH** to **MM**.

The associated format also determines what can be typed into the field. If the associated format is **FINCH** or **MM** then you can enter distances in either inches or feet and inches.

For example:

Chosen Units	Typed-in Value	Displayed Value
FINCH	138	11' 6
INCH	138	138
INCH	11' 6	138
MM	3505.2	3505.2

Note that in every case the text field value is actually held in ***millimetres*** so that the command:

```
q var !!Form.Dist.val
```

prints

```
3505.2.
```

You can use the format object in your PML code when converting a value to a **STRING**:

```
!StringValue = !!Form.Dist.val.String(!!RealFMT)
```

20.14.4 Unset Text Fields

All of text fields displayed a text string representing the value of a variable or object of a given Type. The string representing an unset value is 'Unset'.

See *Unset Variable Representations* and *UNSET Values and UNDEFINED Variables* for a discussion of unset variables.

Note: Some older style 'untyped' PML1 text fields use the (null) string '' to represent unset values for numeric fields. You should avoid using these old style fields.

You can force an unset value for any text field (of any type) using:

```
!Ths.MyTextclear()
```

You can also assign an unset value to a text field:

```
! = REAL() - defines an unset variable type REAL
! = STRING() - defines an unset variable type STRING
!This.MyText = !x
```

You can check for an unset text gadget of any type using the **BOOLEAN** method `unset()`:

```
If( !This.MyText.val.Unset()) then
 -- value of text is unset
 ...
endif
```

20.14.5 Validating Input to Text Fields

The text field gadget has an optional validation callback member which the user can specify:

```
!textfield.ValidateCall = <callback string>
```

When a text input field is actioned (by modifying it and pressing **ENTER**, or when a button on the form is pressed or the form's `OKcall` is executed), it is automatically checked to ensure that the typed-in value matches the field's **TYPE** and its **FORMAT** constraints. If so, then the user's `VALIDATECALL` is actioned.

The `VALIDATECALL` is used to apply any checks you want. If an error is encountered then the callback raises the error and returns.

Note: The validation callback must not attempt to change the value of the text field. It should just detect and raise an error.

When an error is raised, an error alert is displayed and the text field is left with keyboard focus and error highlight. No further form or gadget callbacks are executed and the form is not dismissed from the screen. The User can then modify the field and re-action it.

The `VALIDATECALL` is a standard callback, and so it can be a single command, a PML function, method or macro, or an open PML method or function. For an open callback, for example:

```
!textField.validateCall = '!this.textvalidate('
```

the corresponding method must be:

```
define method .1textvalidate( !textin is GADGET, !action
is STRING )
```

where the `action` string will be 'VALIDATE'.

An Example of Text Validation:

The form **!!ValidForm** has two text fields of types **REAL** and **STRING**.

There are limits on the values that can be typed into each field: the tooltips on each field explain the constraints.

The `textvalidate` method determines from its **!textin** parameter which field is being validated, and applies the constraints that have been defined. The Handle block traps any unexpected PML errors generated by the validation code. The first if-block of the validation code checks that the text field is not unset. This also prevents the possibility of unexpected errors in the validation code.

```
 -- $Header: /dev/eventlayer/PMLLIB/validform.pmlfrm 1
17/09/04 13:37 Robin.langridge $
-- PDMS Customisation User Guide
-- Form ValidForm - Demonstrate text field validation
setup form !!ValidForm dialog
 TITLE |Text Handling|
 HDIST 3
 text .T1 |Real| at wid 8 is REAL tooltip'range 0.0 to
100.0'
 text .T2 |String| wid 12 is STRING tooltip'anything but
FRED'
 button .CANCEL at XMIN FORM YMAX FORM CANCEL
 button .OK at XMAX FORM - SIZE YMAX FORM - SIZE OK
exit
define method .ValidForm()
 -- constructor
 !this.t1.validatecall = '!this.textvalidate('
 !this.t2.validatecall = '!this.textvalidate('
 !this.OKcall = '!this.printValues()'
Endmethod
define method .textvalidate( !textin is GADGET, !action is
STRING )
 -- validate the given field contents
 -- General form of Validation Callback
 -- !action will contain 'VALIDATE'
```

```
 onerror golabel /Errors
 label /Errors
 --  Include handle block for any unexpected PML errors
handle ANY
  -- Example: can get errors from evaluation of the String field
  -- with an 'unset'value
return error 1 'Invalid input'
 endhandle
 -- Validation code ------------------------------------
 --  Check  for  unset  field  first  (to  avoid  possible unexpected errors)
 if( !textin.val.unset() ) then
  -- validation callback Failed
  return error 2 'Field is unset'
 else
   !field = !textin.name()
  if(!field eq 'T1') then
   -- TYPED field must check range
   !x = !textin.val
   if(!x lt 0.0 or !x gt 100.0) then
    return error 1 'value must be in range [0.0 to 100.0]'
   endif
  elseif(!field eq 'T2') then
   -- any string but FRED
   if(!textin.val eq 'FRED') then
    return error 4 'value must not be FRED'
   endif
  endif
 endif
endmethod
define method .printValues()
 !r = !this.t1.val
 !s = !this.t2.val
 $p values $!r and $!s
```

```
Endmethod
```

20.14.6 Setting the Value of a Text Field

A special `SetValue()` method allows you to set the value of a field programmatically, with full interactive checking applied:

```
.setValue( !value is ANY, !doCallback is BOOLEAN )
```

- If the value's type fails to match the field type, a trappable error is raised.
- If the types match, the value is validated and the `VALIDATECALL` is run, if there is one.
- If the value is invalid then a trappable error is raised. If the field is shown then it is set 'in error', and the focus is returned to it so that the user can edit the value.
- If the value is valid then the method executes the field callback if !**doCallback** is true.

20.15 TextPane Gadgets

A **textpane gadget** provides a box on a form into which a user may type and edit multiple lines of text or cut and paste text from elsewhere on the screen.

The contents of the textpane can be set and queried by your PML code. Optionally the text contents can be made non-editable.

You must specify the initial shape of the gadget. Optionally you may also supply a form position and a tag to be displayed to the top-left of the gadget.

```
textpane  .text  'Text:'  AT . . .  width 10  height 20
textpane  .text  'Text:'  AT . . . height 20  aspect 0.5
```

The value of a textpane is its current contents as an array of strings, where each element of the array corresponds to a line of text in the gadget.

To make the textpane's contents read-only, use the `seteditable` method:

```
!This.text.seteditable(FALSE)
```

To clear the current contents of a textpane gadget:

```
!This.text.clear()
```

To move the keyboard focus to this textpane gadget:

```
!This.text.setfocus()
```

To enquire how many lines of text a textpane gadget currently contains, use the count method:

```
!Nlines = !This.text.count
```

To set line 21 in a textpane gadget to `'Hello World'`:

```
!This.text.setline( 21, 'Hello World' )
```

To read the current contents of line 21 as a string variable:

```
!Line = !This.text.line(21)
```

To set the entire contents of the textpane gadget to an array of strings:

```
!Lines[1] = 'Hello World'
```

```
!Lines[2] = ' '
!Lines[3] = 'Goodbye World'
!This.text.val = !Lines
```

To read the entire contents of the textpane gadget as an array of strings:

```
!LinesArray = !This.text.val
```

To set the current cursor position within a textpane gadget to line 21, character 15 you can use:

```
!This.text.setcurpos(21, 15)
```

or

```
!ArrayRowColumn[1] = 21
!ArrayRowColumn[2] = 15
!This.text.SetCurPos( !ArrayRowColumn )
```

To read current cursor position within a textpane gadget:

```
!ArrayRowColumn = !This.text.CurPos()
```

20.15.1 Cut and Paste in Textpanes

You can use the standard keyboard actions **Ctrl-X**, **Ctrl-C**, and **Ctrl-V** for cutting, copying, and pasting in textpane gadgets.

Note: Mouse-only copy and paste, and local popup-editing menus are not supported

20.16 Fast Access to Lists, Selectors and Textpanes using DO Loops

Sometimes you may have large quantities of data in list, selector or textpane gadgets and you may find that accessing the data using the `selection()` methods may be very slow.

The PML textpane mechanism allows fast access to these gadgets.

With a textpane gadget you can use `do list` so that the **do** counter takes the value of the replacement-text for each of the currently selected rows of the list in turn:

```
do !field list !!FormA.listC
 $P Selected field replacement-text is $!field
enddo
```

For a selector gadget you must use `do selector` so that the **do** counter takes the displayed value of each of the currently selected rows of the list in turn:

```
do !field selector !!FormA.SelectorB
 $P Selected field replacement-text is $!field
enddo
```

For a textpane gadget you must use `do pane` so that the **do** counter takes the value of each of the displayed lines in turn:

```
do !line pane !!FormA.TextPaneD
 $P Text line value is $!line
enddo
```

Note: That on exit from the loop PML destroys the `do` variable for each of these gadget types.

20.17 View Gadgets

Horizontal and vertical scroll bars will appear as necessary.

The types of view gadgets supported are:

- ALPHA views for displaying text output and/or allowing command input.
- PLOT views for displaying non-interactive 2D plotfiles.
- Interactive 2D graphical views
- Interactive 3D graphical views

  See *Manipulating VIEWS*, for details on how to manipulate the 2D and 3D interactive views.

20.17.1 Defining a View Gadget

A view is defined by the command sequence `view … exit:`

```
view .MyView AT . . . viewtype
 height 10 width 20
 contents
 attributes
exit
```

The command sequence may optionally be given on a single line, without the `exit`:

```
view .MyView AT . . . viewtype height 10 width 20 contents attributes
```

Where the ***contents*** and ***attributes*** that depend on the type of View.

You do not have to set any attributes and we recommend that you set the view's attributes in the form's default constructor method rather than in the form setup definition.

20.17.2 Resizable View Gadgets

When the size of the form is changed by the operator, the size of a view gadget is automatically adjusted to fill the space available in its container. The origin of the view (the top left-hand corner) always remains fixed in position and the view's bottom right-hand corner adjusts according to the available space.

