ProWesS documentation

PROGS, Professional & Graphical Software
Mechelbaan 344
2580 Putte
BELGIUM
tel : +32 (0)15/ 22 23 26 e-mail : joachim@triathlon98.com
www : http://www.triathlon98.com/PROGS/

---

some articles about ProWesS programming

from QL Today

ProWesS documentation

PROGS, Professional & Graphical Software
Mechelbaan 344
2580 Putte
BELGIUM
tel : +32 (0)15/ 22 23 26 e-mail : joachim@triathlon98.com
www : http://www.triathlon98.com/PROGS/

---

some articles about ProWesS programming

from QL Today

- Global Variables application example

PROGS, Professional & Graphical Software
last edited June 28, 1996

ProWesS Introduction

What is ProWesS ?

ProWesS is short for *PROGS Windowing System*, it is a new window manager, which was designed with a few specific goals in mind :

- Ease of programming. The creation of the user interface of a program is an important task. However, the efforts should go towards making the interface powerful and easy to use, and making the program do its work, not towards creating a window.
- Configurable. It should be easy to change the parameters which determine what a program looks like. If you prefer small text for more information, or large text for better readability, that should be easily configured. In essence, many parameters can be determined about how things are displayed. It should be possible to change these.
- Consistency. A window manager can be used by many programs. It is preferable if all these programs are somewhat consistent. So instead of configuring each program individualy, it would be better to have the general parameters globally configurable. This way each application automatically fits in with the rest, and the programmer is not burdened with it.
- Fast prototyping. ProWesS is designed to allow windows to be created with a limited amount of work. Some extra effort may be required to make it look properly, and definitely to make the scaling work as intended. So ProWesS allows you to concentrate on making a GUI application that works first, and worry about the details later.
- Screen independance and PROforma support. ProWesS is a general

framework which has been designed specifically to allow the use of PROforma for all drawing. This way all the text on your screen can be drawn with the font and size of your choice. Because ProWesS uses PROforma for the drawing, it is possible to have windows which are larger than the screen, and also screen independance. When using a high resolution monitor, the fonts will still be as big as on a screen with less resolution.

- Reentrant code. It should be possible to write reentrant programs using ProWesS. This means that a mechanism for accessing global data has to be provided.

This manual

This manual is intended to explain in detail what ProWesS is about, how it operates, how it should be used and how it can be extended. For some specific details like possible errors of the access routines, we would like to refer to the *ProWesS_ddf* DATAdesign file.

While writing this manual, we often use terms which are not explained until later. It is always a problem to explain something with no or little background knowledge. So if you come accross an unknown term, just read on and everything should become clear later.

There is a separate chapter which contains some example programs. That chapter is written to be read at any time during the exploration of ProWesS. The impatient reader could immediately start there.

The chapter about adding your own types to ProWesS is not for the faint-hearted. That part needs more self-study, it doesn't contain any examples, just the theory. A lot of guidance should however be found in the sources of the standard types. When starting with ProWesS, you should skip that part.

All things considered, ProWesS should not be too difficult to master. We expect that writing programs which use ProWesS should be possible after reading the manual twice. In any case, if you have any problems with all this, do not hesitate to contact us.

We (and everybody who uses these manuals) would like it very much if you could send us any comments about this manual, such as

- omissions
- inaccuracies or mistakes
- typing and/or spelling mistakes
- making this manual into better English
- anything else (positive comments are also always appreciated)

The bottom of each page mentions when the HTML document was last modified. I will try to keep this date correct, however it is only meant to indicate changes in the information provided, I will not change that date when correcting spelling mistakes or HTML errors.

Installation

ProWesS is a job which makes itself available to clients in the form of a Dynamic Link Library (a thing with an efficient access method).

Some extensions have to be loaded for ProWesS to run : the dynamic link library manager, syslib, PROforma, the pointer interface and the thing system (usually part of Hotkey System II). It is also best to have the "Button Frame" loaded for *sleep* to work as expected.

ProWesS has the shape of a job, and loads its configuration file ("ProWesS_cfg") when it starts. A parameter can be given to ProWesS to specify the path where the configuration file can be found (e.g. "win1_pw;flp1" to search on "win1_pw_" and "flp1_" in that order). If no parameter is given or the configuration file is still not found, then first the program default and then the data default devices will be searched.

The fact that ProWesS is a job (and not a resident extension like most libraries such as the Menu Extensions), has certain advantages. Jobs can always be loaded (if you have enough memory), and jobs can always be removed. When loading a job it is possible to pass a parameter (like where to find the configuration file), which is particularly useful. Also, no memory is wasted if ProWesS is loaded while a copy was already running. So if you

want to release the memory which is used by ProWesS, you can just remove the job. Of course the disadvantage of this scheme is that you can accidentally remove the ProWesS job, which is dangerous as all programs which use ProWesS will also be removed, so you could loose data that way (in fact, if you were to remove PROforma, then ProWesS and all jobs using either will also be removed).

The downside

ProWesS is a complex system. It is designed to be easy and fast. However things can go wrong in some situations, so you should watch out for the following points. These things could make ProWesS go wrong completely.

An example is creating an object in a system, which is positioned in another system. This could cause major problems. You should also never use objects after they have been removed, and you should be careful when removing objects that all links to that object are also removed (e.g. removing the owner of a keypress without action, removing an object which can be navigated to in an edline).

Another typical thing which can go wrong are taglists. They are not checked by the c compiler and therefore wou are not warned when things go wrong. When passing invalid tags, just an error will be produced, but you have to be careful about using the correct number of parameters for each tag, and always ending taglists with a NULL.

Fortunately, these problems can be avoided, and a program which is fully tested should always run safely. The main possible problems for properly debugged programs are types which are not available and running out of memory.

ProWesS Configuration

Configuration file

ProWesS is a highly configurable system, many parameters in the system can be changed. However, these can be situated either in the ProWesS program, or in the external types. Therefore, all configurable items have been grouped in a configuration file, called *ProWesS_cfg*, which is loaded when ProWesS is started.

Each line in the configuration file is interpreted as a configuration command. Empty lines are discarded as comments. All the other lines are divided in two types, commands and definitions of configuration constants. The lines with a command have a fixed format : the first character is the actual command, the second character should be a space, and the rest of the line is the parameter. All lines which don't have a space as second character are considered as configuration constants.

The configuration commands currently supported by ProWesS are :

'%' and ';'
: the line is considered as comment and is discarded.

'S'
: set the searchpath for the following commands. The searchpath contains the directories which should be searched to open a file. The directories should be separated by a semicolon (';'). The directories are scanned from left to right. For each directory, the trailing underscore ('_') may be discarded.

'T'

specify a type definition file which should be loaded. The file is searched on the current searchpath. A type is an external module (as supported by syslib), which contains the behaviours of the objects of that type. All types in ProWesS are external, ProWesS has no built-in types. However, the application programmer can safely assume that all types are loaded when this is checked and ensured in the application loader file. It is the responsibility of the person who makes the distribution that this is actually true. The advantage of having no built-in types is that they can all be replaced by better or differently behaving versions. When necessary, the application programmer can also define some application specific types, possibly replacing some that are already loaded.

'?'

this command should not be used in the ProWesS configuration file. The parameter is the name of a ProWesS type (up to four characters, giving the type identifier in binary - usually readable though). It is part of the dynamic configuration system, which allows that an application loader makes sure that ProWesS has a certain type loaded. An error is raised when the type does not exist.

'v'

this command should not be used in the ProWesS configuration file. The parameter is the minimum version number you want to use. An error (ERR_ISYN) will be returned when the version of ProWesS is older than the version requested. An example of the use of this is `v 1.03`.

Lines which contain the definition of a configuration constant contain the name of the configuration constant, followed by the parameter, separated by one or more spaces or tabs. Each definition is passed to the system and each of the types for processing, which can thus modify their behaviour.

By convention, the name of the configuration constant starts with the type which is intended to process the result. However, all other types also see the definition and get a chance to modify their behaviour according. As the name of the configuration constant may not contain spaces, a dashe ('-') is used to separate the words. The names are case dependant.

Dynamic configuration

ProWesS can also be configured further while it is already active. ProWesS contains a special entry point which allows you to pass configuration lines which are then processed.

The ProWesS thing has a `CNFG` extension which is used for this purpose. This extension is a function which accepts a character array as parameter ("char *"). This string is handled as if it were a line in the configuration file. It is thus possible (as mentioned in the previous section) to set the searchpath, load external types, pass a configuration constant or check whether a certain type is already available.

The call to this routine can be done as follows :

```
#include
#include

...
    Error err;    /* the error returned by the config routine */
    char *str;    /* the config line which is passed */
    ...
    err=THINGCall(PW_THING_NAME,PW_THING_CNFG,1,str);
    if (err) ...  /* error handling */
...
```

Definition constants

The constants are type specific. Some are globally defined and interpreted by the ProWesS system, but most are specific to the types. However, all constants are passed to all types which are loaded at that moment. It is allowed to interpret definitions which are originally intended for another type.

However, to make sure that this is not necessary, and to make configuring ProWesS applications easier, ProWesS also stores some default values which can be changed by definition constants, but which can also be queried. This allows types and applications to use this information as default values. This

way, the default font and fontsize can be defined, and the default colours for the background, foreground and middleground can be set.

- ProWesS system
- applic type
- edline/dedline types
- dirselect type
- separator type
- infotext type
- infostring type
- menu type
- title item type
- loose item
- scroll type
- item/itemp types
- label type

Execution model

General

ProWesS is a truly event driven window manager. The programmer is responsible for setting up the window (specifying what has to be in it), and then activating that window (passing control to ProWesS). ProWesS will then draw the window and wait for events to happen. All events are processed by objects which have been put in the window. The objects either have a fixed behaviour for a particular event, or accept functions which can process the event. The exact behaviour of the object depends on its type.

The event handler can then influence the system, for example by telling it that the window should be put asleep, or that control has to be passed back to the application (in which case the window is removed from the screen).

Event handlers can also change something inside the window, like change the appearance of some objects, or change the size and/or existence of some parts of the window. ProWesS will automatically redetermine the sizes of the objects when this is necessary, and the relevant parts of the window are also automatically redrawn when needed. This is done when control is passed back from the event handler to ProWesS.

In fact a ProWesS system is in one of three possible states :

- inactive : the window is invisible. Usually the window is being built or waiting to become active.
- waiting : the window is now visible, and ProWesS is in control. The system is waiting for events to happen. The user can generate events by pressing a key or moving the mouse.

- handling an event : the window is also visible, but an event handler is in control. Any effects of the event handler's actions will only become visible when control is passed back to ProWesS (when the event handler terminates). ProWesS will then figure out whether part of the window has to be redrawn, or whether the window has to be rebuilt completely. It is possible that the window is removed from the screen and control is passed back to the application (when a PW_SYSTEM_BREAKDOWN tag was passed), or that the system is put to sleep (when a PW_SYSTEM_SLEEP tag was passed).

While a system is activated, it is either waiting or handling an event.

Of course an application program can own several systems simultaneously, and they can even be activated, but only one of the systems will be waiting for or handling events (the one that was activated last).

The system is an underlying structure which combines all the objects in it. ProWesS therefore always needs an object to know which system is referred to.

Window Structure

In ProWesS a system (which is displayed as a window), consists of objects. Each object has a type, and types are divided in two groups. The groups are :

- keypresses : the objects of the type of this group are not visible in the window. However, a keypress can be associated with these objects, so that an event is generated when that key is pressed. Keypresses are case dependant, so pressing <a> or <shift a> is not the same. However, if a key is pressed inside a window, and there is no object which reacts to that keypress, then the case is changed and another attempt is made to match that keypress. If there is still no match, than a PW_EVENT_CATCH can be generated to an object which is designated to catch all unmatched keypresses (this object is selected using PW_CATCH_OBJECT).
- regions : the objects of this type of group are associated with a part of

the window. Regions can be nested hierarchically in a window. The regions in a window are alternatedly positioned in rows and columns.  This picture shows the nesting of regions in the window. The nesting level of the regions is displayed in the top left corner. Normally speaking however, the programmer need not bother with this structure, as some hints can be given when creating an object about how to position that object in the window.

ProWesS automatically tries to make a window look good by fitting the regions in it, and if the window should be larger than the minimum size to fit everyhing inside, scaling is automatic. These two operations are related, and hints can be given to make it works as you want. The regions in the window are organised in a hierarchy. At each level regions are either positioned next to each other, or below each other. This is called the primary direction. The primary direction changes with each level of nesting, starting with horizontal (row). Each region has two variables, a scale factor and a windowfit status (they can be changed using the PW_WINDOWFIT and PW_SCALE_FACTOR tags).

The windowfit status determines whether the size of a region has to be adjusted in the secondary direction. The scale factor determines how the size of a region is adjusted in the primary direction. If there is some extra space here, then this extra space is divided among the regions according to the scale factors. If the scale factor for one of the regions is one, and the scale factor for another region is three, then that last region will be increased in size by three times as much as the first region (in absolute figures). This is also demonstrated in the next picture. 

When ProWesS displays a window, it will be drawn with a border. When the pointer moves over this border, the pointer will change into a diagonal arrow which points both ways. This is the scaleborder. If you generate a HIT on this border (by pressing the left mouse button), you can move the window. If you press DO (right mouse button), then you have the opportunity to change the size of the window (unless the window is not scalable, in which case you can just move the window again).

It may happen that the window does not fit on screen. ProWesS will just

display as much as is possible, and you can scroll the invisible part into view by pressing <control shift alt cursor key>.

Access routines

The ProWesS access routines have the following prototypes :

```
typedef struct _PWDummyObject *PWObject;
typedef unsigned long PWType;
typedef struct _PWTypeDef PWTypeDef;

Error PWCreate(PWObject owner, PWObject *ret, PWType type, ...);
Error PWDoCreate(PWObject owner, PWObject *ret, PWType type, int
Error PWChange(PWObject object, ...);
Error PWDoChange(PWObject object, int *taglist);
Error PWQuery(PWObject object, int what, void *ret);
Error PWActivate(PWObject object);
Error PWAddType(PWTypeDef *newtype);
Error PWRemove(PWObject object);
```

The variable arguments list is used to pass a taglist to the function. A tag is a number which indicates a function and which can have its own parameters. Such parameters can have any type, and there can be between zero and 15 of them. A tag can also have a list of parameters (the list is ended by the parameter NULL), or the parameter can be another taglist. A taglist itself is also terminated with a NULL tag. The use of a taglist will become clear in the examples.