Note that view gadgets do not support the **ANCHOR** and **DOCK** attributes. If you want a view that has these behaviours, you can place it in a frame gadget that you declare with the attribute you desire. Since the view will expand to fill the frame, it will be as if you had an anchored or docked the view gadget.

20.17.3 Pop-up Menus in Views

When you create a view, it will automatically be created with a default popup menu, which will enable users to control the attributes of the View, such as the action of the mouse buttons, and whether borders are displayed.

You can assign popup menus to view gadgets using the gadget's `setpopup()` method, with the name of the popup menu as the argument to the method. For example:

```
!MyForm.MyView.setpopup( !MyForm.pop1 )
```

Note: The standard graphical view forms in AVEVA products are normally supplied with fully customised popup menus, so you should not assign your own ones to these views

20.17.4 Defining Alpha Views

```
view .Input AT . . . ALPHA
 height 20 width 40
 channel COMMANDS
exit
view .InputOutput AT . . . ALPHA
 height 20 aspect 2.0
 channel REQUEST
 channel COMMANDS
exit
```

Borders and scroll bars cannot be turned off and the area they occupy is in addition to the size you specify for the gadget.

To define the **data channel** via which the gadget derives its alphanumeric content, use the `channel` command:

channel COMMANDS	causes the alpha view to have a command input field and displays command lines and error messages in the scrollable output region
channel REQUESTS	displays the output resulting from commands, in particular, queries in the scrollable output region

To set the keyboard focus so that keystrokes come to this gadget:

```
!This.InputOutput.SetFocus()
```

To clear the text output region of an alpha view use:

```
!This.InputOutput.clear()
```

It is not an error to have no associated channels when an alpha gadget is created.

Removing the Requests IO-channel from Alpha View Gadget

To delete the 'requests' channel from an alpha view gadget us the command:

```
!alpha.removeRequests()
```

This command will also dissociate the view from the current PDMS Requests IO-channel if necessary.

Thus **request output** will no longer go to this alpha view.

The method is valid only on a form that has been mapped to the screen, and will cause an error (61, 605) if used during form definition.

Note that you can add the channel back again by assigning to the gadget's .channel member:

```
!alpha.channel = 'Requests'
```

This adds the 'requests' channel to the gadget, and causes it to become the current PDMS *requests region*, if the form is mapped, and displaces any previous association.

When the user clicks on an alpha view it will automatically associate itself with the PDMS requests channel or the commands channels, or both, according to its current channel settings.

The Alpha view gadget supports multiple line copy and paste of text. The simple command window form below shows the appearance.

The bottom-most empty blank line is the command input line, where the user can type in, and execute the next command by pressing the 'enter' key.

The Alpha view allows the following operations:

1. Double-click of any displayed line will copy that line to the command input line ready for editing and execution.
2. You can use standard techniques to select text in the view and then use the right-mouse menu to copy your selection.
3. You can use the right-mouse menu to paste a selection into the command line. Each line of the pasted selection is executed in turn as if you had typed it in, except the last line, which becomes the command input line ready for editing and execution.
4. You can use the right-mouse menu to paste a selection as a macro. The set of pasted lines are captured to a temporary file which is then executed as PML macro. The selected lines are not echoed into the view's display.
5. Pasting of multiple lines is aborted if an error alert is displayed.
6. You can drag and drop a PML file into the view, and it will then execute as a PML macro.

7. The right mouse menu also allows you to clear the contents of the view and to change the size of text in the view's display.

20.17.5 Graphical Views

There are two types of Graphical view, namely **2D** and **3D**.

Different applications use **2D** and **3D** graphical views in different ways so each specific view gadget must be qualified by a view subtype which identifies its usage (or application).

The Forms and Menus system supports a standard **2D** view subtype PLOT view, which is a non-interactive view capable of displaying PDMS PLOT files.

Specific PDMS applications support their own view subtypes, for example:

- DESIGN uses **3D** view subtype **VOLUME** and **2D** view subtype **COMPARE**
- DRAFT uses 2D view subtype **AREA** and **3D** view subtype **VOLUME**

  Both **2D** and **3D** view types support a set of common members and methods. Some key ones are described below, before each subtype is dealt with.

For a full list of View members and methods, see the *Software Customisation Reference Manual*.

View Borders

Graphical views have optional sliders or scrollbars which allow modification of the view's geometric transformation. They can be turned on or off using the `.borders( !Boolean)` method. If the border is switched off, the actual display area expands to fill the available space.

Aspect Ratio and View Gadgets

Note when specifying the aspect ratio for a view that the corresponding ratios for ISO drawing sheet sizes are **0.7071** for portrait format and **1.414** for landscape format.

View Colours

Graphical views have two colours associated with them, namely the **background colour** and the **highlight colour**.

Setting Background and Highlight Colours

These can be set using the **Background** and **Highlight** members, either as integer colour indices or as colourname text strings:

```
!view.Background = 'black'
!view.Background = 2
```

The highlight colour is the colour used to highlight selected items in the view. You can set it by

```
!view.Highlight = 'white'
!view.Highlight = 1
```

Some view subtypes ignore this attribute.

Getting Background and Highlight Colours

You can get the background and highlight colour indices from the member variables or you can use the `Highlight()` and `Background()` methods that return the value of the respective property as a name string.

For instance, using the example above:

```
!colourname = !view.Background()
```

Would give us the string "`black`", and

```
!colourindex = !view.background
```

Would give us the integer colour index value 2.

Cursor Types

When the mouse cursor enters a graphical view, the View gadget determines what cursor type should be displayed for the user, or the representation to be assumed during a graphical interaction.

Some of he types of cursor available are:

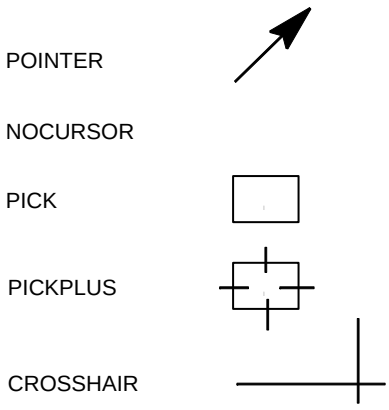


Figure 20:5. Available Cursor Types

Note: The initial cursor type for **VOLUME** views is a Pointer, and you cannot re-define this.

You can specify the initial setting for the cursor type in a **2D** view when you define the view gadget.

The default values for initial cursor types for PDMS **2D** view types are:

2D View Gadget	Cursor Type
PLOT	CROSSHAIR
DRAFT	CROSSHAIR
COMPARATOR	POINTER

You may use the `CURSortype attribute` command to set an initial cursor type within the gadget definition. For example:

```
Cursor  POINTER
```

20.17.6 Defining PLOT Views

```
setup form !!MyForm . . .
 ...
 view .Diagram  AT . . .  PLOT height 20 width 40
  ..
exit
define method !!MyForm( )
 -- form constructor method
 ...
 -- Initialise plot view gadget
 !this.diagram.borders = false
 !this.diagram.background( 'darkslate' )
 !this.diagram.add( 'plot1-1' )
 ...
endmethod
```

The area occupied by borders and scroll bars is within the area you specify for the gadget If the border is switched off, the actual display area expands to fill the available space.

When borders are turned off, the PLOT view becomes fixed and you cannot zoom or pan within it. This may be useful if you are using PLOT views as if they were paragraphs; for example, as decoration or for information.

To define the ***contents*** of the view specify the **PLOTFILE** pathname with the `.add()` method.

To remove a PLOT from the current PLOT view, use the `.clear()` method. This will leave the view empty. If you want to replace the displayed PLOT with another one then just re-use the `.add()` method with the new PLOT file.

20.17.7 Defining DRAFT's Area (2D) Views

```
setup form !!MyForm . . .
 ...
 view .DrawingAT . . .AREA
  height 20 width 40
  put /SITE1/SHEET
  limits 200 100 TO 600 500
 exit
 ...
```

```
exit
define method !!MyForm( )
 -- form constructor method
 ...
 -- Initialise AREA view gadget
 !this.drawing.borders = true
 !this.drawing.background( ‘beige’ )
 !this.drawing.highlight( ‘red’ )
 ...
endmethod
```

The area occupied by borders and scroll bars is within the size you specify for the gadget.

To define the colour items are displayed in when picked with the left-hand mouse button, use the `.sethighlight()` method with one of the standard DRAFT colour numbers or colournames.

The contents of the view may be any valid **2D** graphical element, such as a DRAFT **SHEET**, **VIEW**, **LIBRARY**, etc. To define the contents of the view, use the `put` command thus:

Command	Effect
`put CE`	Draws the current element
`put /SITE1/SHEET`	Draws the named Sheet element

If you put new content into an area view that already contains a drawing, the original content will be *replaced* by the new, *not* added to it.

Note: There is currently no `.put()` method! So to replace the view contents you will need to use the old PML1 syntax:
edit view !!MyForm.drawing AREA put /site/sheet2

The maximum limits for the drawing, such that only part of the specified content is drawn (giving, in effect, a window on the overall content area), may be set using the `limits` keyword as above or using the `.limits()` member in the form’s constructor method, as follows:

```
!box[1] = 200
!box[2] = 100
!box[3] = 600
!box[4] = 500
!this.drawing.limits = !box
```

where the limits define two opposite corners of the required area in sheet co-ordinates.

For a full list of **VIEW** members and methods, see the *Software Customisation Reference Manual.*

20.17.8 Defining DESIGN's Volume (3D) Views

```
setup form !!MyForm . . .
 ...
 view .ModelAT . . .VOLUME
  height 20 width 40
  limits auto
  look east
 exit
 ...
exit
define method !!MyForm( )
 -- form constructor method
 ...
 -- Initialise 3D view gadget
 !this.model.borders = false
 !this.model.background( 'darkslate' )
 !this.model.shaded( true )
 !this.model.projection( 'PERSPECTIVE' )
 !this.model.radius( 100 )
 !this.model.range( 500.0 )
 !this.model.direction( !dir )
 !this.model.eyemode( .false. )
 !this.model.through( !thru )
 !this.model.walkthrough( true )
 !this.model.step( 25 )
 ...
endmethod
```

The area occupied by borders and sliders is ***within*** the area you specify for the gadget.

To enable **colour-shaded** representation use the `.shaded()` method. By default a wireline representation `.shaded(false)` is used.

All aspects of the **3D** view transformation may be specified:

To enable **PERSPECTIVE** mode use the `.projection('PERSPECTIVE')` method. By default the projection is **PARALLEL**.

The view direction is controlled via the **3D** view's **.direction** member. This is a 3 element array of **REAL**, representing a direction vector in model space (**DEAST**, **DNORTH**, **DUP**). So to **look east** you would specify:

```
!dir[1] = 1.0
!dir[2] = 0.0
!dir[3] = 0.0
!this.model.direction = !dir
```

The 3D view gadget labels the view direction information in 'ENU' (East-West, North-South, Up-Down) format within the view status line and scroll borders, by default. The LABELSTYLE member allows the user to specify either 'ENU' or 'XYZ' labelling. In the 'XYZ' scheme the form of the view direction display in the status line is **x<bearing>y<elevation>z**, where the bearing angle is in the range [-180, 180] and the elevation angle is in the range [-90, 90]. The corresponding scroll bars are labelled "-Y -X Y X -Y" and "-Z 0 Z".

The centre of interest or **through point** of the **3D** view is controlled via the **.through** member. This is a 3 element array of **REAL**, representing a position in model space (**EAST**, **NORTH**, **UP**).

```
!thru[1] = 1000.0
!thru[2] = 5600.5
!thru[3] = 500.0
!this.model.through = !thru
```

The `.radius()` method specifies the radius of the view in current units - an imaginary sphere around the through point.

The `.range()` method specifies how far away from the Through point you want the Eye point to be.

To enable **WALKTHROUGH** mode specify `.walkthrough(true)` together with the walking step size in current units using the `.step()` method.

For a full list of View members and methods, see the *Software Customisation Reference Manual*.

Setting Aspects of the View Transformation by Syntax

The commands below allow aspects of the view transformation to be set using the old syntax:

```
edit view !!MyForm.Model VOLUME
```

followed by one of the following modifiers:

Modifier	Effect
ELEVATION EAST	Looks ***from*** the East.
LOOK EAST	Looks ***towards*** the East.
LOOK N45W75D	Looks ***towards*** specified direction.

Modifier	Effect
LOOK THROUGH E10 N50 U85	Looks ***through*** specified point.
LOOK THROUGH /A1	Looks ***through*** named elements origin.
LOOK THROUGH ID @	Prompts for cursor pick on Through-point.
LOOK THROUGH ID VALV @	Prompts for cursor pick on Valve for through-point.
LOOK FROM E10 N50 U85	looks ***from*** specified point (defines eye position)
LOOK FROM /A1	Looks ***from*** named element (as eye position).
LOOK FROM ID @	Prompts for cursor pick on from-point.
LOOK FROM ID VALV @	Prompts for cursor-on Valve for from-point.
LOOK TOWARDS E10 N50 U85	Looks ***towards*** the given direction.
LOOK TOWARDS /A1	Looks ***towards*** named element .
ISOMETRIC 3	Looks ***in specified isometric direction.***
PLAN	Looks down (default).

View Limits

Each volume view needs to know what region of the design model is to be displayed (the **3D** view limits). The DESIGN application has the notion of a current limits box. Unless you specify otherwise, the limits set for a new view gadget will default to the DESIGN current limits box (as set by, say, the `LIMITS` or `AUTOLIMITS` commands in DESIGN).

In addition, each volume view has an `AUTO` flag associated with it, which determines whether or not that view will be updated automatically whenever the DESIGN current limits box is changed.

To define the view limits and the way in which they are to be updated, use the old syntax:

```
edit view !!MyForm.Model VOLUME
```

followed by one of

Modifier	Effect
LIMITS AUTO	Display limits updated automatically to match current view limits (default).
LIMITS LOCAL	Display limits initially set to current view limits but not updated automatically.

Modifier	Effect
LIMITS LOCAL N1000 E2500 U0 TO N3000 E5000 U2000	Display limits initially set to specified positions and not updated automatically.
LIMITS LOCAL /EQUI2 /EQUI4	Display limits initially set to enclose the specified elements and not updated automatically.

You can set the limits box for a view, subject to the current AUTO flag using the .limits() method:

```
-- limits brick set as array of REAL [E1, E2, N1, N2, U1, U2]
!brick[1] = 1000.0
...
!brick[6] = 1560.4
!this.model.limits( !brick )
```

In DESIGN you may change the current limits box at any time by using one of the command formats

```
AUTOLIMITS N1000 E2500 U0 TO N3000 E5000 U2000
AUTOLIMITS /EQUI2 /EQUI4
```

Any volume views which have LIMITS AUTO set will be redrawn to match the new limits.

Mousemode

3D views have a **mousemode** member which can be set to Zoom, Pan, Rotate, or Walk

Saving and Restoring 3D Views

3D views have two methods which are used to save and restore up to four views, until the user leaves PDMS DESIGN:

```
!!view.saveview( !n )
!!view.restoreview( !n )
```

Where 1 <= !n <= 4.

21 Alert Objects

The **ALERT** object provides a simple way of giving a warning or getting a response from the user, avoiding the necessity of creating a separate form. Where an *alert* needs a simple acknowledgement by the user, this can be achieved by pressing the spacebar or the **Enter** key on the keyboard.

Note: Alerts are always **blocking**: they prevent interaction with any other form until they have been actioned.

Below are some examples of how you can define different kinds of *alerts*:

Code	Effect
`!!Alert.Error( 'You cannot do this!' )`	Defines a simple **error alert** with an acknowledge button at the current cursor position: Note that the result, which is always **'YES'**, has here been discarded
`!!Alert.Message( 'Saving your data now')`	Displays a **message alert** with an acknowledge button at the current cursor position:
`!!Alert.Warning( 'Do not press this button again!'`	Displays **a warning alert** with an acknowledge button at the current cursor position:
`!Answer = !!Alert.Confirm( 'You can't be serious!' )`	Displays a **confirm alert** at the current cursor position. This example comes up with two buttons: '**YES'** and '**NO'** , delivering the corresponding text as the string result.

Code	Effect
`!Answer = !!Alert.Question( 'OK to format entire disc?' )`	Displays a **question alert** at the current cursor position: This example comes up with three buttons: '**YES'**, **'NO'** and '**CANCEL'** delivering the corresponding text as the string result.
`!Answer = !!Alert.Input( 'Type Item code: ','P10101' )`	Displays an **input alert** at the current cursor position: The first string appears as the prompt, and the second string is the default entry in the text box if the user does not supply a value.

21.0.1 Position of Alerts

By default, an Alert is automatically positioned when it is displayed so that it is under the cursor. Optionally, two more arguments may be supplied to specify the screen position at which an *alert* is to appear:

```
!!Alert.Error( 'You cannot do this!' , 0.25, 0.1)
```

This shows the *alert* with its origin one quarter of the way across from the left hand side of the screen and one tenth of the way down from the top.

21.0.2 Input Alerts

This allows the user to obtain textual input from the operator via a blocking alert which overrides all other interactive activities. The alert can be summoned by the alert methods:

```
!!Alert.Input( !prompt is STRING, !default is STRING) is STRING
```

```
!!Alert.Input( !prompt is STRING, !default is STRING, Xpos is REAL, Ypos is Real) is STRING
```

!prompt is displayed to prompt the operator and **!default** is offered as the default value in the alert's text input field.

Xpos, **Ypos** define the alert's top left hand corner in normalised screen co-ordinates.

Note: This function has been added to provide the GUI equivalent of the command

```
Var !x read
```

which is only valid in **tty mode**.

To achieve flexible, user-friendly interfaces, the input alert should be used sparingly because of its blocking nature.

22 FMSYS Object and its Methods

The **FMSYS object** (Forms and Menus System object) is a mechanism for providing methods which are not specific to the standard objects, such as forms, gadgets, menus etc. The **FMSYS** methods perform a variety of system-oriented tasks.

22.1 Setting the Default Format of Text Fields

The SetDefaultFormat method is used to provide a default global format object to be used to format typed text fields when no specific format is defined at design time. The SetDefaultFormat method can only be called once to avoid inconsistency across appware.

Call the method as follows:

```
.SetDefaultFormat(!!fmt is FORMAT)
```

where !!fmt must be a global variable.

After calling the method the value can later be retrieved by referencing the DefaultFormat() method as follows:

```
!text=!myVar.String(!!fmsys.DefaultFormat())
```

22.2 Querying Forms & Menus System Information

The `FMinfo()` method returns an array of all current Forms and Menus' information strings:

```
!info = !!FMSYS.FMinfo()
```

where **!info** is an array of **STRING**, currently with seven entries:

Array Index	String
[0]	Module's name
[1]	Module's version
[2]	AppWare version
[3]	Not used
[4]	Help file alias string
[5]	Not used
[6]	Help about callback string

Other methods on the **FMSYS** object are:

Method	Purpose
!form = !!FMSYS.main ()	Returns a variable of type form, which is unset if there was no main form
!formsarray = !!FMSYS.forms()	Queries all user-forms (excludes main form and any system forms. Returns an array of **FORM.**
!formsarray = !!FMSYS.shownForms()	Queries all user-forms which are currently displayed (excludes main form and any system forms).Returns an array of **FORM**.

22.3 Swapping Applications

You can swap to a new application using the `SetMain` method:

```
!main = !!FMSYS.SetMain( !newmain )
!!FMSYS.SetMain( !main )
```

22.4 Progress and Interrupt Methods

Form and Gadget callbacks may take a long time to execute, so it is often desirable to use the Progress Indicator field on the Main Window's status bar (along the bottom) to indicate continuing progress to the user. Additionally for some callback operations it may be possible and desirable to allow the user to interrupt the operation and choose to cancel it. The FMSYS methods Progress, setProgress, Interrupt and setInterrupt allow this to be achieved.