In ProWesS there are three kinds of tags : creation tags, change tags and query tags. A particular tag can belong to more than one kind at the same time.

A *query tag* can only be used as the *what* part of the `PWQuery` function. A *creation tag* has a special meaning when passed to `PWCreate`, and is normally not valid when passed to `PWChange`. A *change tag* can be used to change some characterictic of an object. It can be applied at any time, and is part either of a `PWCreate` or a `PWChange` taglist.

The definitions of the prototypes and of the tags supported by ProWesS and by the standard types are defined in "ProWesS.h".

PWCreate

```
Error PWCreate(PWObject owner, PWObject *ret, PWType type, ...);
```

This command is used to create an object. An owner has to be passed to indicate the system to which the newly created object has to belong. If the owner is NULL then a new system will be created for the object.

On return, *ret* will be filled in with the object identifier of the newly created object. It will be `NULL` if an unrecoverable error occured. If the object identifier is not used in the program, `NULL` can be passed instead.

The *type* parameter indicates which type of object has to be created. Then follows a taglist with some parameters for the newly created object.

When a new region is created, it has to be positioned in the window. This can either be done by specifying one of the positioning tags explained below, or by means of a default rule (see also window structure).

The default rule positions the region as last object inside the owner. Thus it will be displayed at the bottom right in the owner. To achieve this, the children of (that is, the objects inside) the owner are scanned. While the last (rightmost or bottommost) child is not composite and has children, that region is used as owner. If the last region has no children, then the object is linked after that last region. If the owner did not have any children, the newly created region will just be linked as child.

The `PW_POSITION_RIGHT_OF`, `PW_POSITION_LEFT_OF`, `PW_POSITION_BELOW` and `PW_POSITION_ABOVE` tags (which require a "`PWObject`" parameter) specify a position relative to the object given as parameter. If the parameter is inside a region in the correct primary direction, then the new region is just linked in before or after the parameter object. If not, then a new container region (a row or column) is created which will contain both the parameter object and the new object. This container region can also be created explicitly (`PW_TYPE_DIRECTION`). The container automatically windowfits and has a scale factor of 1. If you want something else, you have to do it explicitly. Please note that the introduction of an extra level of nesting of regions can, if

the parameter object has children, have the effect of changing the positioning of these children. Objects which where next to each other will suddenly be positioned below each other and vice versa.

The PW_POSITION_NEXT_ROW and PW_POSITION_NEXT_COLUMN tags will create a new container region (type PW_TYPE_DIRECTION) at the bottom/right inside the owner. The newly created object will be the first object in the newly created row/column. ProWesS will make sure that the primary direction of the container region is as requested.

PWDoCreate

```
Error PWCreate(PWObject owner, PWObject *ret, PWType type, ...);
```

This is a variant of the PWChange command. It is provided because it allows the programmer to pass the parameter list in an array instead of on the stack. This can sometimes prove useful (e.g. for storing window descriptions in a data structure instead of code).

PWRemove

```
Error PWRemove(PWObject object);
```

This command removes an object from the system. If the object which has to be removed is a region object, then all the children will also be removed. When all the regions in the system have been removed (with the exception of the PW_SLEEP_OBJECT and its children), then the system itself will be removed.

PWChange

```
Error PWChange(PWObject object, ...);
```

Some of the parameters of an object or the system can be changed with this command. It requires the object for which the changes should be valid, and a

taglist which specifies what has to be changed.

PWDoChange

```
Error PWDoChange(PWObject object, int *taglist);
```

This is a variant of the `PWChange` command. It is provided because it allows the programmer to pass the parameter list in an array instead of on the stack. This can sometimes prove useful (e.g. for storing window descriptions in a data structure instead of code).

PWQuery

```
Error PWQuery(PWObject object, int what, void *ret);
```

ProWesS can also be queried about the current status of some of its internal settings, or about some of the objects. The what parameter has to be a query tag, and the result is returned in *ret*.

Although this is a very straightforward way to query ProWesS or an object, it may sometimes be too limited (e.g. for querying about a string which should be returned in a buffer of limited length). In such cases, the `PWChange` mechanism can still be used to formulate a query.

PWActivate

```
Error PWActivate(PWObject object);
```

This is the most important command of them all. After you have set up all the objects which make up your window, you can now get it up and running. This function will display your system and wait for events to occur. When they happen, they will be processed by the appropriate event handlers (as you have installed them). This function will only terminate after the system has been passed a `PW_SYSTEM_BREAKDOWN` tag. The window will then be removed.

PWAddType

```
Error PWAddType(PWTypeDef *newtype);
```

In some cases you may need a special type which is application specific. Such a type can be added using this function. When a new object is created, first a list of application specific types is searched for a type with the given identifier. If that fails, the normal list of types is searched for a match. It is thus possible to replace one of the loaded types with something else for this particular application.

Standard types

This document gives a general explanation about all the types which are always available in each ProWesS distribution. It only gives information about the general purpose and use of each type. Links are available to the definitions and tags for the objects of the type.

---

system

The system is not a type, but is can be queried or changed through any object which is part of that system.

Outline

An outline is intended to be the first object when a window is constructed, with all other objects as children. An outline consists of a title item (which by default displays the program name), with a separator line below that.

Optionally there can be some items at the sides, like a *Quit*, *Sleep*, *Help*, *Do*, *Wake*, *Info* item. The info item can be defined by the programmer (both text and action). The other items have their standard use.

The *sleep* item should only be used for primary windows (the first one to be activated). All items have their standard keypresses by which they can be activated.

The *quit* item can automatically ask for a confirmation request before exiting the program. If you want, you can also attach a keypress to the quit item.

The *outline* also contains two empty boxes at the left and right. The `PWObject` of these objects can be queried for, and some objects can be created inside these objects. Thus the functionality of the outline can also be extended. These boxes are created with a zero scale factor.

- tags for outline

Loose Item

A loose item is an object which can be "indicated". When the pointer appears inside a loose item, a border is drawn around the item (if the *item* type is used, the behaviour is different when using *itemp*). The item normally contains some text, although a routine can be provided to draw an icon in it.

A loose item reacts to both *HIT* and *DO* events. If you want you can also attach a keypress to it, which will be the same as a *HIT* event. A loose item can have one of three statusses : *available*, *selected* or *unavailable*. When an item is unavailable, it can't be indicated (and thus will not get a border). A *HIT* on the item will toggle between available and selected, and a *DO* will

select the item. Optionally, a user defined function can also be called on a *HIT* or *DO* event (not when the item is unavailable). The designer of the window can also choose that the status of the item can never change, and that a *DO* event can be propagated to the window.

- tags for loose item
- definitions for loose item

Title Item

This is a type which should be used to display a title. It displays the title centred in a bar (which is at least slightly wider than the title). The bar is usually displayed in a different colour to make the title stand out a little. The title is there for information only, it does not react to any events.

- tags for title item type
- definitions for title item type

Separator & Container

To seperate two (or more) objects, you often want to put a line between them. A separator does exactly that.

In some cases you don't just want a line somewhere, but you want a box around an item. In that case you can use a container object and put the items which should be in the box inside.

- definitions for separator type

Direction & Glue

A direction is a region object which can be used as a container to put other regions inside. The use of this type can be necessary if you want to control the positioning of the objects in the window exactly, and the default scale

factor or windowfitting do not suffice for your purposes.

Glue (which is in fact the same as a direction, but the name better fits the purpose) can be used to create some "special" effects for positioning objects. Glue is an invisible region which can be used for spacing out other objects. For example, if you have two objects, one of which should appear at the left of the window and the other one at the right, then you can put some glue in between, and that glue will stretch to eat unused space (according to the scale factor).

Keypress

A keypress object is not visible in the window. It will react to a keypress in the window. The keypress which it should react to can be set with `PW_KEYPRESS` (like any object which belongs to the keypress group). A function can be set which should be called when the given key is pressed. If a keypress object does not have an action assigned to it, it will be handled as a *HIT* event in the owner (as set when creating the keypress object).

In ProWesS, keypresses are case dependant. However, if there is no object which reacts to a particular keypress, then the case is changed and another attempt is made to match that keypress. If there is still no match, than a `PW_EVENT_CATCH` can be generated to an object which is designated to catch all unmatched keypresses (this object is selected using `PW_CATCH_OBJECT`).

- tags for keypress type

Label

A label object is normally used to indicate the purpose of another object (e.g. to tell what is being edited in an edline). A label assumes that it is positioned either left or above the object to which it refers.

When the label is displayed to the left of another object, the name in the label will be displayed at the top right. If the label is positioned above another

object, then the text will be displayed at the bottom left.

- tags for label type
- definitions for label type

Edline/dedline

The edline object is designed to allow the user to edit one or more lines of text (strings). The maximum length of each line can be set when you create the edline (or a default maximum length will be used). As an edline can not reasonably determine its width, that also can be given (or a default will be used).

An edline is available in two variants. The normal edline has to be indicated first to start editing the string. While the string is being edited, the pointer cannot be used in the window.

There is also a direct edline. This variant uses the keypress catching mechanism. To make sure that you can move the cursor, a direct edline will make sure that `PW_CURSOR_SEPARATE` is `TRUE`. A direct edline doesn't have to be indicated to start typing, but it has to be selected in some way as current catch object. If there is no catch object when a direct edline is created, then that direct edline will designate itself as such.

An edline can have an action which has to be executed when the user stops editing (presses <enter>). Also the programmer can set some objects which can be navigated to (using <tab>, <shift tab>, <up>, <down> or <enter>). The objects which can be moved to have to be able to work as a catch object.

Edline objects can handle the edited text being larger than the object can display (thanks to PROforma). The edline will also always try to assure that the cursor is visible with its surrounding characters. The cursor will therefore never hit the right end of the object, and the left end will only be touched when the cursor is at the start of the string.

Although you can choose between normal and direct edline, it is possible for the users to configure their setup to always use either. To allow the users to

choose where they want to input their text, it should always be possible to navigate to all the other edline objects in the window without using the mouse.

- tags for edline/dedline types
- definitions for edline/dedline types

Infostring & Infotext

These two types are quite similar to each other. They both display a text (lines separated by '\n', '\0' terminated) for information only. These objects don't react to any events.

An infostring object is always completely visible. When the text is changed, the size of the object will change (as with many other types, but this can be switched off).

Infotext objects have a size which is determined by the designer of the window. Scroll bars are displayed to allow the user to scroll invisible parts into view.

- tags for infotext type
- definitions for infotext type
- tags for infostring type
- definitions for infostring type

Menu

A Menu is a special purpose application window (explained later). It provides a scrollable list of loose items. The contents of each loose item is a string.

A menu is always vertically scrollable. The number of visible lines and the minimum width can be chosen by the designer of the window (the minimum width can be the maximum width of an item in the menu).

The items in the menu can be sorted using a compare routine as provided.

When a menu is displayed, the items are all displayed in rows. Each row contains as many items as can be completely visualised (i.e. the width of the menu is divided by the itemwidth to get the number of items on each row). The menu is only scrollable vertically.

A menu can have no selectable items, at most one selected item, or all items can be selected and deselected at will. The programmer can also query all selected, all available or just all items (returning a pointer to the string). Also the status of each item can be changed at will (normally an item is just available when it is selected).

- tags for menu type
- definitions for menu type

Canvas

A canvas is a general purpose region object. It is a gateway for the application programmer to have direct access to all events which occur in it, for example to be used as the drawing area in a graphics program.

The canvas does not know what is in it. The contents has to be drawn using a user supplied routine. Therefore the size of the canvas has to be given explicitly. To allow for scrolling etc. the canvas also has an origin, which is the coordinate at the top left corner of the canvas. The origin is there only for information (to be used by the action routines) and is not used by the canvas type itself.

A canvas can react to the events which can occur inside it. A user supplied routine can be called when the pointer enters or exits the canvas, when a timeout occurs inside the region, or when the pointer is moved. Other events which can be reacted upon are a *HIT*, *DO*, or dragging (see next paragraph). An action routine can also be called when the canvas is resized.

Dragging is a *HIT* or *DO* which lasts longer than is normal, usually while moving the pointer. A difference is made between dragging with a *HIT* and dragging with a *DO*. An event is generated both when the dragging starts and when it ends. During dragging, the pointer is not allowed to move outside the

canvas region. It the user does attempt to move outside, the pointer position will be adjusted to stay inside the canvas. This can also be trapped, for example to automatically scroll the canvas when it occurs.

- tags for canvas type

Scroll

This object is a scrollbar. It is specially designed to interact with a canvas. The scrollbar allows vertical scrolling when it is positioned in a row (usually left or right of the canvas), and it allows horizontal scrolling when it is in a column. The scrollbar should be attached to a canvas. That canvas can then be queried for its size and origin, so that the scrollbar can display that info.

The scrollbar normally contains arrows and a bar to indicate the visible part in the whole. On creation, the designer of the window can choose not to display that bar (then arrows will then be larger). The minimum and maximum origin can be set. This will allow the bar to be displayed accurate and the scrolling can be restricted to stay between these two extremes. Because the origin may be in any user defined unit, it may be necessary to provide a routine to convert a size (of the canvas) into the unit used for the origin of the canvas.

The user can scroll the canvas by indicating the scroll arrows. The designer of the window can set the distances which should be scrolled in the case of a *HIT* (minimum scroll distance), or a *DO* (maximum scroll distance) on the scroll arrows.

- tags for scroll type
- definitions for scroll type

Application Window

An application window is a composite object. It provides a canvas together with some scrollbars if you want them. It is very basic, all you can do is tell

the application window to allow horizontal or vertical scrolling (with or without bar).

It is intended that you never directly use the scroll type, but if you want a canvas with a scrollbar, that you always use an application window.

- tags for applic type
- definitions for applic type

Directory Select

This is an object which allows you to select a directory in a directory select window. However, as there is no special method to have a window type of object, it is implemented as a keypress object. If the associated key is pressed, the window is activated, but you can also explicitly active the window. (Note that if you don't attach a keypress to the dirselect object (which is done with the `PW_KEYPRESS` tag), the window can't be activated with a keypress).

A directory select window consists of an edline where the directory can be edited directly, a list of devices and a menu which contains both the subdirectories to the current directory, and possibly some directories which have been configured because they are often used.

The devices which are displayed are the devices which have been configured as being the accessible (or most used) devices on your system (using `DIRSELECT-DEVICE`). The often-used directories can be configured using `DIRSELECT-DIRECTORY`).

The directory which should be used as default in the dirselect window can be set explicitly (it defaults to "", which is converted to the data default (as in `DEVDataGet`) when the window is activated). The title of the window can also be set. When the directory has been selected (the window is exited), a user-defined action is called, which can use the selected directory to poor it into a parent window or do some other operation.