Your callback which provides the desired operation, must be cyclical so that there is some repeated access point where you can report progress (approximately as a percentage of the total task), and check to see if the user has clicked a stop button which you have provided on some displayed form.

In your callback function you must first notify the system of the identification of the stop button using the **FMSYS** method `SetInterrupt`:

```
!!FMSYS.setInterrupt(!!fmstop.stopButton )
```

Then at an appropriate point in the callback, check to see if the stop button has been pressed using the **FMSYS** `Interrupt` method, and exit with a message if it has:

```
-- Initialise progress bar
!!FMSYS.setProgress(0)
do !next from 1 to 50
 ...
 !!RoutePipe( !next, . . . )    $*Route next pipe
 ...
```

```
    -- Update the progress bar - first update the percentage
    completion
    !percent = ...
    !!FMSYS.setProgress( !percent )
     -- Check if user pressed interrupt button
     if ( !!FMSYS.Interruppt() ) then
     return error 1 'Processing aborted'
      endif
    enddo
```

Following is the PML code for a simple interrupt form !!fmstop.pmlfrm:

```
$* F&M test harness: Stop form for interrupt management
setup form !!fmstop dialog NoAlign
    title 'STOP (!!fmstop)'
    path down
    para   .stopText  text |Press to quit task|  width 20
    button .stopButton |Quit| width 25 height 2
exit
define method .fmstop()
    --Constructor
    -- define callbacks
    !this.firstShownCall = |!this.stopButton.background =
     'red'|
endmethod
```

22.5 Refreshing View Gadgets

You can refresh all View gadgets using the `Refresh` method:

```
!!FMSYS.Refresh()
```

22.6 Checking References to Other Forms

By default, when a form is loaded, all the references from it to other forms are checked to make sure that the other forms exist. If you are experiencing performance problems, you can switch the reference checking off using the `CheckRefs` method:

```
!!FMSYS.CheckRefs( true )
```

Using this method can significantly improve performance.

22.7 Splash Screen

There are occasions when the PDMS entry screen, known as a splashscreen, may be left on the screen when PDMS fails to start up, for example, if it has not been correctly installed, or there are network problems.

To ensure that this screen is removed from the display, you can use the `SplashScreen` method:

```
! !FMSYS.SplashScreen( false )
```

22.8 Default Form Positioning

By default, **MDI forms** are placed at the maximum (right-hand side) of the main PDMS window and dialog forms are placed at the minimum (left-hand side) of the PDMS window.

To switch off default positioning, use the `DocsAtMaxScreen` method:

```
!!FMSYS.DocsAtMaxScreen( false )
```

This method may be useful for wide screen and twin screen devices

22.9 CurrentDocument() Method

This method returns the current Document of the application framework as a **FORM** object. If there is no current document then the returned form has value **Unset**.

It has the following declaration:

```
!!FMSYS.currentDocument( ) is FORM
```

This method should be useful when constructing callbacks for menus or gadgets which are not on the current document form, i.e. where the context is not the current form or view gadget, e.g. callbacks from the Main menu system or from pullright menus on *Addin* forms.

```
!myDocument = !!FMSYS.currentDocument()
```

22.10 LoadForm() Method

The method `LoadForm` allows force loading of a form definition and/or the ability to get a reference to a form object by name.

The method's signature is:

```
!!Fmsys.loadForm( !formname is STRING ) is FORM
```

If the form exists then the method returns a reference to the form object. If it doesn't exist, then an attempt is made to force load its definition.

If this fails then an unset form reference is returned. You can check for this using the `Unset` method on the form object.

22.11 Cursor Function Support

The F&M system prevents a cursor function from being executed if there are no displayed graphical views which support cursor functions, and raises the usual 'quit' error (61, 528):

'`User exit from submode`' so that Appware can handle the effective 'quit', and then raises error (61, 115): '`Invalid command when GRAPHICS OFF or no suitable view available`', which will alert the user unless trapped by the Appware. However, only the Appware knows exactly which graphical views can handle the specific cursor command, so the OKCurfnView methods have been provided to allow Appware to use this knowledge intelligently.

Methods provide are:

Query whether graphical views of the specified view type are displayed. Graphical view types supported are: 'G2D'; 'G3D'; 'ANY' and any view subtype is implied.

OKCurfnView(!viewtype is STRING) is BOOLEAN

Query whether graphical views of the specified view type and subtype are displayed. Graphical view types supported are: 'G2D'; 'G3D'; 'ANY'. View subtypes supported are: 'ANY' and for

G2D: 'NORMAL' (Draft); 'PLOT'; 'ISOSPOOL'

G3D: 'NORMAL' (Design)

OKCurfnView(!viewtype is STRING, subtype is STRING) is BOOLEAN

Example using FMSYS OKCurfnView methods:

The following code snippet shows an example of use within a form method:

```
define method .runCurfn( !curfn is string )
    -- run specified cursor function
    if( !!fmsys.OKCurfnView('g2d', 'normal') ) then
       -- valid view exists, so run cursor function
       $!curfn@
    else
       -- no valid view
       return error 1 |No suitable displayed 2D views for
       cursor function '$!curfn$@'|
    endif
endmethod
```

Notes:

1. OKCurfnView('g2d') is equivalent to OKCurfnView('g2d', 'ANY')
2. OKCurfnView('ANY ', 'NORMAL') will look for both 2D Draft and 3D Design views

23 PML Add-ins

A PML add-in can:

- appear as an application on the main menu bar;
- define menus and toolbars;
- add menu fields and toolbars to applications;
- remove menu fields from applications;
- cause functions to be run when starting or switching between applications.
  Each existing AVEVA application is implemented as a PML add-in.

It is important to distinguish between an add-in that defines an application and one which just modifies an existing application. An application add-in must have an entry in the Applications main menu so that it can be switched to.

This chapter describes mechanisms to allow a number of cooperating applications to be added into a module. Each AppWare application is a PML Add-in. Users can add their own applications or add additional functionality to applications using PML Add-ins.

Warning: These mechanisms are still being developed and so may change at future releases of PDMS. Even though we will make every effort to preserve the validity of added-in functional code, we recommend you isolate the definition of PML Add-ins from the functional code of your applications to minimise the impact of any such changes.

It is important to distinguish between an add-in that defines an application and other add-ins. An application appears on the Applications menu and can be switched to, whereas an add-in simply provides a means of adding functionality to a module.

23.1 Application Switching

The application switching mechanism makes use of the following objects to control applications, toolbars, forms, and menus.

Object	Description
appCntrl	Controls applications' switching
appDefn	Each application has an **appDefn** object that stores information about that application.
appTbarCntrl	Controls visibility of toolbars and which toolbars are active in each application

Object	Description
appMenuCntrl	Controls which menus and menu fields are available in each application
appFormCntrl	Controls those forms which register themselves

A feature of **Application Switching** is **serialisation**, which allows the working environment to be saved and then restored to its previous state on entry to PDMS; that is, the current application is stored and reloaded on entry. The visibility and position of toolbars and forms is also preserved.

The new system of add-ins makes it unnecessary to swap the main form when changing between applications. Instead, the menu items and toolbars are all contained on one main form; they are then shown only when the appropriate application is active.

23.1.1 Main Form

The main form is now stored as a PML Form.

For DESIGN, this **is !!appDesMain** and for DRAFT it is **!!appDraMain**. These contain the basic gadgets and menu-options common to all applications in the module. All others are created using add-ins.

We recommend that you create add-ins to modify the main form rather than editing the standard product code.

23.1.2 Callbacks

Each application can now have different types of callback, which are run at different times:

Callback	When Executed
module startup calls	Runs when PDMS first starts.
startup calls	Runs the first time the application loads.
switch calls	Runs every time PDMS switches into the application.

23.1.3 Defining an Add-in

Each add-in is specified by an add-in definition file, with pathname `%PDMSUI%\<module>\ADDINS\<addin>`, where `<module>` is the module name e.g. `DES`, and `<addin>` is the addin name e.g. `MYADDIN`. This is similar in format to the entries in the old `APPLICATIONS` file. Each line contains a key and description, separated by a colon.

Some keys are available to all add-ins; others can be used only for applications, since they refer to menu text, gadgets and callbacks specific to that application.

Keys Available to all Add-ins

The following can be used by all add-ins.

Key	Description
Name	The unique identifier of the add-in, e.g. GEN, DES. This key is compulsory.
Title	Text description of the add-in (if the add-in defines an application, the title appears in the title bar of the main window when the application is loaded)
ShowOnMenu	Determines whether the add-in has an entry on the Applications menu, i.e. whether the add-in defines an application (false by default).
Object	Add-in object: user-defined object type used to help define the add-in, e.g. **APPGENERAL**
ModuleStartup	Callback run when the PDMS module first starts
StartupModify	Name of application to modify and the callback run when an application is first started, separated by a colon. e.g. `EQUI:!!equiAppStartupCall().`
SwitchModify	Name of application to modify and the callback to run when the application is switched to, separated by a colon. e.g. `PIPE:!!pipeAppSwitchCall()`

The following keys are only used for applications, i.e. add-ins that have a menu option on the Applications menu and can be switched to.

Key	Description
Menu	Entry for application on the applications menu (the title is used if this isn't specified)
Directory	Name of the application directory under %PDMSUI%\module
Synonym	Synonym created for the directory (only used if the directory is specified)
Group	Group number (applications with the same group number are put into a submenu together)
GroupName	Description of submenu to appear on main menu

Key	Description
Gadgets	Integer specifying the gadgets that appear on main form for this application: 1 - no gadgets 2 - sheet/library gadgets 3 - sheet/library and layer gadgets 4 - sheet/library and note/vnote gadgets
Type	Type of application for user applications. Valid values are DRA, DIM, LAB, USR1, USR2, USR3, USR4, USR5.
SwitchCall	Callback to be run every time the application is switched to.
StartupCall	Callback run when the application first starts.

To make it easier for user-defined add-ins to modify existing applications, it is possible for an add-in to add a startup or switch call to an application. You can do this by adding the lines of the following form to the add-in file.

```
startupModify: APPNAME: callback
switchModify: APPNAME: callback
```

where **APPNAME** is the name of the application to modify and callback is the callback to be assigned to it.

If an application with name **APPNAME** is not defined, the callback is not run.

23.1.4 Add-in Object

A user-defined object type, the add-in object, is used to define toolbars and menus for an add-in. An instance of this object is created and its methods are run at specific points.

The method `.modifyForm()` of each add-in object is called during definition of the main form so that add-ins may create their own toolbars on the main form; `.modifyMenus()` is called when starting PDMS to create menu options.

For applications, `.initialiseForm()` is called when switching to the application.

It is not mandatory for all these methods to be present, and if one is not present, execution continues as normal. It is possible to specify no object if you wish only to use the other properties of the add-in.

23.1.5 Initialisation

The add-in definitions are loaded from file during PDMS startup using the function `!!appReadAppDefns(!module is STRING)`, where **!module** is the 3-letter identifier for the module.

This creates the global **!!appCntrl** object, which holds information about the add-ins.

After reading the definitions of all add-ins**,** **!!appCntrl** assigns all callbacks to the applications.

For backward compatibility, if an add-in is an application, a directory is specified for the application and the file `module\directory\INIT` exists, then this file is added as a

startup-call. Similarly, if the file `module\directory\ISYSTEM` exists, it is added as a switch call.

23.2 Menus

Each add-in is able to specify menus that can be shown, along with the application in which they can be shown. Menus are created by running the `.modifyMenus()` method of the add-in object when PDMS first starts.

23.2.1 APPMENU Object

The **APPMENU** object provides a means of adding menu fields to menus using the usual PML 2 syntax while allowing the menu controlling object to retain information about the fields created. This information is used to show and hide menu fields depending on the application.

23.2.2 Addition of Menu Items

To add menu fields to an application, proceed as follows:

Create an **APPMENU** object:

```
!menu = object APPMENU('menuName')
```

This associates the object with the menu named menuName on the main form, creating the menu if it does not already exist.

Add items to the menu using the methods of the **APPMENU** object.

The arguments for these methods are exactly the same as the menu creation functions provided by PML. A unique fieldname for each field in a menu can be specified.

You may specify a field name when adding menu items in this way; if no field name is specified, then the **Dtext** of the menu field is used as the field name. To add to the end of the menu, use the commands

```
!menu.add('<FieldType>', '<Dtext>', '<Rtext>', {'<FieldName>'})
!menu.add('SEPARATOR', {'<FieldName>'}).
```

Note: If a menu field with the same field name is already present, the field will not be added. This is so two add-ins can define the same menu field, and it will appear independently of whether one or both add-ins are present.

This method does not always add an item to the end of the menu. If a function tries to add a menu field with the same field name as one that's already present, the next menu field created (unless it's inserted explicitly before or after another field or it's a separator) will appear after the existing field. The reason for this is to preserve the intended order of the fields.

To insert a new field relative to existing named field, use

```
!menu.insertBefore('<TargetFieldName>',          '<FieldType>',
'<Dtext>', '<Rtext>' {, '<FieldName>'})
!menu.insertAfter('<TargetFieldName>',           '<FieldType>',
'<Dtext>',  '<Rtext>' {, '<FieldName>'})
```

```
!menu.insertBefore('<TargetFieldName>',      'SEPARATOR'      {,
'<FieldName>'})
!menu.insertAfter('<TargetFieldName>',       'SEPARATOR'      {,
'<FieldName>'}).
```

To register the **APPMENU** object with `!!appMenuCntrl`, specifying the applications in which the menu field should be visible, use:

```
!appMenuCntrl.addMenu('APP', !menu)
```

It is possible to add menu fields from an **APPMENU** object to every application. The application name `'ALL'` has been reserved for this purpose.

23.2.3 Removing Menu Items

The **APPMENU** object can also be used to remove menu fields from an application. The method `.remove(!fieldName is STRING)` hides the field with unique field name **!fieldName**.

The following commands can be used to remove the field named **removeField** from the menu **menuName** in the general application.

```
!menu = object APPMENU('menuName')
!menu.remove('removeField')
!!appMenuCntrl.addMenu(!menu)
```

23.2.4 Modifying the Bar Menu

The **APPBARMENU** object provides similar functions for the bar menu of the main form. To create additional menus on the bar menu, proceed as follows:

Create an **APPBARMENU** object:

```
!bmenu = object APPBARMENU()
```

This automatically associates the object with the bar menu on the main form.

You can then change the contents of the bar menu using the following commands.

Command	Description
!bmenu.add(!dtext is STRING, !menuName is STRING)	Adds a menu to the end of the bar menu, use the command. This adds the menu **!menuName** to the bar menu with label **!dtext**. The menu is automatically positioned before the Window and Help menus if they are present.
!bmenu.insertBefore('<TargetMenuName>', '<Dtext>', '<MenuName>') **!bmenu.insertAfter('<TargetMenuName>', '<Dtext>', '<MenuName>')**	To insert menus relative to existing menus, use

Command	Description
!bmenu.remove(!menuName is STRING	To remove the menu **menuName** from the bar, use
!!appMenuCntrl.addBarMenu('APP', !bmenu)	Register the **APPBARMENU** object with `!!appMenuCntrl`, specifying the applications in which the menu should be visible.

23.3 Toolbars

Toolbars are defined by creating a frame of type toolbar on the main form, using the following syntax.

```
frame .toolbarName toolbar 'Toolbar description'
 [PML commands to define buttons, toggle gadgets, option gadgets and text gadgets]
exit
```

It is not possible to add toolbars to a form once it has been created, so all toolbars must be defined before any application is started. The method `.modifyForm()` of the add-in object is run during building of the form so that each add-in can add its own toolbars.

23.3.1 Toolbar Control

Toolbar visibility in different applications is controlled by the object **APPTBARCNTRL**. A global instance **!!appTbarCntrl** is created when starting PDMS.

Each toolbar is registered with a list of applications in which it can be active. **!!appTbarCntrl** then controls the active state and visibility of the toolbar as applications are switched. Toolbars can be shown and hidden using the popup menu which appears when you right click on a toolbar or menu bar of the main form. If a toolbar is shown in one application, then it will be shown in all other applications for which it is registered.

To register a toolbar, use the command

```
!!appTbarCntrl.addToolbar(!toolbarName is STRING, !appName is STRING, !visible is BOOLEAN),
```

where **!toolbarName** is the name of the toolbar frame**, !appName** is the name of the application to register the toolbar in and **!visible** is whether the toolbar is visible by default.

Instead of registering a toolbar in multiple applications individually, an array of applications can be specified:

```
!!appTbarCntrl.addToolbar(!toolbarName is STRING, !appNames is ARRAY, !visible is BOOLEAN)
```

registers the toolbar for all applications in the array **!appNames**.

The application name `'ALL'` has been reserved so that toolbars can be registered in every application. To register a toolbar in all applications, use

```
!!appTbarCntrl.addToolbar(!toolbarName is STRING, 'ALL', !visible is BOOLEAN)
```

where **!toolbarName** is the name of the toolbar and **!visible** is whether it is visible by default.

The command

```
!!appTbarCntrl.addToolbar(!toolbarName is STRING)
```

is a shorthand way of registering the toolbar to be visible in all applications, making it visible by default.

Any toolbars on the main form that are not registered are automatically detected and hidden by default. They are registered to be active in all applications.

23.3.2 Removing Gadgets from a Toolbar

It is possible to show a toolbar in an application but only show some of its gadgets. In this case, the user must register which gadgets are hidden in that application.

To do this, use

```
!!appTbarCntrl.hideGadgets(!gadgets is STRING, !appName is STRING),
```

where **!gadgets** is a space-separated list of the gadgets to hide and **!appName** is the name of the application in which to hide them.

23.3.3 Deactivating Gadgets on a Toolbar

To grey out gadgets on a toolbar in a given application, use

```
!!appTbarCntrl.deactivateGadgets(!gadgets is STRING, !appName is STRING),
```

where **!gadgets** is a space separated list of the gadgets to inactivate and **!appName** is the name of the application in which to hide them.

23.4 Forms

The **APPFORMCNTRL** object allows control of the behaviour of forms on application switching and re-entering PDMS. A global instance **!!appFormCntrl** is created when PDMS starts.

If a form is to be controlled, it must register itself with the object during the constructor method.

23.4.1 Registering a Form

The most complete syntax for registering a form is

```
!!appFormCntrl.registerForm(!form is FORM, !visibleInApps is ARRAY, !requiredApps is ARRAY, !showOnStartup is BOOLEAN, !showOnSwitch is BOOLEAN)
```

where **!form** is the form to be registered, **!visibleInApps** is an array containing the applications the form should be visible in, **!requiredApps** is the applications that must have been loaded in order for the form to be displayed, **!showOnStartup** is whether the form is re-shown on entry to PDMS if it was shown on exit and **!showOnSwitch** is whether the form is re-shown when switching to an application if it was previously shown in that application.

If **!visibleInApps** is an empty array, then the form will be visible in all applications.

However, this can be simplified for most forms that do not need all this information stored about them.

The command:

```
!!appFormCntrl.registerForm(!form is FORM)
```

registers the form so that it is re-shown on startup if it is shown when PDMS exits. It does not depend on any other applications and its visibility is not otherwise controlled.

23.4.2 Hiding Forms when Exiting Applications

Previously, the main form was swapped when switching applications. Forms opened as children of the main form were hidden when switching out of an application.

Since the main form is no longer swapped when switching applications, this no longer occurs so **!!appFormCntrl** controls the hiding of forms. When switching between applications, if a form is shown as a child of the main form and is not registered to be shown in the new application, then it is hidden. (Descendents of this form will therefore also be hidden.)

Dockable forms are shown as free forms, so will be not be closed. However, the effect can still be achieved by registering the form in just one application with the argument **!showOnSwitch** set to false:

```
!visibleInApps[1] = 'APP'

!!appFormCntrl.registerForm(!form, !visibleInApps, ARRAY(),
false, false)
```

The form will be hidden when switching from the application `'APP'` and will not be shown next time the application is started.