- tags for dirselect type

- definitions for dirselect type

File Select

This is an object which allows you to select one or more files in a file select window. However, as there is no special method to have a window type of object, it is implemented as a keypress object. If the associated key is pressed, the window is activated, but you can also explicitly activate the window. (Note that if you don't attach a keypress to the fileselect object (which is done with the `PW_KEYPRESS` tag), the window can't be activated with a keypress).

A fileselect window normally allows the user to select just one file, but it can also be created to allow the user to select multiple files (in that case there is no edline to enter the filename, and the *All* item appears).

A fileselect window always displays a directory (of which the subdirectories and the files in it are also visible). The directory can be edited (with a *HIT* on the edline), or changed via the directory select window (a *DO* on the edline). The fileselect window also contains an edline in which you can set some file extensions (separated by a semicolon (;)). Only files which end in one of the extension will be displayed in the menu (or if *Not* is indicated, only files which do not end in any of these extensions will be displayed).

Of course the directory which has to be displayed, and the default extensions, the status of the *Not* item and the window name can be set explicitly. If only one file can be indicated, the default filename can also be set (note that all these values are preserved across activations of the window).

The designer of the window should pass a routine which has to perform some action with the selected file(s), otherwise the fileselect window serves no purpose to the user (this routine can for example load that file).

The fileselect window will by default use up the entire possible height. It contains a wake and quit item, but will also quit in case of a *DO* inside the window.

The fileselect can be customized to a large extent. You can insert some extra objects in the window in a box between the title bar (the outline) and the rest of the window. You can also change the title bar itself. Thus you could add a sleep item, redefine *Do* etc. This way you don't have to worry about the file selection etc. features in some applications.

- tags for fileselect type

item/itemp

This is not a general purpose type. It is a support type which is used for example by the loose item and the edline types. It is used to draw the border around an item, normally when the pointer is indicating that item. It is a separate type because it is common to more than one type, which prevents both duplication of code and strange configurations (the borders should always be equally thick, no mather what kind of object it is around).

This type should normally only be accessed from within other types. It therefore only contains some of the handlers, only the `Draw`, `Entry`, `Exit` and `DetermineSize` routines are provided. The `Draw` routine may draw a permanent border (as is the case for the *itemp* variant). The `Entry` routine will draw the border, and the `Exit` routine will remove it. The `DetermineSize` should be used to increase the total size of the region to include the border and to set the widths of the margins around the hit area.

- definitions for item/itemp types

List Select

A listselect object is a very compact way to allow the user to make a choice from a selection of things. It looks like a loose item which displays its current value and starts with a pointing down arrow to indicate that there are more choices. When the item is indicated, you get a submenu from which you can choose a new value.

- tags for listselect type
- definitions for listselect type

PROGS, Professional & Graphical Software
last edited May 8, 1997

ProWesS examples

Simple "Hello World"

Let us start with the easiest ProWesS program you can imagine. All this does is create a window with one item in it, which says "Hello World". The item can be indicated (the status will change), but this will do nothing.

```
#include <ProWesS.h>

void init()
{
    PWObject first;

    PWCreate(NULL,&first,PW_TYPE_LOOSE_ITEM,
                PW_LOOSE_TEXT, "Hello World",
                NULL);
    PWActivate(first);
}
```

Obviously, the first line includes the definition of all the prototypes and the tags of the standard types. The program just creates a loose item, with "Hello World" as text. The system thus created is then activated.

Proper "Hello World"

The program given above is as simple as you can get. Unfortunately it is a bit too limited. It would be useful to be able to terminate the program gracefully.

Also it would be nice to be able to put the program to sleep, and a title bar wouldn't go astray. Therefore, some little changes.

```
#include <ProWesS.h>

void init()
{
    PWObject outl;

    PWCreate(NULL,&outl,PW_TYPE_OUTLINE,
                PW_OUTLINE_QUIT,
                PW_OUTLINE_SLEEP,
                NULL);
    PWCreate(outl,NULL,PW_TYPE_LOOSE_ITEM,
                PW_LOOSE_TEXT, "Hello World",
                NULL);
    PWActivate(outl);
}
```

All the things we want to add (and more) are provided by the outline type of object. Therefore, it is also sufficient to create an outline object in the window. The outline always contains a title bar with the program name (by default). We have specified that it should also contain a quit and a sleep item.

The outline is designed such that all other objects in the window should preferably be children of the outline. Therefore the outline has to be created first, and the contents are created later.

A scrollable canvas

This example program contains a scrollable application window which can display something (a piece of big text in this case).

```
#include <sms.h>
#include <win.h>
#include <ProWesS.h>

void init()
{
    PWObject outl;

    LinkPROforma();         /* get a DLL link to PROforma */
```

```
    PWCreate(NULL, &outl, PW_TYPE_OUTLINE,
                PW_OUTLINE_SLEEP_TEXT, "PROforma applic test",
                PW_OUTLINE_QUIT,
                NULL);

    PWCreate(outl, NULL, PW_TYPE_APPLIC,
                PW_APPLIC_CANVAS_LIST,
                    PW_CANVAS_SIZE_PIX, 100,40,
                    PW_CANVAS_ACTION_REDRAW, &redraw,
                    PW_CANVAS_POINTER, SPRITE_HAND,
                    NULL,
                PW_APPLIC_SCROLLBAR_Y,
                PW_APPLIC_YSCROLL_LIST,
                    PW_SCROLL_MINDIST, short2pt(20),
                    PW_SCROLL_MAXDIST, -short2pt(20),
                    PW_SCROLL_MAXIMUM, YMAX,
                    NULL,
                PW_APPLIC_SCROLLBAR_X,
                PW_APPLIC_XSCROLL_LIST,
                    PW_SCROLL_MINDIST, short2pt(20),
                    PW_SCROLL_MAXDIST, -short2pt(20),
                    PW_SCROLL_MAXIMUM, XMAX,
                    NULL,
                NULL);

    PWActivate(outl);
}
```

The first call to PWCreate will create and "outline" object, which is the title bar for the application. The title defaults to the job name, so we don't have to worry about that. We then make sure the application can be put to sleep, and give the text which should be displayed when the program is sleeping (the `PW_OUTLINE_SLEEP_TEXT` tag with one parameter, the text). The `PW_OUTLINE_QUIT` tag makes sure that there is also a quit item. The `NULL` terminates the list of tags. If we wanted a confirmation request when quitting the program, we could have included the `PW_OUTLINE_QUIT_CONFIRM` tag which needs a boolean parameter.

Then the application window is created. An application window is in fact a composite object, built from a canvas (which displays the info) and scrollbars. Therefore the window is instructed that is should contain a scrollbar in each direction, and lists of tags are passed on to the underlying

objects. These lists specify the size of the canvas (in pixels in this case), how to draw the window, and the pointer which should be used inside the window. The scrollbars are passed the distances for scrolling (`MINDIST` for a *HIT*, `MAXDIST` for a *DO* on the scroll arrow, the negative value indicates canvas "size+distance"). Then the maximum value for the scroll is passed (the minimum defaults to zero).

All that remains to be done is write the routine to draw the canvas.

```
#define FONT "English 157"
#define STRING FONT
#define SIZE short2pt(100)
#define XMAX short2pt(500)
#define YMAX SIZE*2

Error redraw(PWObject object, CanvasInfo *rinfo)
{
    int i,j,siz;
    Gstate gid=rinfo->gstate;

    /* set colour for drawing and clear "window" */
    PFPaperColourRGB(gid,pt_hundred,pt_hundred,pt_hundred);
    PFColourRGB(gid,0,0,0);
    PFClearPage(gid);               /* clear background */

    /* adjust page origin to reflect position in applic */
    PFSetPageOrigin(gid,rinfo->xorg,rinfo->yorg);

    /* draw the contents (just some text) */
    PFLoadFont(gid,FONT);
    PFScaleFont(gid,SIZE);
    PFMoveTo(gid,short2pt(10),SIZE);
    PFShowString(gid,STRING);
}
```

This is a relatively straightforward piece of code which just prints the name of a font (using the same font). The page origin is set to reflect that the window may have been scrolled. The gstate is automatically set to the area of the canvas with (0,0) as coordinate at the top left in the canvas.

Add an edited line

This is a program example program which demonstrates the support for dynamic windows and for reentrant code. The user can edit a line, and when he/she confirms it (presses <enter>), an item is added in the window displaying that line of text.

```
#include <mem.h>
#include <ProWesS.h>

#define MAXLEN 50

void init()
{
    PWObject obj,it1;

    PWCreate(NULL,&obj,PW_TYPE_OUTLINE,
                PW_OUTLINE_SLEEP,
                PW_OUTLINE_QUIT,
                PW_OUTLINE_ACTION_DO, &newitem,
                NULL);
    PWCreate(obj,&it1,PW_TYPE_EDLINE,
                PW_EDLINE_SET, "type text here",
                PW_EDLINE_MAXLENGTH, MAXLEN,
                NULL);
    PWChange(obj,PW_GLOBAL_AUXILIARY,it1,NULL);
    PWActivate(obj);
}
```

Again we start by creating an outline which contains a sleep and quit item (the sleep button will now contain the job name as text). It will also contain a "do" item. When "do" is indicated or a *DO* is generated somewhere in the window the "newitem" event handler will be called.

Also the item to edit a line of text is added. The text inside the item is set, and the maximum length of the text is also set (this is an example of a creation tag).

To make sure that no global variables are needed, the data which is needed by the event handler routine is put in a global auxiliary variable. In ProWesS each object and the window (global) have an int/long sized global variable. This can be used to store e.g. pointers to global data structures.

```
Error newitem(PWObject object)
{
```

```
    PWObject edline;
    char str[MAXLEN];
    PWQuery(object,PW_GLOBAL_AUXILIARY,&edline);
    PWChange(edline,PW_EDLINE_GET,MAXLEN,str,NULL);
    return PWCreate(edline,NULL,PW_TYPE_LOOSE_ITEM,
                        PW_LOOSE_TEXT_COPY, str,
                        PW_POSITION_BELOW, edline,
                        NULL);
}
```

To start, we need to get the object id of the edline object, so we can get the string from it. Then the string is copied into a local string. Because we have to pass the buffer size, `PWQuery` can't be used, so it is handled by `PWChange`. The actual work is done in the call to `PWCreate`. A new item with the text from the edline is created. The text is set with `PW_LOOSE_TEXT_COPY`. Normally, just the pointer to the text is used by a loose item, but because the text is now stored in local memory which will be released at the end of the function, the string has to be copied by the loose item. The new item is positioned below the edline object (thus above previously added objects).

This also shows that ProWesS never explicitly works with coordinates. In some cases the size of the object has to be given, but the position is determined either by default rules, or by positioning it above, below, left or right of another object.

File Viewer

Let's make some progress here, and more to a somewhat useful program. This one is just a window which has a title and an area to view the file. The user can indicate "Load file" and a file select window will be displayed. The thus indicated file will then be displayed in the view area.

This is a rather crude file viewer which is really only suited for text files, as the infotext type is used to display the file. This means that if the file contains a `NULL` character, the rest will no longer be displayed.

```
#include <ProWesS.h>

typedef struct {
```

```
    PWObject file,it;
    char *base; int length;
} Global;

Error init()
{
    PWObject obj;
    Global g;
    Error err;

    g.base=NULL; g.length=0;
    err=PWCreate(NULL,&obj,PW_TYPE_OUTLINE,
                PW_OUTLINE_SLEEP,
                PW_OUTLINE_QUIT,
                PW_OUTLINE_QUIT_KEYPRESS, 27,
                PW_OUTLINE_INFO_TEXT, "Load file",
                PW_OUTLINE_ACTION_INFO, loadfile,
                PW_GLOBAL_AUXILIARY, &g,
                NULL);
    if (err) return err;
    err=PWCreate(obj,&g.it,PW_TYPE_INFOTEXT,
                PW_INFOTEXT_WIDTH_FS, short2pt(30),
                PW_INFOTEXT_LINES, 12,
                NULL);
    if (err) return err;
    err=PWCreate(obj,&g.file,PW_TYPE_FILESELECT,
                PW_KEYPRESS, 'l',
                PW_FILESELECT_ACTION, doload,
                NULL);
    if (err) return err;
    err=PWActivate(obj);
    return err;
}
```

As usual this just creates an outline object, and then puts an infotext in that. The infotext will be used to display the file. Some kind a minimum size is given to the infotext, this is rather approximate.

To include the "Load file" item, we used the info item in the outline. This is an auxiliary item which can be used for any purpose, and load file is just one example (it is intended to be used for displaying program info like the author, version, etc). An event handler has to be provided for the "Load file" item to activate the fileselect window.

```
Error loadfile(PWObject object)
```

```
{
    Global *g;
    PWQuery(object,PW_GLOBAL_AUXILIARY,&g);
    return PWChange(g->file,PW_FILESELECT_ACTIVATE,NULL);
}
```

The fileselect window is very straightforward. It can be activated by pressing 'l' or 'L', or indicating the "Load file" item. The fileselect is passed an action routine, which has to be executed when a file selection is confirmed (when the fileselect window is quitted from).

```
Error doload(PWObject object, char *dir, char *name)
{
    Channel ch;
    Global *g;
    Size size;
    Error err;

    PWQuery(object,PW_GLOBAL_AUXILIARY,&g);
    err=IOOpenPath(name,OPEN_READ,dir,&ch);
    if (err) return err;
    IOLength(ch,&size);
    size++;
    if (size>g->length)
    {
        if (g->base)
            MEMRelease(g->base);
        g->base=NULL; g->length=0;
        err=MEMAllocate(size,&g->base);
        if (err) return err;
        g->length=size;
    }
    size--;
    IOLoadFile(ch,size,g->base);
    g->base[size]='\0';
    IOClose(ch);
    return PWChange(g->it,PW_INFOTEXT_TEXT,g->base,NULL);
}
```

This routine doesn't have much to do with the window manager. The global data structure has to be retrieved (with PWQuery), and the the file has to be loaded in memory. A end-of-string marker has to be added and the new text for the infostring has to be set (abra cadabra, that's it).

PROGS, Professional & Graphical Software
last edited February 11, 1995

Extending ProWesS

BEWARE : only persons already capable of writing programs which use ProWesS should continue. This document is only intended for persons who want to add or modify ProWesS types.

How to write your own types

Shape and Form

A ProWesS type is simply an external module, as is supported by syslib. These modules can easily be built by using some lines in your makefile, similar to these (example for the menu type) :

```
type_menu : type_menu_o core-dll_o
        ${LD} -ms -otype_menu type_menu_o \
        core-dll_o -lpw -lpf -lsms -sxmod
        mkxmod type_menu \"ProWesS External Type Definition\"
```

The `init` bit in the module should be a structure which defines the type and its functionality. This structure is defined in the `PWTypeDef.h` header file. This header file is always included when you include the `PWHandler.h` header file. So a generic definition for the init structure would be :

```
PWTypeDef init={
    PW_TYPEDEF_FLAG, PW_TYPE_CANVAS, PW_TYPE_EVENT_REGION,
    &CanvasCreate, &CanvasChange, &Query, &CanvasRemove,
    &CanvasEntry, &CanvasExit, &CanvasHandle,
    NULL, &CanvasDraw, &CanvasDetermineSize, &DoScale
};
```

This mechanism also allows you to define more than one type in one module file. This is for example used to combine the container, separator and direction types in one module. In this particular case, the definition of the types looks like this :

```
PWTypeDef keep={
    PW_TYPEDEF_FLAG, PW_TYPE_DIRECTION, PW_TYPE_EVENT_REGION,
    &KEEPCreate, NULL, NULL, &Remove, NULL, NULL, NULL, NULL,
    NULL, GetColour, NULL
};

PWTypeDef cont={
    (int)&keep, PW_TYPE_CONTAINER, PW_TYPE_EVENT_REGION,
    &CONTCreate, NULL, NULL, &Remove, NULL, NULL, NULL, NULL,
    &CONTDraw, GetColour, NULL
};

PWTypeDef init={
    (int)&cont, PW_TYPE_SEPARATOR, PW_TYPE_EVENT_REGION,
    &SEPACreate, NULL, NULL, &Remove, NULL, NULL, NULL, &Define,
```

```
    &SEPADraw, &SEPADetermineSize, NULL
};
```

You can see that you can link several types by using the pointer to a different type definition instead of the identifier flag. This example also shows that many of the fields in the type definition can be NULL.

Each type has a name (new names should be registered at PROGS to prevent name clashes). You also have to specify whether the type is used to build region or keypress objects. Apart from that, the type is completely defined by a set of routines which are called when certain events occur.

```
#define PW_TYPEDEF_MODULE "ProWesS External Type Definition"

#define PW_TYPEDEF_FLAG 0x58505753  /* flag for PWXType */

#define PW_TYPE_EVENT_REGION    1   /* region associated with obj
#define PW_TYPE_EVENT_KEYPRESS  2   /* keypress associated with o

typedef struct _PWTypeDef {
    int flag;                       /* flag for external object '

    PWType identifier;              /* type identifier, e.g. 'OUT

    char event_type;                /* identifies type of event a
                                    /* with object of this type *

    /* the routines to handle calls which affect this type of obj
    Error (*PWCreate)(PWObject owner, PWObject *ret, PWEvent even
    Error (*PWChange)(PWObject object, int *taglist);
    Error (*PWQuery)(PWObject object, int what, void *ret);
    Error (*PWRemove)(PWObject object);

    /* handler routines */
    Error (*PWRegionEntry)(PWObject object);
    Error (*PWRegionExit)(PWObject object);
    Error (*PWHandle)(PWObject object, PWWait event);

    /* routine to define the parameters according to the configur
    Error (*PWDefine)(char *name, char *value);

    /* draw routine for object (if it is drawable) */
    Error (*PWDraw)(PWObject object);

    /* routines to handle the size aspects of the object */
```

```
    Error (*PWDetermineSize)(PWObject object);
    Error (*PWScale)(PWObject object, pt xinc, pt yinc);
};
```

The use of each of the member functions is explained in detail below. In principle, each function is called to handle a particular event which may occur. The exception to this is the `PWHandle` function. This function is actually called for all events which don't have a specific function. The event itself is in that case identified in the `event` parameters.

When defining a type, you can determine which information is contained in each object. Each object automatically contains some information which references (amongst other things) the system in which the object is contained, and a description of the object as positioned inside the system. This last reference differs depending on whether the objects are windowing or keypress objects.

To make sure that ProWesS can use this information, the start of the object is fixed, and should always be `PWBaseObject`. All the things which have to be known when writing a ProWesS type are defined when you include the `PWHandler.h` header file.

```
#include "PWHandler.h"

typedef struct _Object {
    PWBaseObject;
    some extra variables, specific to each object
} Object;
```

Events

In ProWesS there are three types of events. Some events have a specific handler routine. Those events are described below. All the other events have to be handled by the `PWHandle` routine which is part of the type. For these events, there are two types. Some events are only generated when you ask to be notified for those events. Some events can always be generated.

Each windowing object has a link to a structure of type `PWEvent`. This sctructure contains a variable called `wait_event`. This variable is a mask

which indicates which events have to be passed and which don't. You can add events by ORing the event code to the value. You should never mask an event which is always generated in this value !

Events which can be enabled/disabled

PW_EVENT_HIT

This event is generated when the user generates a *HIT* inside the object.

PW_EVENT_DO

This event is generated when the user generates a *DO* inside the object.

PW_EVENT_TIMEOUT

In the region, you can also set the timeout which should be used. When this event is enabled and a timeout occurs while the pointer is inside this object, then this event will occur. This event can be useful for the display of information when the user has not moved the pointer for some (short) time.

PW_EVENT_MOVE

This event is generated when the pointer moves (and is still inside the window). This event can for example be used while rubber banding, to display the new position of a line. The pointer position inside the window can be determined using the following lines

```
PWSystem *system=object->event.region->system;
pix ptrxpos=system->ptr_xpos-system->win_xpos-reg->hit_xo
pix ptrypos=system->ptr_ypos-system->win_ypos-reg->hit_yo
```

PW_EVENT_HITDRAG

When this event is activated, ProWesS will check whether the user did a simple *HIT* or started dragging. When the user started dragging, this event will be generated. Often, wyou will then start checking for either move or timeout events to make visible what happens while dragging.

PW_EVENT_DODRAG

This event is similar to `PW_EVENT_HITDRAG`, except that is generated when the user is dragging using a *DO*.

PW_EVENT_DRAGADJUST

While dragging, ProWesS does not allow the pointer to move outside the current object. When the pointer position is adjusted to stay inside the region, and this event is active, it will be generated. This can be used to automatically scroll the contents of the region.

When this event is generated, you can determine how much the pointer was adjusted by reading the `Xdragadjust` and `Ydragadjust` variables in the `PWsystem` structure.

Events which are always active

PW_EVENT_DRAGEND
: When the user ends a dragging operation, this event is generated. Although this event is always active, it can only be generated when you actually activated one of the hit or do dragging events.

PW_EVENT_CATCH
: When a keypress in a window can not be matched to a keypress object in the system, the keypress will be redirected to the current catch object, which can then try to take some action. The key which was pressed can be read from the `PWSystem` structure using the `catch` field.

PW_EVENT_CATCHSTART
: This event is generated when the object is selected as the current catch object for the unmatched keypresses.

PW_EVENT_CATCHEND
: When a new object is selected as catch object, then the previous object will receive this event to indicate that it can no longer receive keypresses.

PW_EVENT_AUTOSIZE
: At most one object in the window can be given a chance to change the window size to match the data which has to be displayed in the object. This object can be designated using the `PW_OBJECT_AUTOSIZE` tag. When an object receives this event, it is intended to check whether the window size has to be adjusted. If it is usefull, the `SSTATUS_WINCHANGE` bit in the `status` field of the `PWSystem` whould be set. The object can check what the maximum increase of the window size is by checking the `Xspaceleft` and `Yspaceleft` fields.

Behind the scenes

To allow the objects to function properly, you get access to a part of the ProWesS data structures. These data structures are described here. All these data structures are defined when you include the `PWHandler.h` file.

Please note that only the fields which have a plus sign at the beginning of the description are allowed to be modified. The other fields should be treated as read only !

```
typedef struct _PWSystem PWSystem;
typedef union {
    struct _PWRegion *region;       /*    region */
    struct _PWKeyPress *keypress;   /* OR keypress */
} PWEvent;

/* -- definition of the dummy object (base of the real object) --

#define OSTATUS_REDRAW      (short)0x0001   /* set when redraw ne
#define OSTATUS_REMOVE      (short)0x0002   /* set when should be
#define OSTATUS_INITIALISED (short)0x0004   /* set when initialis
#define OSTATUS_POINTER     (short)0x4000   /* pointer in object
#define OSTATUS_AUTOSIZE    (short)0x2000   /* increase size to f

#define PWBaseObject \
    PWTypeDef *type;                /* type of object */
    PWEvent event;                  /* event associated with obje
    void *auxiliary;                /* auxiliary parameter for cl
    short status                    /*+object status */

struct _PWDummyObject {
    PWBaseObject;                   /* common start for all objec
};
```

The *status* field contains a mask of statusses for the object (OSTATUS_xxx).

```
typedef unsigned char Scale;        /* scale factor */
typedef int pix;                    /* pixel values */
typedef short PWWait;               /* type for events */

struct _PWRegion {
    PWSystem *system;               /* global data structure for
    PWObject parentObject;          /* parent (owner when creatin
    PWObject owner;                 /* owner/creator of this regi
    struct _PWRegion *parent;       /* parent region */
    struct _PWRegion *next;         /* next region on this level
    struct _PWRegion *prev;         /* previous region on this le
    struct _PWRegion *children;     /* list on next level of nest
    short level;                    /* nesting level (==parent->l
                                    /* parameters for region hand
    PWWait wait_event;              /*+events to wait for */
    TimeOut timeout;                /*+timeout for events */
```

```
                                    /* total area covered by regi
    pt tot_xpos, tot_ypos;          /* origin of region in parent
    pt tot_xsiz, tot_ysiz;          /*+size of region */
                                    /* hit area, only this area i
                                    /* the object type handle rou
    pix hit_off_lft, hit_off_rgt;   /*+margins of hit area in tot
    pix hit_off_top, hit_off_bot;   /*+margins of hit area in tot
    pix hit_xorg, hit_yorg;         /* top left coordinate of are
    pix hit_xend, hit_yend;         /* bottom right coordinate */
                                    /* scale parameters for objec
                                    /* the size of the margins ar
                                    /* area stays the same */
    Scale scale;                    /* scale factor for size (pri
    char windowfit;                 /* secondary direction */
    char composite;                 /*+this is a composite object
    char dragging;                  /* are we dragging in this re
};

struct _PWKeyPress {
    PWSystem *system;               /* global data structure for
    PWObject parentObject;          /* parent (owner when creatin
    PWObject owner;                 /* owner/creator of this keyp
    struct _PWKeyPress *next;       /* next keypress in list of k
    char key;                       /* key which should be presse
    char alternative;               /* alternative for this actio
};

typedef struct _PWRegion PWRegion;
typedef struct _PWKeyPress PWKeyPress;

/* Each type can have a general state, to keep information about
/* Normally the base type is followed by extra type specific info
/* The PWTypeState information is kept in a list with sentinel (t

#define PWBaseTypeState                                         \
    PWType type;                    /* type identifier */   \
    struct _PWDummyTypeState *next  /* next item */

typedef struct _PWDummyTypeState {
    PWBaseTypeState;                /* common start for all types
} PWDummyTypeState;

/* System data structure */

#define PWPublicSystem
    unsigned long status;           /*+system status (values SSTA

    Gstate gstate;                  /* PROforma gstate is used (o
```

```
    Window window;                  /* Window identifier */

    PWObject catchkeys;             /* object which has to catch
    char catch;                     /* key for which catching is
    char activated;                 /* TRUE when system activated

    short ptr_xpos, ptr_ypos;       /* current pointer position *
    pix win_xpos, win_ypos;         /* position of window */
    pix xorg, yorg;                 /* origin of area in window *

    Heap heap;                      /* heap with/for this system

    int Xdragadjust, Ydragadjust;   /* adjustment of pointer whil
    int Xspaceleft, Yspaceleft      /* space left for auto increm

#ifndef _PW_PRIVATE
struct _PWSystem {
    PWPublicSystem;
};
#endif _PW_PRIVATE

#define SSTATUS_REDRAW      0x00000001  /* set when part of windo
#define SSTATUS_WINCHANGE   0x00000004  /* set when window build
#define SSTATUS_PAGESHOW    0x00000040  /* set when PFPageShow sh
```

Create routine

.....

Change routine

.....

Query routine

.....

Remove routine

.....

RegionEntry routine

.....

RegionExit routine

.....

Handle routine

.....

Define routine

.....

Draw routine

.....

DetermineSize routine

.....

Scale routine

.....

ProWesS Core routines

To aid in the development of types, some hooks back into ProWesS are provided. These hooks can be used by linking the 'core-dll_o' file into your

external module. When you use these functions you should include the line

```
#include "PWHandler_h"
```

in your source file.

PWHandleDefine

To make it easier to handle definition constants, the `PWHandleDefine` function is provided. Starting from a table of possible constants and their type, place to store the new value etc. The type for the constant can be either a builtin type, or you can provide a routine to parse this value.

To aid in using the definition constants, you can also provide a post processing routine (e.g. to assure that a certain value is always even), and you can pass the address of a character which is set to `TRUE` when that constant was defined. This can be used to know whether the value as defined should be used, or whether a default value has to be obtained (probably by querying the ProWesS system).

```
typedef struct _Defines {
    char *name;                 /* name of constant, no prefix */
    Error (*Handle)(struct _Defines *possible, char *value);
    void (*Post)(void *data);   /* post processing routine */
    void *data;                 /* place where data should be put
    char *set;                  /* set to TRUE when data set (if
} Defines;

#define PW_DEFINE_COLOUR        ((Error (*)(Defines*,char*))1)
#define PW_DEFINE_BOOLEAN       ((Error (*)(Defines*,char*))2)
#define PW_DEFINE_CHARACTER     ((Error (*)(Defines*,char*))3)
#define PW_DEFINE_SHORT         ((Error (*)(Defines*,char*))4)
#define PW_DEFINE_INT           ((Error (*)(Defines*,char*))5)
#define PW_DEFINE_PT            ((Error (*)(Defines*,char*))6)
#define PW_DEFINE_FONT          ((Error (*)(Defines*,char*))7)

Error PWHandleDefine(char *prefix, Defines *possible, char *name,
```

This function is used in most of the builtin types. For example in the scroll type. A small extract of that code is shown here :

```
Error SetTRUE(Defines *this, char *value)
```

```
{
    *(char *)this->data=TRUE;
    return ERR_OK;
}
Error SetFALSE(Defines *this, char *value)
{
    *(char *)this->data=FALSE;
    return ERR_OK;
}
void cwidth(void *data)
{
    xwidth=PF2xpix(width);
    ywidth=PF2ypix(width);
}

Defines defines[]={
{ "ARROWS-LEFT",            SetTRUE,            NULL,   &arrowsle
{ "ARROWS-RIGHT",           SetFALSE,           NULL,   &arrowsle
{ "BAR-MARGINWIDTH",        PW_DEFINE_PT,       cwidth, &width,
{ "BAR-COLOUR",             PW_DEFINE_COLOUR,   NULL,   &colour,
{ 0 }
};

Error ScrollDefine(char *name, char *value)
{
    return PWHandleDefine("SCROLL-",defines,name,value);
}
```

PWHandleKeyPress

This command can force ProWesS to handle the current keypress in the window. ProWesS always handles keypresses as window actions. Only when no keypress handler object is created for a keypress, the current object has the chance to react to that key. However, a *HIT* or *DO* is an event which can be acted upon by the region handler. In some cases, it may be useful to pass this on to the window (possibly in the form of a <space> or <enter> keypress). Of course, you are free to activate any keypress in the window.