23.4.3 Storing Data Between Sessions

APPFORMCNTRL also provides a means for forms to store data so that they can restore their state next time the module is loaded. For each form that is registered, **APPFORMCNTRL** calls the `.saveData` method of the form when leaving PDMS: the form can return an array of strings of data to be stored. When the form is next shown, it can retrieve this data using the following method:

```
!data = !!appFormCntrl.getFormData(!form is FORM).
```

The contents of the array **!data** will be the strings in the array returned by the `.saveData` method when PDMS exited. If no data has been stored for that form, it will be an empty array

You can also use the command:

```
data = !!appFormCntrl.getFormData(!formName is STRING)
```

if only the name of the form that stored the data is known.

Note: The form must be able to cope if no data is returned. This will happen the first time PDMS is run or if the form was not open the previous time the user exited PDMS

The following example shows a form that stores the state of some of its gadgets between sessions.

```
setup form !!testForm
```

```
 title 'Test of form data storage'
 text .name 'Name' width 10 is STRING
 toggle .toggle 'Toggle'
exit
define method .testForm()
 -- register form with form controlling object
 !!appFormCntrl.registerForm(!this,  ARRAY(),  ARRAY(),  true,
true)
 -- retrieve data if available
 !data = !!appFormCntrl.getFormData(!this)
 -- restore name and state of toggle if data present
 if (!data.empty().not()) then
  !this.name.val = !data[1]
  !this.toggle.val = !data[2].boolean()
 endif
endmethod
define method .saveData() is ARRAY
 !return[1] = !this.name.val
 !return[2] = !this.toggle.val.string()
endmethod
```

23.5 Converting Existing User-defined Applications

Previously, applications for each module were defined by an entry in the `module\DFLTS\APPLICATIONS` file. A separate main form for each application was built using the `module\GEN\FSYSTEM` file; application-specific gadgets and menus were added by calling the `module\application\DBAR` file from the `FSYSTEM` file.

In order that new applications can be added more easily and independently of each other, application definitions are now stored in the directory `module\DFLTS\APPDEFNS`. There is one file per application, which uses the same format as the old `APPLICATIONS` file with the extensions discussed previously.

23.5.1 Replacement of the DBAR File

Each add-in has a corresponding object, the type of which is specified in the add-in file, that adds toolbars and menu entries for that add-in. The functionality of the `DBAR` file has been replaced by the methods of the add-in object.

The method `.modifyMenus()` of the application object is called to add items to the menus. This replaces all sections of the `DBAR` file where menu fields are added. Menu fields must be added using an **APPMENU** object - see *Menus* - otherwise they will be visible in all applications. The **APPMENU** object must then be registered with the application in which the menu fields are to be visible, otherwise they will never be shown.

The method `.modifyForm()` is called to make modifications to the main form, specifically to add toolbars, during form definition. This replaces the label /DATA section of the DBAR

file. Exactly the same syntax can be used here, but if you wish alter any gadgets after they have been created you must refer to the main form explicitly rather than using the keyword **!this**, which will refer to the object.

Toolbars can be registered - see *Toolbar Control* - any unregistered toolbars will be hidden by default and will not be controlled.

23.5.2 Menu Name Clashes

Menu names and menu field names must be checked against existing applications to ensure there is no clash. To avoid clashes, it is recommended that each user-defined application prefixes the names of the menus it creates and the fields it creates on existing menus with the name of the application.

23.5.3 Converting the DBAR File

In order to use an existing application, the DBAR file must be converted from the existing format. If the DBAR file contains only PML 1 syntax for creating menu items, the object **CONVERTDBAR** can be used to create the corresponding add-in object.

To convert an existing DBAR file, create a **CONVERTDBAR** object with the following parameters:

```
!convert = object CONVERTDBAR(!module is STRING, !appNameStr is
STRING, !outputDir is STRING, !addinFileDir is STRING)
```

where **!module** is the name of the module in which the add-in is used, **!appNameStr** is the name of the application (e.g. 'PIPE'), **!outputDir** is the directory in which to put the add-in object and **!addinFileDir** is the directory in which to put the add-in file.

Then run the `.convert()` method of the object to perform the conversion:

```
!convert.convert().
```

This creates the add-in object and corresponding add-in file. Information about the existing application is stored in the %PDMSUI%\module\DFLTS\APPLICATIONS file. This information is read and used to create the add-in file. Therefore, the application being converted must have an entry in the APPLICATIONS file in the current PDMSUI path.

23.6 Example Application

23.6.1 Adding a Menu Field

It is simple to use a PML add-in to add a field to a menu in PDMS. The following example shows how to add an extra field to the Query menu in the *Design Pipework* application.

Create an **APPMENU** object corresponding to the menu.

```
!menu = object APPMENU('sysQry')
```

Add the required field to the menu

```
!menu.add('CALLBACK', 'Query Owner', 'q owner', 'QueryOwner')
```

Register the **APPMENU** object with **!!appMenuCntrl**, so the menu is visible in the Pipework application

```
!!appMenuCntrl.addMenu(!menu, 'PIPE')
```

The same method can be used to add fields to any menu descended from the bar menu of the main form.

A sample add-in object definition, which must be put in the PMLLIB path, is shown below.

```
define object APPADDQUERYMENUFIELD
endobject
define method .modifyMenus()
 !this.queryMenu()
endmethod
define method .queryMenu()
 -- define APPMENU object associated with the Query menu
!menu = object APPMENU('sysQry')
 -- add field to query the owner of the current element
 !menu.add('CALLBACK', 'Query Owner', 'q owner', 'QueryOwner')
 -- register the APPMENU object to be visible in the Pipework
 -- application
 !!appMenuCntrl.addMenu(!menu, 'PIPE')
endmethod
```

The corresponding add-in definition file, which is put in the DES\DFLTS\ADDINS directory, contains:

```
Name: ADDQUERYMENUFIELD
Object: APPADDQUERYMENUFIELD
```

23.6.2 Creating a Custom Delete Callback

Users may want to replace the callback for the **CE** option on the **Delete** menu in *Design* with a customised callback. This can be done using an add-in, without modifying the AVEVA supplied files. To apply the changes in a new version of PDMS, the files can simply be copied across.

Create an **APPMENU** object corresponding to the **Delete** menu.

```
!menu = object APPMENU('sysDel')
```

Hide the menu field **CE** on this menu

```
!menu.remove('CE')
```

Insert a new menu option to replace it

```
!menu.insertAfter('CE', 'CALLBACK', 'Delete',
'!!customDelete()', 'customDeleteCE')
```

Register the object with **!!appMenuCntrl** so the menu item is replaced in all applications.

```
!!appMenuCntrl.addMenu(!menu, 'ALL')
```

The callback of the **Delete** button on the main toolbar can also be modified.

```
!!appDesMain.mtbDeleteCe.callback = '!!customDelete()'
```

A sample add-in object definition, which must be put in the PMLLIB path, is shown below.

```
define object APPCUSTOMDELETE
endobject
define method .modifyForm()
 -- change callback on Delete toolbar button
 !!appDesMain.mtbDeleteCe.callback = '!!customDelete()'
endmethod
define method .modifyMenus()
 !this.deleteMenu()
endmethod
define method .deleteMenu()
 !menu = object APPMENU('sysDel')
 -- replace CE option on menu with custom delete function
 !menu.remove('CE')
 !menu.insertAfter('CE', 'CALLBACK', 'CE', '!!customDelete()',
'customDelete')
 -- we want the menu to be replaced in all applications, so
 -- register this object in all applications
 !!appMenuCntrl.addMenu(!menu, 'ALL')
endmethod
```

The corresponding add-in definition file, which is put in the `DES\DFLTS\ADDINS` directory, contains:

```
Name:CUSTOMDELETE
Object:APPCUSTOMDELETE
```

A Core Managed Objects

This material is chiefly for AVEVA developers and not users, because it requires cooperation between AppWare and core-software.

The mechanisms described below allow PML defined forms, menufields and gadgets which are subsequently managed by core code. It should be noted that the use of core code managed forms, menufields and gadgets requires well planned co-operation between software and AppWare.

It is permissible to have a mixture of PML and CORE managed gadgets on a PML managed form.

A CORE managed form may also have some PML managed gadgets and menufields, but extreme caution should be adopted in this case. Making a form core managed involves considerable effort in the supporting core code, as it bypasses most of the F&M form management support.

A.1 Form Core Support

PML defined forms now have a new qualifier attribute ‘control type’, with values PML or CORE. A form is PML managed by default, but can be declared core managed by means of the ‘CORE’ keyword in the FSETUP command. This is applicable to any form type except **MAIN** forms, for example:

```
SETUP FORM !!myform blockingdialog resi CORE
```

When a form is first displayed, if it is declared as core managed, then F&M builds the DRUID widget, and then notifies core code so that it can take over the form’s management. **This is guaranteed to precede any core gadget or core menu notifications.**

Core code may plug in its own callback functions to DRUID for certain form events which will then no longer go to F&M.

Typically core code will need to manage the following:

- Form UI_QUIT event.
- The **OK**, **CANCEL**, and **APPLY** buttons if present.
- Showing and hiding the form.

In general the PML form definition for a core managed form should **not** define the following PML callbacks:

- INITCALL
- AUTOCALL
- OKCALL
- CANCECALL.

If **OK**, **CANCEL** or **APPLY** buttons are present they should also be declared as CORE managed.