The current keypress in stored in :

```
PWSystem *system;       /* the current system */
...
system->ptr_info.keystroke
```

PWGetType

Get the type definition of another type in the system. This command is needed when you want to use (part of) another already existing type. It can be compared with inheritance in object oriented programming languages. You can thus use the access routines of another type and possibly extend it, or call the existing routines in some cases, and handle some things differently.

An example where this is useful is the implementation of loose items. The loose item needs to get a border when the pointer enters the region. This is no real problem, but there may also be other types of items, like edit text which also needs such a border. However, I don't want to repeat the method of drawing this border for each type, and I want all these borders to look the same. Therefore the item type is used for the entry and exit routines of the type, and the item routine can also determine the size of the border around the hit area. The size and colour of the border can be configured, and the system background colour is used to remove the border.

PWAllocate & PWRelease

These commands allow you to allocate and release some memory for a ProWesS type. It is advised that these routines are used specifically when allocating and releaseing the object itself.

PWSkipTag

Some of the type handler routines have a taglist as parameter. Some of these tags may have to be processed, and some just have to be skipped. The `PWSkipTag` routine skips a tag and all the parameters. This is important because there are many kinds of tags (fixed number of parameters, list or taglist as parameter), and because there has to be a correct method of skipping unrecognised tags.

PWpixel2point, PWpoint2pixel

Get the conversion factors to convert points (PROforma coordinates) into pixels (window coordinates), or vice versa.

```
{
    pt Xpt2pix, Ypt2pix;
    pt Xpix2pt, Ypix2pt;
    pt pix, x, pf;
    short Pix, X;

    PWpixel2point(&Xpix2pt,&Ypix2pt);
    PWpoint2pixel(&Xpt2pix,&Ypt2pix);

    /* convert x (PROforma coor), into equivalent in pixels (pt o
    pix=fixmul(x,Xpix2pt);
    Pix=x/Xpt2pix;
    Pix=pt2short(fixmul(x,Xpix2pt));

    /* convert X (pixel coor, short), into PROforma coor (pt) */
    pf=X*Xpix2pt;
    pf=fixmul(short2pt(X),Xpix2pt);

}
```

PF2xpix, PF2ypix

These routines will convert PROforma coordinates (in pt) into window coordinates (in pixels, short). The pixel coordinates are not rounded, but truncated.

xpix2PF, ypix2PF

These routines will convert window coordinates (in pixels, short) into PROforma coordinates (in pt).

PFxround, PFyround

These routines can truncate PROforma coordinates to an integral number of pixels. You could do this by using the conversion factors returned by the PWpixel2point and PWpoint2pixel routines, or applying xpix2PF(PF2xpix(X)), but these calls are more efficient. Also, doing the

calculations yourself with the conversion factors can cause degradation. For example, in the following function

```
char test(pt x)
{
    pt Xpt2pix, Ypt2pix;
    pt Xpix2pt, Ypix2pt;
    pt y;

    PWpixel2point(&Xpix2pt,&Ypix2pt);
    PWpoint2pixel(&Xpt2pix,&Ypt2pix);

    y=XROUND(x);
    return XROUND(x)==XROUND(y);
}
```

the result would be TRUE if XROUND is defined as

```
#define XROUND(x) PFxround(x)
```

but it would be false with the following definition

```
#define XROUND(x) (Xpix2pt*pt2short(fixmul(Xpt2pix,(x))))
```

QL Today

The new magazine for all QDOS/SMSQ users !

Published by Jochen Merz Software, and edited by Dilwyn Jones, QL Today is a magazine for everyone who has a QL or compatible. It contains listings of events, news, reviews of hardware and software, reports from QL shows and meetings, programming articles, letters, histories of QL alumni and many more subjects. Articles are written by a number of prominent QL users, and (we hope) by the readers too. QL Today also includes a German language supplement for German subscribers.

Jochen Merz has been supplying software and hardware for several years and built up a good reputation for quality and fair trading. The representative in Britain is Miracle Systems Ltd, who take subscriptions and do the distribution.

There are many exciting QL hardware developments currently in the pipeline. QL Today will keep you in touch with all the latest news. Whether you are a beginner or an advanced user, we hope you will find articles of interest in QL Today.

Issue 1 included the following articles :

- Beginners' Basics Part 1,
- My BOOT,
- Small Ads,
- QL Shows,
- Bugs 'n' Fixes,
- Epson Stylus Colour II,
- A worked example of a ProWesS application,
- News Ticker,
- Hardware & Software Updates,

- Use the button frame in BASIC,
- QL Today Deutsch (supplement for German readers),
- Report from the Quanta AGM in Tynemouth,
- News from Quo Vadis Design,
- QL-Not for profit!,
- QL Rodents (article about mouse systems),
- QBranch company profile,
- Hints on using QTPI,
- Beginners' Corner,
- Questions and Answers,
- Review of CueShell,
- DISA version 3 report,
- QPC - a dream comes true (advance news of a software QL emulator running on the PC).

Subscription details

QL Today is published bi-monthly. Our volume begins on 15th May. Subscriptions begin with the current issue at the time of sign up. Subscription rates are as follows.

- Germany : DM 70,-
- Great Britain : DM 60,- or `25.00
- Rest of the World : DM 70,- or `30.00

Payment in DM (drawn on a German bank) can be made by either cheque or Eurocheque. Payment in Sterling (GBP) by cheque (drawn on a British bank). Please make cheques payable to Jochen Merz Software (German office), or to Miracle Systems Ltd (English office). Credit card holders may subscribe by either calling or sending their credit card number and expiry date. Credit cards will be charged in DM (German office) or in Pounds (English office).

News / articles / advertising deadlines

Issue 1 : 15 April
Issue 2 : 15 June

Issue 3 : 15 August
Issue 4 : 15 October
Issue 5 : 15 December
Issue 6 : 15 February

Articles for publication should be on a 3.5 inch disk (DD or HD) in ASCII (plain text), Quill, or Text 87 format. Pictures can be in _SCR (QL screen), and we can also handle GIF and TIF formats. To enhance your article you may wish to include saved screen dumps. Please send a hardcopy of all screens to be included and specify clearly where in the text you would like the screen placed.

German Office and Publisher

Jochen Merz Software, Im stillen Winkel 12, 47169 Duisburg, Germany.
Telephone: +49 203-502011 Fax: +49 203-502012

English Office

Miracle Systems Ltd., 20 Mow Barton, Yate, Bristol, BS17 5NF, Great Britain.
Telephone and Fax: +44 1454-883602

Editor

Dilwyn Jones, 41 Bro Emrys, Tal-Y-Bont, Bangor, Gwynedd, LL57 3YT, Great Britain.
Telephone and Fax: +44 1248-354023

IQLR subscribers

If you have issues outstanding from your IQLR subscription and have not received a refund, read this before subscribing.

If you subscribed through Miracle Systems Ltd (i.e. you were issued an invoice from Miracle Systems Ltd) or through Jochen Merz Software then

you will get the number of QL Today issues free of charge that you were owed by IQLR. If you have an outstanding subscription to IQLR purchased through a different channel then tell us - you will be eligible for half price issues to substitute for the IQLRs you are owed provided you take out a subscription. Contact Miracle Systems Ltd or Jochen Merz Software for more details.

---

written by Dilwyn Jones
last edited July 15, 1996

Global Variables application example

This is a smallish program which allows you to investigate the value of all the *Global Variables* which have been defined, and to change, add or delete global variables.

Global Variables are an operating system extension which is introduced in ProWesS. It is a system which is similar to (but not the same as) environment variables on Unix systems. It allows you to assign a value (a string) to a name. This value can be queried by everybody and can also be changed by anybody. It is mainly used to ease the installation process for programs. It is for example used when loading ProWesS. The device and directory where ProWesS is loaded from is stored in a global variable (`PWSDIR`). The value of this variable is then used to find files.

The libraries (syslib) support the use of global variables when a file is opened. For example, when a file if opened with the name "`$PWDIR_doc_loader_html`", then the "`$PWDIR`" is automatically replaced by its value (e.g. "`win1_pws`").

This program is also a very good example for the ease with which interactions between several parts of the window can be programmed.

The window contains a few items. The two items labeled "constant" and "value", contains the name and value of a constants. These items can be edited at wish. When the name of a *Global Variable* is indicated in the menu at the bottom of the window, then that name and the current value will be displayed in the items just above the menu. A *Global Variable* can be (re)set by indicating the "Set constant" item, or deleted by indicating "Delete constant".

The source code

Here is a run through of the source code. Obviously, it starts by including the header files, and some constants which are used to access the *Global Variables* thing. Also a special macro is defined which helps to catch errors when they occur (hence the name).

```
#include "str.h"
#include "thing.h"
#include "ProWesS.h"

#define catch(x) if (err=(x)) return err; else

#define GLOBAL_NAME "Global Variables"
#define GLOBAL_GET  0x47455420  /* "GET " */
#define GLOBAL_SET  0x53455420  /* "SET " */
#define GLOBAL_DELE 0x44454c45  /* "DELE" */
#define GLOBAL_FRST 0x46525354  /* "FRST" */
#define GLOBAL_NEXT 0x4e455854  /* "NEXT" */
```

The text which is displayed inside the items and as labels is defined separately. This makes it easier to change them (for example to produce a copy of the program in a different language). In fact, they could just as well be made configurable.

```
#define LabelConstant   "constant : "
#define LabelValue      "value : "
#define ItemSet         "Set constant"
#define ItemDelete      "Delete constant"
```

The maximum length of the strings which can be edited (the name and value of the constant) are defined here. These length of the value is not limited by the *Global Variables* thing, but limiting it makes them easier to handle. In fact, the "edline" object (which allows you to edit a string) always works with a fixed length string. This length can be defined when the object is created. Otherwise a default length is used (this default length is configurable).

```
#define MAX_NAME    64
#define MAX_VALUE   256
```

Because we think it useful to make programs re-entrant, you should not use global variables, as their value will be shared between all the copies of the program (especially when the program is loaded as an executable thing). Therefore a global structure is needed which is used to pass the parameters so

that they are accessible in all the functions. The base of this structure can be stored in the ProWesS system.

```
typedef struct {
    PWObject menu;
    PWObject constant, value;
} Global;

/* forward declarations */
Error readall(PWObject object);
Error set(PWObject object);
Error delete(PWObject object);
Error select(PWObject object, char *item);
```

The program starts by creating the outline for the window. This outline includes the title (which is the default, the program name), a wake, quit and a sleep item. The quit item is also activated when is pressed. The action which has to be called for the wake action is defined (re-read which global variables are defined). To allow the event handlers to find the globally used variables, the `Global` structure is stored in the global auxiliary.

```
Error init()
{
    Global g;
    Error err;
    PWObject win, box;

    catch( PWCreate(NULL, &win, PW_TYPE_OUTLINE,
                PW_OUTLINE_SLEEP,
                PW_OUTLINE_QUIT,
                PW_OUTLINE_QUIT_KEYPRESS, 27,
                PW_OUTLINE_ACTION_WAKE, readall,
                PW_GLOBAL_AUXILIARY, &g,
                NULL) );
```

Inside the outline, there are many items. The items are normally all below each other at the first level of nesting inside the outline, so a box is created to change the direction as I want the items to be side by side. Inside this box, there are the two loose items to set and delete a constant. The event handlers which have to be called when the items are indicated are defined. It is also specified that the status of the items should not be changed when they are indicated.

```
catch( PWCreate(win, &box, PW_TYPE_DIRECTION, NULL) );
catch( PWCreate(box, NULL, PW_TYPE_LOOSE_ITEM,
            PW_LOOSE_TEXT, ItemSet,
            PW_LOOSE_CHANGE_STATUS, FALSE,
            PW_LOOSE_ACTION_HIT, set,
            NULL) );
catch( PWCreate(box, NULL, PW_TYPE_LOOSE_ITEM,
            PW_LOOSE_TEXT, ItemDelete,
            PW_LOOSE_CHANGE_STATUS, FALSE,
            PW_LOOSE_ACTION_HIT, delete,
            NULL) );
```

Below these items, there are the two objects to edit the strings with the name and value of the constant. The maximum length of these strings is given. Then the two edline objects are connected with each other to make sure that the user can move the cursor between the two items. This is done using the up and down keys. Also after editing the string in the first edline, the user can automatically modify the value for that constant.

```
catch( PWCreate(win, NULL, PW_TYPE_SEPARATOR, NULL) );
catch( PWCreate(win, &g.constant, PW_TYPE_EDLINE,
            PW_EDLINE_MAXLENGTH, MAX_NAME,
            NULL) );
catch( PWCreate(win, &g.value, PW_TYPE_EDLINE,
            PW_EDLINE_MAXLENGTH, MAX_VALUE,
            NULL) );
catch( PWChange(g.constant,
            PW_EDLINE_EDLINE_AFTER, g.value,
            PW_EDLINE_EDLINE_DOWN, g.value,
            NULL) );
catch( PWChange(g.value,
            PW_EDLINE_EDLINE_UP, g.constant,
            NULL) );
```

Of course, we also need a menu which will contain all the constants which are defined at a given moment. This menu is separated from the rest of the window with a separator line. At least six lines are always visible in the menu. All the items inside the menu are always sorted, using a case independant compare (compare routine is given). The event handler which has to handle the selection of an item is specified, but no item can appear to be selected.

```
catch( PWCreate(win, NULL, PW_TYPE_SEPARATOR, NULL) );
```

```
    catch( PWCreate(win, &g.menu, PW_TYPE_MENU,
                PW_MENU_VISIBLE_LINES, 6,
                PW_MENU_SORT_COMPARE, STRCompareCI,
                PW_MENU_ACTION_SELECT, select,
                PW_MENU_NONE_SELECTED,
                NULL) );
```

The edline objects which were defined a bit higher are not yet labeled. Therefore, the labels are added to the left of the items. To make sure the edlines are as large as possible when the window is scaled, we make sure that the label itself is not scaled.

The labels are added here because otherwise the default ordering of the objects in the window could no longer be used. The alternative solution for this is used when defining the loose items above. These are also positioned side by side, but because they are positioned inside a direction box, the default positioning rule is not hampered (as this is defined to be a structuring object).

```
    catch( PWCreate(win, NULL, PW_TYPE_LABEL,
                PW_POSITION_LEFT_OF, g.constant,
                PW_LABEL_TEXT, LabelConstant,
                PW_SCALE_FACTOR, 0,
                NULL) );
    catch( PWCreate(win, NULL, PW_TYPE_LABEL,
                PW_POSITION_LEFT_OF, g.value,
                PW_LABEL_TEXT, LabelValue,
                PW_SCALE_FACTOR, 0,
                NULL) );
```

Before we can start, we have to fill the menu with all the constants which are defined at the moment. So we call the event handler which will also handle the wake event. Then the window is activated.

```
    readall(win);

    return PWActivate(win);
}
```

To read all the definition constants, an iterator which loops over all the *Global Variables* has to be used. To start we have to extract the global structure from the ProWesS system (the global auxiliary). The menu is then

cleared to remove the old contents. A little loop is then started which iterates over all the constants which are defined. The *Global Variables* system is accessed using the thing system. Each constant of which the name is thus obtained, is then adde in the menu. The menu object will automatically make sure that its contents remains sorted.

```
Error readall(PWObject object)
{
    Error err;
    Global *g;
    char name[MAX_NAME], value[MAX_VALUE];

    PWQuery(object,PW_GLOBAL_AUXILIARY,&g);

    PWChange(g->menu,PW_MENU_CLEAR,NULL);

    err=THINGCall(GLOBAL_NAME,GLOBAL_FRST,3,name,value,MAX_VALUE)
    while (!err)
    {
        catch( PWChange(g->menu, PW_MENU_ADD_COPY, name, NULL) );
        err=THINGCall(GLOBAL_NAME,GLOBAL_NEXT,3,name,value,MAX_VA
    }
    return ERR_OK;
}
```

When a constant in the menu is indicated, the name and value of that constant have to be displayed in the edline objects. So to start, we have to retrieve the object identifiers for the edline objects. These are stored in the `Global` structure which is referenced in the global auxiliary for the window. The value for the constant then has to be queried by calling the *Global Variables* thing. The strings with the name and value of the constant then have to be passed to the edline objects.

```
Error select(PWObject object, char *item)
{
    Error err;
    Global *g;
    char value[MAX_VALUE];

    PWQuery(object,PW_GLOBAL_AUXILIARY,&g);

    err=THINGCall(GLOBAL_NAME,GLOBAL_GET,3,item,value,MAX_VALUE);
    if (err) item="", value[0]='\0';
```

```
    PWChange(g->constant, PW_EDLINE_SET, item, NULL);
    PWChange(g->value, PW_EDLINE_SET, value, NULL);

    return ERR_OK;
}
```

Setting a *Global Variable* is approximately the reverse of the select routine above. After querying the global auxiliary, the strings which are stored in the edline objects have to be obtained. Then a *Global Variable* is defined with the given name and value. To make sure that the menu stays synchronized with the existing variables, the contents of the menu is rebuilt.

```
Error set(PWObject object)
{
    Error err;
    Global *g;
    char name[MAX_NAME], value[MAX_VALUE];

    PWQuery(object,PW_GLOBAL_AUXILIARY,&g);

    PWChange(g->constant,PW_EDLINE_GET,MAX_NAME,name,NULL);
    PWChange(g->value,PW_EDLINE_GET,MAX_VALUE,value,NULL);

    THINGCall(GLOBAL_NAME,GLOBAL_SET,2,name,value);

    return readall(object);
}
```

Deleting a constants is also quite similar with setting one. In this case, the value for the name is irrelevant, but it is adviseable that the edlines are cleared after the constant was deleted. Again, the menu is also rebuilt to stay up to date.

```
Error delete(PWObject object)
{
    Error err;
    Global *g;
    char name[MAX_NAME];

    PWQuery(object,PW_GLOBAL_AUXILIARY,&g);

    PWChange(g->constant,PW_EDLINE_GET,MAX_NAME,name,NULL);

    THINGCall(GLOBAL_NAME,GLOBAL_DELE,1,name);
```

```
    PWChange(g->constant, PW_EDLINE_SET, "", NULL);
    PWChange(g->value, PW_EDLINE_SET, "", NULL);

    return readall(object);
}
```

The makefile

The makefile for this program is quite straightforward. In fact, most of the makefile is standard, as it originates from a simple template makefile. The most import lines are the line which starts with "`OBJ =`". The parameter is a list of all the object files for the application. In this case, the entire program is in one file.

Another important line starts with "`all :`". This lists all the targets in this directory which have to be created. All dependencies are automatically checked and everything is rebuilt when necessary.

The line starting with "`global`" lists first the dependencies, and then the programs which have to be called to build the file. This starts by calling the linker, with all the object files. The output file (-o) is called "global", and the map and symbol table are produced (-ms). All the necessary libraries are included (-lpw -lpf -lsms). Because the program which is built will be an executable, the proper startup file has to be used. This is done with the -sexec parameter.

After the linking stage, some post processing has to be done to make the dataspace of the output file correct and add the program name. This is done with the "mkexec" program which has the file and the program name as parameters. Optionally, an extra parameter with the requested extra amount of dataspace can be passed (the default is 4kB). The program name is enclosed in quotes (a quote has to be preceded by a backslash or the "make" program will discard it). The yen symbol is used to separate the actual program name from an extra comment which will be part of the file.

```
# makefile for ProWesS application software
# possible flags - none define just yet
DEFINES =
```

```
# specify compiler etc
CC = cc
CFLAGS = -c -O
LD = ld
MAC = qmac

OBJ = global_o

all : global

global : ${OBJ}
        ${LD} -ms -oglobal \
        ${OBJ} \
        -lpw -lpf -lsms -sexec
        mkexec global \"global◆v1.00, manipulate \"\"Global Varia

_c_o : ; ${CC} ${CFLAGS} ${DEFINES} $<

_s_o : ; ${CC} -c $<

_asm_rel : ; ${MAC} $<
```

PROGS, Professional & Graphical Software
last edited June 28, 1996
originally published in QL Today May/June 1996

defines for PW_TYPE_APPLIC

APPLIC-SCROLLBAR-LEFT

This constant indicates that the vertical scrollbar has to be displayed to the left of the canvas when present.

APPLIC-SCROLLBAR-RIGHT

This constant indicates that the vertical scrollbar has to be displayed to the right of the canvas when present. This is the default.

APPLIC-SCROLLBAR-ABOVE

This definition constant indicates that if a horizontal scrollbar has to be displayed, it is to be positioned above the canvas which can be scrolled.

APPLIC-SCROLLBAR-BELOW

This definition constant indicates that if a horizontal scrollbar has to be displayed, it is to be positioned above the canvas which can be scrolled. This is the default position.

---

defines for PW_TYPE_DIRSELECT

DIRSELECT-DEVICE

Set a device which should be displayed by each directory select window as an easy to select option to change devices (e.g. "win1_", "flp2_",...)

DIRSELECT-DIRECTORY

Set a directory (including device) which should always be displayed by each directory select window as an easy to select directory. It will be displayed in the window which also lists the subdirectories.

---

defines for PW_TYPE_EDLINE and PW_TYPE_DEDLINE

EDLINE-INK-COLOUR
: Define the RGB colour in which the text in edline objects has to be displayed.

EDLINE-PAPER-COLOUR
: Define the RGB paper colour of the edline objects.

EDLINE-FONT
: Set the PROforma font which should be used by the edline objects.

EDLINE-FONTSIZE
: Set the fontsize in PROforma points which should be used for the text in edline objects.

EDLINE-MAXLENGTH
: Set the default maximum length for the text in edline objects. By default his is 256 (including ending '\0').

EDLINE-ITEMWIDTH
: Set the default minimum size of the edline objects in PROforma points. By default this is 120 (1/6 of the width of the screen).

EDLINE-ALWAYS-TYPE
: This allows you to choose which type of edline should be used. By default, there is a difference between a notmal and a direct edline, but this can be changed. This definition needs a number as parameters. Three values are allowed, 0 (zero) for the default case. When the paremeter is 1 (one), then normal edlines are always used (even when creating a direct edline). When the parameter is 2 (two), then you will always use direct edlines (even when creating the normal variant).

---

defines for PW_TYPE_INFOSTRING

INFOSTRING-INK-COLOUR

Define the RGB colour in which the text in infostring objects has to be displayed.

INFOSTRING-PAPER-COLOUR

Define the RGB paper colour of the infostring objects.

INFOSTRING-FONT

Set the PROforma font which should be used by the infostring objects.

INFOSTRING-FONTSIZE

Set the fontsize in PROforma points which should be used for the text in infostring objects.

---

defines for PW_TYPE_INFOTEXT

INFOTEXT-INK-COLOUR

Define the RGB colour in which the text in infotext objects has to be displayed.

INFOTEXT-PAPER-COLOUR

Define the RGB paper colour of the infotext objects.

INFOTEXT-FONT

Set the PROforma font which should be used by the infotext objects.

INFOTEXT-FONTSIZE

Set the fontsize in PROforma points which should be used for the text in infotext objects.

---

defines for PW_TYPE_iTEM

ITEM-BORDER-COLOUR

Define the corlour which should be used for the border around the (current) item. The colour is given as an RGB colour, so by specifying the red, green and blue components. This will default to the default middleground colour.

SYSTEM-BACKGROUND-COLOUR

Set the RGB colour which should be used to remove the border around the current item. By default, the global background colour is used when this is not defined explicitly.

ITEM-BORDER-WIDTH

Set the width of the border which should be displayed. The value is given in PROforma coordinates, so with a virtual screen size of 720 by 540.

---

defines for PW_TYPE_LABEL

LABEL-INK-COLOUR

The RGB Colour which has to be used to display the label name. The ProWesS middleground colour will be used as default when not defined.

LABEL-FONT

Set the font to display the label name. The ProWesS default font will be used when not defined.

LABEL-FONTSIZE

Set the fontsize to display the label name. The ProWesS default fontsize will be used when not defined.

---

defines for PW_TYPE_LISTSELECT

LISTSELECT-ARROW-COLOUR

Set the colour (RGB) which should be used for the arrow which is displayed in the listselect item. When not specified, the default middleground colour is used.

LISTSELECT-ARROW-SIZE

Set the size for the arrow which is displayed in the listselect item.

---

defines for PW_TYPE_LOOSE_ITEM

LOOSE-ITEM-AVAILABLE-INK-COLOUR
: Set the RGB colour to display the text in the item when the item is available.

LOOSE-ITEM-UNAVAILABLE-INK-COLOUR
: Set the RGB colour to display the text in the item when the item is unavailable.

LOOSE-ITEM-SELECTED-INK-COLOUR
: Set the RGB colour to display the text in the item when the item is selected.

LOOSE-ITEM-AVAILABLE-PAPER-COLOUR
: Set the RGB colour for the background in an available loose item.

LOOSE-ITEM-UNAVAILABLE-PAPER-COLOUR
: Set the RGB colour for the background in an unavailable loose item.

LOOSE-ITEM-SELECTED-PAPER-COLOUR
: Set the RGB colour for the background in a selected loose item.

LOOSE-ITEM-FONT
: Set the font which has to be used to display the text inside the loose item. If this is not defined, the ProWesS default font is used.

LOOSE-ITEM-FONTSIZE
: Set the fontsize to display the text. When this is not defined, the ProWesS default fontsize is used.

LOOSE-ITEM-AUTOREPEAT-TIMEOUT
: Set the timeout value for the test for autorepeat.

---

defines for PW_TYPE_MENU

MENU-BORDER-WIDTH

Set the width of the border which should be displayed around the current item. The value is given in PROforma coordinates, so with a virtual screen size of 720 by 540.

MENU-AVAILABLE-INK-COLOUR

Set the RGB colour to display the text in the item when the item is available.

MENU-UNAVAILABLE-INK-COLOUR

Set the RGB colour to display the text in the item when the item is unavailable.

MENU-SELECTED-INK-COLOUR

Set the RGB colour to display the text in the item when the item is selected.

MENU-AVAILABLE-PAPER-COLOUR

Set the RGB colour for the background in an available loose item.

MENU-UNAVAILABLE-PAPER-COLOUR

Set the RGB colour for the background in an unavailable loose item.

MENU-SELECTED-PAPER-COLOUR

Set the RGB colour for the background in a selected loose item.

MENU-FONT

Set the font which has to be used to display the text inside the loose item. If this is not defined, the ProWesS default font is used.

MENU-FONTSIZE

Set the fontsize to display the text. When this is not defined, the ProWesS default fontsize is used.

MENU-PAPER-COLOUR

Define the colour which should be used as background colour inside the menu. This will default to the system background colour.

MENU-INK-COLOUR

Set the colour which should be used to display the items inside the menu. This defaults to the system foreground colour.

MENU-BORDER-COLOUR

Set the colour which is used to display the border around the current item. The default value is the system middleground colour.

MENU-DISPLAY-ROWS

Indicates whether the items should be displayed by row or by column. The default is by column. The value is either "true" or "false".

---

defines for PW_TYPE_SCROLL

SCROLL-ARROWS-LEFT
: Make sure that the scroll arrows are always displayed to the left of the scroll bar in horizontal scroll objects.

SCROLL-ARROWS-RIGHT
: Make sure that the scroll arrows are always displayed to the right of the scroll bar in horizontal scroll objects.

SCROLL-ARROWS-ABOVE
: Make sure that the scroll arrows are always displayed above the scroll bar in vertical scroll objects.

SCROLL-ARROWS-BELOW
: Make sure that the scroll arrows are always displayed below the scroll bar in vertical scroll objects.

SCROLL-BAR-MARGINWIDTH
: Specify the marginwidth which is used inside the scroll arrow. This margin is always visible around the bar which indicates the visible area.

SCROLL-ARROW-SIZE
: Define the size of the scroll arrows. The size is given in pt (PROforma coordinates).