A.1.1 FORM Core Code Interface

F&M will call the following routine, which must be provided by the core code, to notify core-code of each form with **CORE** control type:

```
CORFRM ( int FORM, int WMID, int FTYPE, int LSTR, string FORMNAME )
```

where:

- **FORM** is the F&M handle for the form element
- **WMID** is the **DRUID** driver handle for the form. This will allow the core-code to provide its own callbacks for the form, and to organise the form's core managed gadgets and menus.
- **FTYPE** is the form's type (**DIALOG**, **BLOCKINGDIALOG**, **DOCUMENT**).
- **FORMNAME** is a name-string unique to the application-module. It has a length of **LSTR.**

The following F&M routines will be useful for core code management of PML forms with core managed gadgets:

Command	Effect
`FLDFRM ( int LNAM, string FNAME, int FORM )`	Load named form from a definition file. **FORM** is a handle to the created form object.
`FDSCOR ( int FORM, int IERR)`	Display the given form. **IERR** is a PML error token. It takes the value 0 when there was no error.
`FHDCOR ( int FORM, boolean CANCEL )`	Hide the given form. If `CANCEL=true` it means the form was cancelled. Otherwise it was **OK**'ed

A.2 Gadget Core Support

PML defined forms may now have a mixture of PML managed and core code managed gadgets.

A new gadget qualifier attribute 'control type', with values PML or CORE has been introduced.

The following gadgets can be declared to be CORE controlled or PML controlled during their definition:

- **BUTTON**
- **TEXTIN**
- **TOGGLE**
- **PARAGRAPH** (doesn't have a callback)
- **TEXTPANE** (doesn't have a callback)

- **FRAME**
- **LIST** (single, multi and option)
- **SELECTOR** (single, multi)

  Alpha and Graphical view gadgets are not candidates for this mechanism.

  These gadgets' definition graphs now support the 'CORE' qualifier:

```
            .-------<---.
           /            |
<gadget>--+-- tagtext --|  gadget tag
          |             |
          +-- <fgpos> --|  gadget position
          |             |
          +-- CORE -----|  Core managed gadget
          |             |
          .
          .
          .
```

When a form is first displayed its gadgets are created. If a gadget is declared as core managed then F&M, builds the DRUID widget, sets its active, visible and selected states, and then calls core code to allow it to take over the gadget callback.

Core code may plug in its own callback function to DRUID so that events on that gadget will go directly to core code and will no longer go to F&M. Core code will then be responsible for managing the gadgets state in conjunction with PML AppWare. For gadgets with no callback, the core code may merely wish to read values from or write to the gadget. You must exercise great care to avoid clashes between AppWare and software management.

TEXTPANE and **PARAGRAPH** gadgets do not have a callback.

A.2.1 GADGET Core Code Interface

F&M will call the following routine, which must be provided by the core code, to notify core-code of each gadget with CORE control type:

```
CORGDT ( int GADGET, int WMID, int GTYPE, int LSTR, string
GDTNAME )
```

where:

- GADGET is the F&M handle for the gadget element
- WMID the DRUID driver handle for the gadget. This will allow the core-code to provide its own callback for the gadget.
- GTYPE is the gadget type as defined above.
- GDTNAME an application module unique gadget name string of length LSTR

The element GADGET is owned by an F&M FORM element, so its owner (in this case the form) element can be obtained from:

```
FQUOWN( int GDTELT, int FORM, int OWNER ).
```

The following F&M routines may be useful for core code management gadgets:

Query immediate owner and owning form of given element ELT (gadget, menu, menufield etc.):

```
FQUOWN ( int ELT, int FORM, int OWNER )
```

where:

- **FORM** and **OWNER** are returned arguments.
- FORM/GADGET/MENU/MENUFIELD/ELT/OWNER is the F&M handle for an element.
- **IERR** is a PML error token, where 0 means no error.

A.3 Menufield Core Support

A menufield is PML managed by default, but can be declared to be CORE controlled during its definition.

Menufield creation syntax now has the ‘CORE’ qualifier.

Add/Insert methods now allow any menufield type to be pre-pended with ‘CORE’ to indicate a core managed field.

```
!menu.Add( ‘CORECALLBACK’, ‘<Dtext>’, ‘ ’, ‘<FieldName>’ )
!menu.Add( ‘CORESEPARATOR’, ‘<Dtext>’, ‘ ’ , ‘<FieldName>’  )
!menu.Add( ‘COREMENU’, ‘<Dtext>’, ‘ ’ , ‘<FieldName>’  )
```

The F&M menu fields are mapped to DRUID MenuItem widgets at the first `UI_FILL_MENU` or `UI_UPDATE_MENU` event. At this point F&M, builds the DRUID menufield widget, sets its active, visible and selected states, and then calls core code to allow it to take over the field.

Core code will plug in its own callback function into DRUID so that events on that field will go directly to core code and will no longer go to F&M. Core code will now be responsible for managing the menu field’s state in conjunction with PML AppWare.

A.3.1 MENUFIELD core code interface

F&M will call the following routine, which must be provided by the core code, to notify core-code of each field with CORE control type:

```
CORMNU ( int FLDELT, int WMID, int FTYPE, int LSTR, string
FLDNAME )
```

where:

- **FLDELT** is the F&M handle for the field
- **WMID** is the DRUID driver handle for the field. This will allow the core-code to provide its own callback for the menu item.
- **FTYPE** is the field type (0-4). These values correspond to the following types: **CALLBACK**, **TOGGLE**, **SEPARATOR**, **MENU**, **FORM**.
- **FLDNAME** is the core-code/AppWare agreed menu field name string of length **LSTR**

F&M will provide the following interface for core-code to manage the menufield state: Menu field active, visible, checked.

```
FSTMFA( int FLDELT, logical ACTIVE, logical UPDATE=.TRUE. )
FSTMFV( int FLDELT, logical VISIBLE, logical UPDATE=.TRUE. )
FSTMFT( int FLDELT, logical CHECK, logical UPDATE=.TRUE. )
```

The last example will be ignored by non-toggle fields

Note that the **FLDELT** element is owned by an F&M MENU element, so it’s owning menu and form elements can be obtained from:

```
FQUOWN( int FLDELT, int FORM, int OWNER )
```

A.3.2 PML Defined Menufields Managed by Application Framework Addins

It is a requirement that all forms can be shown from the main menu bar of the application framework. For non F&M forms, i.e. the new C# defined forms the following procedure is necessary.

Define PML menu in normal way, but agree menufield names between AppWare and software (Addin). Do not assign a PML callback or declare the field as CORE.

When the menufield is created, DRUID raises a ‘menu field created‘ event which must be subscribed to by the appropriate Addin software. This event will supply the field name. When the Addin receives the event with its agreed field name, it must add its own callback to the menu item. This callback will typically display the appropriate Addin specific form.

B Manipulating VIEWS

This Appendix tells you how to interact with the 2D and 3D View gadgets.

B.1 Manipulating 2D Views

This section describes the standard mechanisms provided by the Forms and Menus software for manipulating a **2D** view using the mouse buttons, number keypad and function keys.

The mouse buttons are referred to as:

- **MB2** (adjust)
- **MB3** (popup)

  If you press **MB3** when the cursor is over the view canvas, the popup view menu is displayed. This will either be the default menu supplied by the Forms and Menus software, or an application-specific version if provided, either by AVEVA or by your own customisation.

  This section describes the default menu.

Zooming is done using:

- The **MB2** (adjust) mouse button.
- The numeric keypad.
- **PageUp**, **PageDown** and **Home** keys

Panning is done using:

- The view scrollbars.
- The numeric keypad.

  Note the **Reset Limits** option on the **View** popup-menu, which will return to the initial view limits.

Zooming Using the Mouse Button MB2

To zoom in and out:

- Move the cursor to a corner of an imaginary box enclosing the area of interest. Hold down **MB2**, move the mouse to the diagonally opposite corner and release the button.
- Click **MB2** to zoom out from the cursor position by factor of 1.5.
- Hold down **Shift** and click **MB2** to zoom in on the cursor position by factor of 1.5.
- Hold down **Ctrl** as well to double the zoom factor.

Zooming Using Page Up, Page Down and Home Keys

These keys will be particularly useful on, for example, laptop keyboards, which frequently do not have a numeric keypad:

Key	Effect
4, 6, 8, 2	Rotate **LEFT**, **RIGHT**, **UP**, **DOWN**, in 10 degree steps.
	Ctrl gives 45 degree steps.
	Shift gives 1 degree steps.
	Ctrl with **Shift** gives 4.5 degree steps.
Page Up	Zoom *in* on view centre (zoom by 1.5).
	Ctrl speeds up to zoom by 2.5.
	Shift slows down to zoom by 1.2.
Page Down	Zoom *out* on view centre (zoom by 1.5).
	Ctrl speeds up to zoom by 2.5.
	Shift slows down to zoom by 1.2
Home	Zoom fully out to view limits.

Panning Using the View Scrollbars

The view scroll bars enable you to pan across the view:

- The slider bubble pans continuously according to speed of movement.
- The arrowheads move the bubble by approximately 1/100 of the bar per click.
- Click in the slider trough to move the bubble by approximately 1/10 of the bar per click.
- Hold down **Ctrl** and click in the slider trough to pan to the opposite side in one step.

Manipulating 2D Views Using the Numeric Keypad

7	**8**	**9**
Zoom in x 1.5 on view centre	Pan down by half view width	Zoom in x 2.5 on view centre
4	**5**	**6**
Pan left by half view width	Zoom out to view limits	Pan right by half view width
1	**2**	**3**
Zoom out x 1.5 on view centre	Pan up by half view width	Zoom out x 2.5 on view centre

Panning Using Alt and the Arrow Keys

Each click pans the view by 1 step distance, where a step is half the size of the current displayed window.

- Holding down **Ctrl** does 10 steps.

- Holding down **Shift** does 0.1 step.

D View, Save and Restore View Methods

2D views now have 4 view stores that allow storage and retrieval of the current view state. They can be accessed by the `saveview()` and `restoreview()` methods:

```
!myview.SaveView(  !storeNumber )
!myview.RestoreView( !storeNumber )
```

where **!StoreNumber** must be in the range 1 to 4.

If the view's picture is deleted or changed then the view-stores are all cleared. The stored view settings should survive saving and restoring to a binary-save file.

B.2 Manipulating 3D Views

This section describes the standard mechanisms provided by the Forms and Menus software for manipulating a **3D** view using the mouse buttons, number keypad and function keys.