SCROLL-ARROW-COLOUR
: Define the colour of the scroll arrows. The colour is given by specifying the RGB components. When not specified, the default ProWesS foreground colour is used.

SCROLL-BAR-BACKGROUND-COLOUR
: Set the RGB colour which is used as background in the scrollbar. If this is not defined, then the ProWesS default background colour is used.

SCROLL-BAR-COLOUR
: Set the RGB colour which is used to display the current size and position of the visible area in the scrollbar. This colour is also used to display the scroll arrows. If this colour is not defined, then the ProWesS default middleground colour is used.

---

defines for PW_TYPE_SEPARATOR

SEPARATOR-COLOUR

Set the colour to be used by separator objects and containers. The parameter is an RGB colour, where each component is specified as a percentage. When this colour is not defined, the ProWesS middleground default colour will be used.

SEPARATOR-THICKNESS

Set the thickness of the separator and container objects. The parameter is given in PROforma coordinates. The thickness is always at least one pixel, even if the parameter was zero.

---

defines for ProWesS system

SYSTEM-SHADOW-RIGHT

Defines the width of the shadow at the right of the window, in pixels.

SYSTEM-SHADOW-BOTTOM

Defines the width of the shadow below the window, in pixels.

SYSTEM-BORDER-WIDTH

Defines the width of the width of the border, in pixels.

SYSTEM-BORDER-COLOUR

Defines the border colour, in device colour.

SYSTEM-SCALEBORDER-WIDTH

When a window can be moved or scaled, then the border is extended with the scaleborder. This is the area which has to be indicated to initiate a move or scale. The scaleborder is surrounded by the normal border. The value is in pixels.

SYSTEM-SCALEBORDER-COLOUR

Colour for the scaleborder, in device colour.

SYSTEM-PREVIEW-MOVE

Should the new position of the window be previewed when moving, 'true' or 'false'.

SYSTEM-PREVIEW-SCALE

Should the new size and position of the window be previewed while scaling, 'true' or 'false'.

SYSTEM-PREVIEW-TIMEOUT

Set the timeout value which should be used when a preview should be given of the window during window move or scale. The default value is 10. The unit is ticks. There are between 50 and 72 ticks a second (depending on your system and country). If the window should be previewed (as set by SYSTEM-PREVIEW-MOVE and SYSTEM-PREVIEW-SCALE), then a preview will be shown at the requested interval.

SYSTEM-LOAD-RESIDENT-FONT

ProWesS types may use PROforma fonts to display text. To make sure that fonts don't have to be reloaded all the time, they can be kept resident by specifying the name as parameter. This becomes essential when the fonts have to be loaded from disk. The fonts will be loaded when ProWesS is started. If they have to be loaded later, errors may occur.

However, the types will probably not report these and just continue.

SYSTEM-FONT-CALCULATED

Make sure that the given font is always available and that the characters are pre calculated at the given size. This allow maximum speed when all the often used combinations are available, as the character never have to be rendered. This will take up some memory though (the pre-calculated glyphs are not stored in the cache to make sure they are never released). You first have to pass the size, and then the fontname, e.g.

```
SYSTEM-FONT-CALCULATED 10 Goudy Old Style
```

SYSTEM-SCROLL-DISTANCE

Set amount which should be scrolled for windows which are bigger than the screen (or bigger than the primary). The parameter should be a value in PROforma coordinates.

For each application, the scrolling distance is limited to at most 3/4 of the window size in that direction.

SYSTEM-DRAGTEST-TIMEOUT

Timout value which should be used for testing whether a hit/do or drag action occurs. If this is too small, then some hit or do events could be interpreted as dragging. If it is too high, then the response to a hit or do may be sluggish.

SYSTEM-BACKGROUND-COLOUR

Set the background colour of the window, given as a device independent RGB colour, where 100 100 100 is white, 0 0 0 is black, 100 0 0 is red, 0 100 0 is green and 0 0 100 is blue. This value is also used by many ProWesS types as default background colour.

SYSTEM-FOREGROUND-COLOUR

Set the default ProWesS foreground colour. This is used by many ProWesS types as default colour. It is typically used to as colour to display important information. The colour is given by specifying the RGB components.

SYSTEM-MIDDLEGROUND-COLOUR

Set the default ProWesS middleground colour. This is used by many ProWesS types as default colour. It is typically used to as colour to display guidelines etc. These thing which are not really important, but are displayed to make the window look better and make the programs easier to use.

SYSTEM-FONT
: Set the font which should be used by default by the ProWesS types (and possibly also by some applications).

SYSTEM-FONTSIZE
: Set the default fontsize which should be used by the ProWesS types (and possibly also by some applications).

PROGS, Professional & Graphical Software
last edited December 27, 1996

defines for PW_TYPE_TITLE_ITEM

TITLE-ITEM-INK-COLOUR

Set the ink colour which is used to display the title in the title item. As normal, the colour is specified by the RGB components. If the ink colour is not specified, the ProWesS default foreground colour is used.

TITLE-ITEM-SURROUND-COLOUR

Set the colour which is used as "border" around the title string. The colour is given by stating the percentages of the RGB components. By default the ProWesS middleground colour is used when no colour is explicitly given.

TITLE-ITEM-PAPER-COLOUR

Set the colour which is used as background under the title string. The ProWesS default background colour is used when no specific value has been assigned.

TITLE-ITEM-FONT

Set the font which should be used for the title name. If no value is given, the ProWesS default font is used.

TITLE-ITEM-FONTSIZE

Set the fontsize (in points) to be used for displaying the title string. If this definition constant is not passed, then the ProWesS default fontsize will be used.

---

change

PW_APPLIC_SCROLL_X
: The application window should be scrollable horizontally, but without a scrollbar, no parameters.

PW_APPLIC_SCROLLBAR_X
: The application window should be scrollable horizontally, with a scrollbar, no parameters.

PW_APPLIC_SCROLL_Y
: The application window should be scrollable vertically, but without a scrollbar, no parameters.

PW_APPLIC_SCROLLBAR_Y
: The application window should be scrollable vertically, with a scrollbar, no parameters.

PW_APPLIC_XSCROLL_LIST
: Pass the parameters of this tag on to the horizontal scrollbar in the application window. The parameter should be a list of tags.

PW_APPLIC_YSCROLL_LIST
: Pass the parameters of this tag on to the vertical scrollbar in the application window. The parameter should be a list of tags.

PW_APPLIC_CANVAS_LIST
: Pass the parameters of this tag on to the canvas of the application window. The parameter should be a list of tags.

PW_APPLIC_REDRAW
: Redraw the contents of the application window. This will redraw both the canvas and the scroll bars. In priciple it is more efficient to pass PW_CANVAS_REDRAW to the canvas if the scroll bars don't need to be redrawn as well.

PW_APPLIC_CANVAS_CATCH
: Makes sure that the canvas in the applic is selected as current catch object.

---

query

PW_CANVAS_XORIGIN

Get the current value of the x origin. This value can be in any metric, as determined by the person who set the origin. The origin can only be modified by setting it, or by scrolling (which can be done either directly or by a scroll object). The origin of a canvas is the position at the top left corner of the visible part of the canvas.

PW_CANVAS_YORIGIN

Get the current value of the y origin. This value can be in any metric, as determined by the person who set the origin. The origin can only be modified by setting it, or by scrolling (which can be done either directly or by a scroll object). The origin of a canvas is the position at the top left corner of the visible part of the canvas.

PW_CANVAS_XSIZE

Get the current width of the visible part of the canvas. This is the width in PROforma coordinates (pt).

PW_CANVAS_YSIZE

Get the current height of the visible part of the canvas. This is the heigth in PROforma coordinates (pt).

PW_CANVAS_XSIZE_PIX

Get the current width of the visible part of the canvas. This is the width in pixel coordinates (pt).

PW_CANVAS_YSIZE_PIX

Get the current height of the visible part of the canvas. This is the heigth in pixel coordinates (pt).

change

PW_CANVAS_POINTER

Set the pointer which should be used inside the canvas. The parameter is of type "Sprite *", as defined in "win_h".

PW_CANVAS_ACTION_REDRAW

Set the routine which is used to redraw the canvas. The parameter should be of type "Error (*)(PWObject, CanvasInfo *)". The WindowSub will be set when the redraw routine is called. All the necessary information for redrawing the canvas is supplied in the CanvasInfo parameter.

PW_CANVAS_ACTION_EXIT
: Set the exit routine for the canvas. The parameter should be of type "Error (*)(PWObject, CanvasInfo *)". This routine is called when the pointer exits the area covered by the canvas. It can for example be used to remove the border around something in the canvas.

PW_CANVAS_ACTION_HIT
: Set the routine which should be called when a PW_EVENT_HIT occurs inside the canvas. The parameter has type "Error (*)(PWObject, CanvasInfo *)".

PW_CANVAS_ACTION_DO
: Set the routine which should be called when a PW_EVENT_DO occurs inside the canvas. The parameter has type "Error (*)(PWObject, CanvasInfo *)".

PW_CANVAS_ACTION_MOVE
: Set the routine which should be called when a PW_EVENT_MOVE occurs inside the canvas. The parameter has type "Error (*)(PWObject, CanvasInfo *)".

PW_CANVAS_ACTION_SCALE
: Set the routine which should be called when a the canvas object is scaled. This can be used to extract some information from the size of the canvas. The parameter has type "Error (*)(PWObject, CanvasInfo *)".

PW_CANVAS_TIMEOUT
: Set the timeout value for reading the pointer inside the canvs. The parameter should be of type "TimeOut" as defined in "io_h". The timeout can be used for example to draw a preview of an action. The PW_EVENT_TIMEOUT is only triggered when no other event has occured during the duration set with this tag.

PW_CANVAS_ACTION_TIMEOUT
: Set the routine which should be called when a timeout occurs inside the canvas. Obviously, this only works when a timeout is set for the canvas. The parameter has type "Error (*)(PWObject, CanvasInfo *)".

PW_CANVAS_SIZE_PIX
: Set the size of the canvas on screen. Two parameters are required, the x and y size, both positive integers in pixels.

PW_CANVAS_XSIZE_PIX
: Set the width of the canvas on screen. One parameters is required, the width, a positive integer in pixels.

PW_CANVAS_YSIZE_PIX
: Set the height of the canvas on screen. One parameters is required, the height, a positive integer in pixels.

PW_CANVAS_SIZE
: Set the size of the canvas on screen. Two parameters are required, the x and y size, both PROforma coordinates in "pt".

PW_CANVAS_XSIZE
: Set the width of the canvas on screen. One parameters is required, the width, a PROforma coordinate in "pt".

PW_CANVAS_YSIZE
: Set the height of the canvas on screen. One parameters is required, the height, a PROforma coordinate in "pt".

PW_CANVAS_ORIGIN
: Set the current value of the x and y origin. The two parameter values can be in any metric. The origin of a canvas is the position at the top left corner of the visible part of the canvas. It could be in PROforma coordinates, lines, pixels, or anything else. The origin is directly passed on to application programmer via the CanvasInfo structure in the action handlers. The default origin is (0,0).

PW_CANVAS_XORIGIN
: Set the current value of the x origin. The parameter value can be in any metric. The origin of a canvas is the position at the top left corner of the visible part of the canvas. It could be in PROforma coordinates, lines, pixels, or anything else. Th origin is directly passed on to application programmer via the CanvasInfo structure in the action handlers.

PW_CANVAS_YORIGIN
: Set the current value of the y origin. The parameter value can be in any metric. The origin of a canvas is the position at the top left corner of the visible part of the canvas. It could be in PROforma coordinates, lines, pixels, or anything else. Th origin is directly passed on to application programmer via the CanvasInfo structure in the action handlers.

PW_CANVAS_XSCROLL
: Increment the x origin of the canvas with the parameter. The canvas will automatically be redrawn when control is next handed back to ProWesS.

PW_CANVAS_YSCROLL
: Increment the y origin of the canvas with the parameter. The canvas will automatically be redrawn when control is next handed back to ProWesS.

PW_CANVAS_REDRAW
: Tell ProWesS that the canvas object should be redrawn when control is next handed back to ProWesS.

PW_CANVAS_ACTION_HITDRAG
: Set the action routine which should be called when the user starts dragging with a hit. The parameter is of type "Error (*)(PWObject, CanvasInfo *)".

PW_CANVAS_ACTION_DODRAG
: Set the action routine which should be called when the user starts dragging with a do. The parameter is of type "Error (*)(PWObject, CanvasInfo *)".

PW_CANVAS_ACTION_DRAGEND
: Set the action routine which should be called when the user stops dragging. The parameter is of type "Error (*)(PWObject, CanvasInfo *)".

PW_CANVAS_ACTION_DRAGADJUST
: When the user is dragging, ProWesS makes sure that the pointer will not move out of the current region (it won't even pass control to children). To prevent the pointer from exiting, the pointer position may be adjusted. Such an event can be trapped by the canvas with this tag. The parameter is of type "Error (*)(PWObject, CanvasInfo *, pt xdist, ydist)".

PW_CANVAS_ACTION_CATCH
: Set the action routine which should be called when the canvas has to catch a keypress. The parameter is of type "Error (*)(PWObject, CanvasInfo *, char)". The last parameter is the key which was pressed.

PW_CANVAS_ACTION_CATCHSTART
: Set the action routine which should be called when the canvas receives a catchstart event. The parameter is of type "Error (*)(PWObject, CanvasInfo *)".

PW_CANVAS_ACTION_CATCHEND
: Set the action routine which should be called when the canvas receives a catchend event. The parameter is of type "Error (*)(PWObject, CanvasInfo *)".

CanvasInfo

All the client routines which can be called by the canvas type are passed a special structure, which contains a lot of relevant information about the canvas. This structure is known when {\tt ProWesS_h} is included, and is defined as follows :

```
typedef struct {
    pt xorg, yorg;          /* coordinate at topleft in canvas */
                            /* could be any type of size sizeof(i
                            /* PROforma coordinates */
    pt xsiz, ysiz;          /* size of area */
    pt xpos, ypos;          /* pointer position in area */
    Gstate gstate;          /* Gstate to draw in */
                            /* window coordinates */
    short xpixsiz, ypixsiz; /* size of area */
    short xpixorg, ypixorg; /* origin of area in window */
    short xpixpos, ypixpos; /* pointer position in area */
    Window window;          /* window to draw in */
                            /* conversion factors */
    pt Xpix2pt, Ypix2pt;    /* pixel to point conversion */
    pt Xpt2pix, Ypt2pix;    /* point to pixel conversion */
    short zero;             /* always zero */
    } CanvasInfo;
```

To allow redrawing (part of) the canvas, the WindowSub is always set to cover the area of the canvas when this structure is passed to a routine.