The mouse buttons are referred to as:

- **MB2** (adjust)
- **MB3** (popup)

  If you press **MB3** when the cursor is over the view canvas, the **popup view menu** is displayed. This will either be the default menu supplied by the Forms and Menus software, or an application-specific version if provided, either by AVEVA or by your own customisation.

This Section describes the default menu.

View controls using the mouse are analogue. They work best when the redraw time is not too large, so that you get rapid feedback. With long redraw times (large models), you may find that the keyboard digital controls are easier to use.

Status Line

The view parameters are displayed in the **status line**, just below the view. For example:

```
n25w12d Pers<36  Model  Rotate Fast
```

Where the fields have the following meanings:

Field	Meaning
`n25w12d`	The current direction, with angles in degrees
`Pers<36`	The current perspective field of view, in degrees, or else parallel.
`Model`	This is either `Eye` or `Model`. It controls whether rotations keep the from-point fixed (eye point), or the through-point fixed (centre of interest).

Field	Meaning
`Rotate`	This is the current mouse mode for view adjustments (**Adjust** or **MB2**). This mode may be `Zoom`, `Pan`, `Rotate` or `Walk`.
`Fast`	This is the current value of the gearing modifiers for view adjustment. Pressing **Ctrl** displays `Fast` Pressing **Shift** displays `Slow` Pressing **both** displays `Medm`. Otherwise the field is blank. These modifiers apply to both mouse adjustments and keypad actions. In general, `Fast` is about 10 times faster, and `Slow` is about 10 times slower. However, for rotations and zooms different values are used. This field displays `Rest` when the **Alt** key is pressed, and `Save` when **Alt** and **Ctrl** are pressed together. This relates to saving and restoring view settings, described below.

Popup Menu

Pressing MB3 displays the popup view menu when the cursor is over the view canvas.

This will either be the default menu supplied by the *Forms and Menus* software, or an application-specific version if provided, either by AVEVA or by your own customisation.

This Section describes the default menu.

Menu option		Function Key	
Zoom		**F2**	
Pan		**F3**	
Rotate		**F5**	
Walk		**F6**	
Options >	**Eye**	**F7**	
	Shade	**F8**	
	Borders	**F9**	
	Persp	**F4 (F10)**	
Restore >	**R n**	**Alt n**	(**n** is **1, 2, 3** or **4** on keypad)
Save >	**S n**	**Alt Ctrl n**	(**n** is **1, 2, 3** or **4** on keypad)

The **Zoom**, **Pan**, **Rotate** and **Walk** options select a mouse mode.

The **Options** pullright toggles the states of *Eye*/*Model*, *Shaded*/*Wireline*, *Borders On*/*Off*, and *Perspective*/*Parallel*.

These actions may also be achieved by pressing the relevant Function Button while the view canvas has keyboard focus.

The **Restore** and **Save** pullrights allow up to four view settings to be saved and restored. To save the current view settings, choose **S1**, **S2**, **S3** or **S4** from the **Save** menu. To restore a previously saved setting, choose **R1**, **R2**, **R3**, or **R4** from the **Restore** menu. These functions are also available using **Alt**, **Ctrl**, and the keypad numerals **1** to **4** while the canvas has keyboard focus.

Mouse Modes

Mode	Navigation
Zoom (**F2**).	Press **Adjust** down over the view canvas, and then, keeping the button down, move the mouse upwards to zoom in, and downwards to zoom out. This changes the perspective angle, or the view scale in parallel views. It does not change the eye point or the view direction. Use **Ctrl** or **Shift** to speed up or slow down.
Pan (**F3**).	Press **Adjust** over the canvas, and then, keeping the button down, move in any direction to shift the line of sight in that direction. The current view direction is not changed. Note that the picture moves the opposite way to the mouse movement; it is the observer that is being moved. It may be useful to think of moving the mouse towards the part of the picture you wish to see. Use **Ctrl** or **Shift** to speed up or slow down.
Rotate (**F5**).	Press **Adjust** and keep it down, then move left/right to change the bearing of the view direction, or up/down to change the elevation. The initial movement of the mouse determines which is changed, and ***only*** that component of the view direction is modified. To modify the other component, the mouse button must be released and then pressed down again, while moving in the other direction.
Walk (**F6**).	Press **Adjust** button and keep it down, then move up to walk forwards, or down to walk backwards, along the line of sight. Use **Ctrl** or **Shift** to speed up or slow down.
Eye (**F7**).	View rotations keep the from- point fixed, and in Model mode rotations keep the through-point fixed. Use **Ctrl** or **Shift** to speed up or slow down.

In all modes, clicking **MB2** will shift the point under the cursor to the centre of the view. In perspective, the from-point will be maintained, and the view direction and through-point will be modified. In parallel, the view direction will be maintained, and the from- and through-points moved. The only exception to this rule is in *Pan Mode*, when the view direction is never modified.

Manipulating 3D Views Using the Numeric Keypad

Important: Some keyboards have auto-repeat on keypads and function-buttons.

When the keyboard focus is in the view, the following functions are available.

7	**8**	**9**
Zoom in x 1.5 on view centre	Rotate in 10-degree steps	Zoom in x 2.5 on view centre
4	**5**	**6**
Rotate in 10-degree steps	Zoom out to view limits	Rotate in 10-degree steps
1	**2**	**3**
Zoom out x 1.5 on view centre	Rotate in 10-degree steps	Zoom out x 2.5 on view centre

Key	Effect
4, 6, 8, 2	Rotate Left, Right, Up, Down, in 10 degree steps.
	Ctrl gives 45 degree steps,
	Shift gives 1 degree steps
	Ctrl with **Shift** gives 4.5 degree steps.
7, 1	Zoom in/out, changing the view radius by a factor of 1.5. **Ctrl** does two steps at once
	Shift does 1/4 of a step.
	Using them both does 1/2 a step
9, 3	Walk in or out, by 1 step-distance (user selectable).
	Ctrl does 10 steps
	Shift does 1/10 of a step.

Saving views: Use **Alt** and **Ctrl** and keypad numerals **1**, **2**, **3**, or **4.**

Restoring views: Use **Alt** and keypad numerals **1**, **2**, **3**, or **4**.**Function Keys**

Key	Effect
F2	Select *Zoom* mouse mode
F3	Select *Pan* mouse mode

F4	Toggle between *Perspective* and *Parallel*
F5	Select *Rotate* mouse mode
F6	Select *Walk* mouse mode
F7	Toggle rotations mode between *Eye* and *Model*
F8	Toggle between *Shaded* and *Wireline*
F9	Toggle between *Borders Off* and *Borders On* (sliders)
F10 (if available)	Toggle between *ENU & XYZ display modes*

Arrow Keys (Up, Down, Left, Right)

The arrow keys change the line of sight as in Pan mode, by 1 STEP distance.

Ctrl Arrow does 10 STEPs

Shift Arrow does 1/10 of a STEP.

The following changes have been made to allow consistency with ReviewLE and to improve usability on 'lap top' keyboards which frequently do not have the Numeric Keypad.

Key and Mode		**Effect**
PageUp and **PageDown Keys**		*Walk* in or out.
Perspective		*Walk* in or out by 1 step-distance.
	Ctrl	*Walk* in or out by 10 step-distances.
	Shift	*Walk* in or out by 1/10 step-distance.
Parallel		*Zoom* in or out, changing view radius by a factor of 1.5.
	Ctrl	*Zoom* in or out two step-distances at once
	Shift	*Zoom* in or out 1/4 of a step-distance.
	Ctrl + Shift	*Zoom* in or out 1/2 a step-distance.
	Alt	*Zoom* in or out regardless of projection type. For a perspective view this allows the user to choose the perspective view angle for use during the walk through sequence.
Arrow Keys		*Rotate Left*, *Right*, *Up*, and *Down* in 10-degree steps.
	Ctrl	*Rotate* in 45 degree steps.
	Shift	*Rotate* in 1 degree steps.
	Ctrl + Shift	*Rotate* in 4.5 degree steps.
	Alt	*Pan Left*, *Right*, Up, *Down*. Moves line of sight as in *Pan mode*, by 1 step-distance.

	Alt + Ctrl	*Rotate* by 10-step-distances.
	Alt + Shift	*Rotate* by 1/10 of a step-distance.
Numeric Keypad Keys 4, 6, 8, 2		*Rotate Left*, *Right*, *Up*, *Down*, in 10 degree steps.
	Ctrl	*Rotate* in 45 degree steps.
	Shift	*Rotate* in 1 degree steps.
	Ctrl&Shift	*Rotate* in 4.5 degree steps.
	Alt	*Pan Left*, *Right*, *Up*, *Down*. Moves line of sight as in *Pan* mode, by 1 step-distance.
	Alt + Ctrl	*Rotate* by 10-step-distances.
	Alt + Shift	*Rotate* by 1/10 of a step-distance.
Numeric Keypad Keys 7, 1		*Zoom* in or out, changing view radius by a factor of 1.5.
	Ctrl	*Zoom* in or out by two step-distances at a time.
	Shift	*Zoom* in or out by 1/4 step-distance at a time.
	Ctrl + Shift	*Zoom* in or out by 1/2 step-distance at a time.
Numeric Keypad Keys 9, 3		*Walk* in or out.
Perspective		*Walk* in or out, by 1 step-distance.
	Ctrl	*Walk* in or out, by 10 step-distances.
	Shift	*Walk* in or out, by 1/10 step-distance.
Parallel		For *parallel projection* reverts to *Zoom* in or out.

Notes:

- There is no syntax to set the mouse mode, or to enquire it.
- The *eye mode* can be selected by setting *Walkmode On*, and *Model* by setting *Walkmode Off*.
- If your keyboard focus is set by moving the cursor into a window without having to click a mouse button, it is possible to get the state of the **Ctrl** and **Shift** keys confused. If you move the mouse into the view, press **Ctrl**, press **Adjust**, move the mouse out of the view, release **Adjust**, then release **Ctrl**, you will still be in *Fast mode*, as shown in the status line. To remedy this, move the cursor back into the view, and then press and release **Ctrl**.

Index

Symbols

Numerics

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z