---

change

PW_DIRSELECT_SET

Set the current directory for the directory select object. The parameter is of type "char *". By default the directory is the the null string. If the directory is changed to the null string, the data default will be used as directory.

PW_DIRSELECT_TITLE_TEXT

Set the text which has to appear in the title bar of the directory select window. The parameter is of type "char *". By default the title is "dir select".

PW_DIRSELECT_ACTIVATE

Activate the directory select window. This tag has no parameters. The directory select object is a keypress object. A keypress can be given to it and the window is displayed when that key is pressed. However, the window can also explicitely be asked for by passing this tag to the object.

PW_DIRSELECT_ACTION

Set a function which should be called when the directory select window is closed. The parameter is of type "Error (*)(PWObject obj, char *directory)". This routine is called with the name of the selected directory passed to it.

PW_DIRSELECT_GET

Get the current directory for the directory select object. Two parameters are needed, the length of the buffer (an "int"), and the pointer to the buffer ("char *"). If the text in the object is larger than the buffer, the buffer will be filled as much as possible.

All the tags which are valid for edline objects, can als be applied to direct edline (dedline) objects.

create

PW_EDLINE_MAXLENGTH

Set the maximum length of the string which can be edited in an edline object. The parameter is of type "int".

PW_EDLINE_VISIBLE_LINES

An edline can have several lines. They are independent of each other (i.e. there is no word wrap) and you can get to/from each line with the up/down cursor keys. This tag determines how many lines there are in the edline. They are normally all visible in the window. Beware, you cannot change the number if lines later on - if this tag is not given, the edline only has one line... The parameter is the number of lines. The length of each line is limited by the maxlength setting above.

change

PW_EDLINE_SET

Set the string which should be presented for editing in the edline object. If the string is longer than the maximum length, then as much as possible will be used. The parameter is of type "char *". For multi-line edlines, this will set the current line for the object. That is the line which was last edited by the user.

PW_EDLINE_SET_LINE

Set the string which should be presented for editing in any line of the edline object. The parameters are the string to set ("char *") and the line to set it to ("int"). If the string is longer than the maximum length, then as much as possible will be used. The line number is forced in range.

PW_EDLINE_SET_ARRAY

Use an array to set the strings for the edline. There are three parameters : the number of lines to fill in (which should be the number of elements in the array), the maximum length of each string in the array, and the array itself. The type makes sure that no overflow can occur if there are more lines in the array than in the edline, or if the length of each line in the edline is less than that of each element in the array.

PW_EDLINE_GET

Get the string which is contained in the edline. Two parameters are needed, the length of the buffer (an "int"), and the pointer to the buffer ("char *"). If the text in the object is larger than the buffer, the buffer will be filled as much as possible. For edlines which contains more than one line, you will modify the current line.

PW_EDLINE_GET_LINE

This will set the current line, and then fill in a string with the value of that line. You have to give three parameters, the length of the buffer ("int"), the base of the buffer ("char *") and the line number. The line number is forced in range. If the text in the object is larger than the buffer, the buffer will be filled as much as possible.

PW_EDLINE_GET_ARRAY

Fill an array with the lines from the edline object. There are three parameters : the number of lines to fill in (which should be the number of elements in the array), the maximum length of each string in the array, and the array itself. The type makes sure that no overflow can occur if there are more lines in the edline than in the array, or if the length of each line in the edline is larger than that of each element in the array.

PW_EDLINE_WIDTH

Set the size of the edline object, one parameter in PROforma points.

PW_EDLINE_WIDTH_PIX

Set the size of the edline object, one parameter in pixels ("int").

PW_EDLINE_WIDTH_FS

Set the size of the edline object, on parameter in fixpoint ("pt"). The window size is calculated as the parameter times the fontsize used for the text in the object.

PW_EDLINE_KEYPRESS

Attach a keypress with the edline. The pressing of the key will be equivalent with a PW_EVENT_HIT on the item, so the item can be edited. The parameter is the primary keypress, which is of type "char".

PW_EDLINE_ACTION_AFTER

Set the routine which should be called when the user finished editing the edline (when the user pressed), as a post processing routine. The parameter should be of type "Error (*)(PWObject object)". This can be used to modify some part of the system according to the data entered.

PW_EDLINE_ACTION_DO

Set the routine which should be called when the user indicates the edline with a PW_EVENT_DO. If such a routine exists, then this routine will be called, and the item in the edline cannot be edited. If there is no ACTION_DO routine, then a PW_EVENT_DO is treated the same as a PW_EVENT_HIT. The parameter should be of type "Error (*) (PWObject object)". This can be used to modify some part of the system according to the data entered.

PW_EDLINE_EDLINE_AFTER

Some navigation is possible with edlines. Several edline objects can be linked together so that you can edit several items and keep your hands at the keyboard. This tag allows you to set the edline which should be edited after this one (when pressing ENTER). The parameter is of type "PWObject" and has to be a possible catch object ! Of course the ACTION_AFTER routine is called before moving to the next object.

PW_EDLINE_EDLINE_NEXT

Some navigation is possible with edlines. Several edline objects can be linked together so that you can edit several items and keep your hands at the keyboard. This tag allows you to set the edline which should be edited as the next one, which can be reached by pressing TAB. The parameter is of type "PWObject" and has to be a possible catch object ! Of course the ACTION_AFTER routine is called before moving to the next object.

PW_EDLINE_EDLINE_PREV

Some navigation is possible with edlines. Several edline objects can be linked together so that you can edit several items and keep your hands at the keyboard. This tag allows you to set the edline which should be edited as the previous one, which can be reached by pressing SHIFT TAB. The parameter is of type "PWObject" and has to be a possible catch object ! Of course the ACTION_AFTER routine is called before moving to the next object.

PW_EDLINE_EDLINE_UP

Some navigation is possible with edlines. Several edline objects can be linked together so that you can edit several items and keep your hands at the keyboard. This tag allows you to set the edline which should be edited as the above one, which can be reached by pressing the UP cursor key. The parameter is of type "PWObject" and has to be a possible catch object ! Of course the ACTION_AFTER routine is called before moving

to the next object.

PW_EDLINE_EDLINE_DOWN

Some navigation is possible with edlines. Several edline objects can be linked together so that you can edit several items and keep your hands at the keyboard. This tag allows you to set the edline which should be edited as the below one, which can be reached by pressing the DOWN cursor key. The parameter is of type "PWObject" and has to be a possible catch object ! Of course the ACTION_AFTER routine is called before moving to the next object.

PW_EDLINE_ACTIVATE

Activate the edline object to which this tag is passed. The edline object will wait for input from the user. When this tag is passed to a direct edline object, then that object will be selected as catch all keypresses object. So the edline is ready for input from the user. This tag has no parameter. The actions are in principle the same as hitting the edline.

PW_EDLINE_ACTION_TEST

This tag allow you to define a routine which can constrain which keypresses should be reacted to by the edline object, or even give a special meaning to certain keypresses. The tag needs one parameter, which is a function of type "Error (*)(PWObject object, EdlineInfo *info)". This function is called each time a key is pressed inside the edline object, and once before the editing starts (with a nul key).

To make it somewhat easier to use, there are some standard builtin routines to constrain the text in the edline object to numbers, either natural numbers (`PW_EDLINE_ACTION_TEST_NATURAL`), integers same as natural, but including sign) (`PW_EDLINE_ACTION_TEST_INTEGER`), of floating point numbers (also including decimal point) (`PW_EDLINE_ACTION_TEST_FLOAT`).

The test routine is passed a structure which contains details about the current value of the string, its maximum length, current length, the cursor position and the key which was pressed. Depending on these values, the routine can do one of three things :

- Accept the keypress. The function decides that the keypress is valid and should be reacted to in the standard way. You can also change

the key to something else (e.g. convert to upper case). In this case, the function should return `PW_EDLINE_TEST_ACCEPT`. Only the `key` may be modified in the `EdlineInfo` structure.

- Discard the keypress. The function decides that the key which was pressed is not allowed in this edline. This can for example be used to make sure no letters are inserted in a number, or that you can only have one decimal point in a floating point number. The function should return `PW_EDLINE_TEST_NOACCEPT` and should not modify the `EdlineInfo` structure.
- Give a different meaning to the key which was pressed. You can change the string which is displayed and/or the current cursor position. The function should return `PW_EDLINE_TEST_CHANGE`. The `EdlineInfo` may be modified (this is intended). However, you should take care that no more than `maxlength` bytes are used in the string, and the `length==strlen(string)`.

PW_EDLINE_PRINT_ESCAPE

This tag allows you to specify whether any escape code in the string have to be displayed in full (`TRUE`) or whether they should be displayed as the character they represent (`FALSE`). On parameter is needed, the new status. The default is `FALSE`.

---

create

PW_FILESELECT_MULTIPLE

Make sure that multiple files can be selected in the file select window.

query

PW_FILESELECT_BOX

To allow the user to modify the behaviour and look of the fileselect window, there is always an empty box between the outline and the rest of the winwod. The user may put some objects in this box. The object identifier is returned by this query, so the value filled in is of type "PWObject".

PW_FILESELECT_OUTLINE

To allow the user to modify the behaviour and look of the fileselect window, the user is allowed to change the bahaviour and look of the outline object in the window. Therefore, the user can get the object identifier of the outline with this query tag. The value which is filled in is of type "PWObject".

PW_FILESELECT_MENU

When the user has modified the outline, it is possible to include a 'Do' item which should use the indicated files for its action. Therefore, you can query the PWObject id of the menu which contains all the files.

PW_FILESELECT_DIRECTORY

When the user has modified the outline, it is possible to include a 'Do' item which should use the indicated files for its action. Apart from knowing which files have been indicated, you probably also need to know in which directory to find the files. This can be queries with this tag. A string with maximum length IO_MAXDIRECTORY is copied into the given address.

PW_FILESELECT_FILENAME

When the fileselect object only allows you to indicate one file, then this query tag can be used to get the name of the file which was selected. A string with maximum length IO_MAXFULLNAME is copied into the given address. If the fileselect window allows you to select multiple files, then `ERR_IPAR` is returned.

change

PW_FILESELECT_TITLE_TEXT

Set the text which has to appear in the title bar of the file select window. The parameter is of type "char *". By default the title is "file select".

PW_FILESELECT_ACTIVATE

Activate the file select window. This tag has no parameters. The file select object is a keypress object. A keypress can be given to it and the window is displayed when that key is pressed. However, the window can also explicitely be asked for by passing this tag to the object.

PW_FILESELECT_ACTION

Set a function which should be called when the file select window is closed. The parameter is of type "Error (*)(PWObject obj, char *directory, void *extra)". The PWObject which is passed back is the fileselect object, the directory is the current directory in the fileselect window, which should be used as searchpath when opening the file. The extra parameter depends on the fileselect object. If only one file can be selected, then this is a "char *", which contains the filename which was selected. If multiple files can be selected, it is of type "PWObject", being the object of type PW_TYPE_MENU which contains the list of files. The PW_MENU_SELECTED_FIRST and PW_MENU_SELECTED_NEXT queries can be used to deteremine which files have been selected.

PW_FILESELECT_FILENAME

Set the default filename which should be suggested. The parameter is of type "char *". If PW_FILESELECT_MULTIPLE was passed during creation of the fileselect object, then nothing will happen.

PW_FILESELECT_DIRECTORY

Set the directory which should be displayed in the fileselect menu. If the directory is "", then it will default to the data directory (cf. DEVDataGet and DEVDataSet). The parameter has type "char *".

PW_FILESELECT_EXTENSION

Set the extensions which should be selected upon in the display of the fileselect window. The parameter has type "char *".

PW_FILESELECT_NOT_STATUS

Set the status for the "not" item in the fileselect window. This tag needs on parameter, any of PW_STATUS_AVAILABLE (default), PW_STATUS_SELECTED or PW_STATUS_UNAVAILABLE.

PW_FILESELECT_SHOWSUB

This tag (which has either TRUE or FALSE as parameter) determines whether subdirectories should be displayed in the fileselect window.

---

change

PW_INFOSTRING_TEXT

Set the text which should be displayed inside the infostring object. The parameter should be of type "char *". The parameter is an ordinary c string in which the lines are separated by '\n' symbols.

PW_INFOSTRING_TEXT_UPDATE

Set the text which should be displayed, and displey it immediately (don't wait until control is passed back to ProWesS). This allows you to display what is happening in a window. You have to pass the new text which has to be displayed. When the text is larger than can be displayed in the object, then the window will NOT be resized !

PW_INFOSTRING_AUTOSIZE

The parameter is either TRUE or FALSE. By default, the value is TRUE. When autosize is TRUE, then the size of the items in the window are redetermined when the text in the infostring changes. If autosize is FALSE then the text is always displayed as best possible in the item, and the window is not notified when the text changes.

---

change

PW_INFOTEXT_TEXT

Set the text which should be displayed inside the infotext object. The parameter should be of type "char *". The parameter is an ordinary c string in which the lines are separated by '\n' symbols.

PW_INFOTEXT_WIDTH

Set the minimum width of the infotext object, one parameter in PROforma points ("pt").

PW_INFOTEXT_WIDTH_PIX

Set the minimum width of the infotext object, one parameter in pixels ("int").

PW_INFOTEXT_WIDTH_FS

Set the minimum width of the infotext object, one parameter in fixpoint ("pt"). The window size is calculated as the parameter times the fontsize used for the text in the object.

PW_INFOTEXT_LINES

Set the minimum number of visible lines in the infotext object.

---

change
PW_KEYPRESS_ACTION

Set the action routine which should be called when the primary or secondary keypress associated with this object is activated. The parameter should be of type "Error (*)(PWObject)". If there is no action associated with the keypress object, then a PW_EVENT_HIT will be generated in the parent object (if that exists).

---

change

PW_LABEL_TEXT

Set the text of the label. The parameter should be of type "char *". The text should not contain '\n'.

PW_LABEL_UNDERLINE

Indicate that the specified letter has to be underlined in the label. This normally indicates that the letter can be used to select the item which is labelled. This tag needs one parameter, the character which has to be underlined. The label will underline the character with the same case if it exists in the text, or otherwise it will underline the same character in different case. (This is consistent with the fact that keystrokes are case dependent in ProWesS).

---

query

PW_LISTSELECT_CURRENT
: Get the current value for the listselect item. The return value is of type "char *" and can be NULL.

PW_MENU_xxx
: All the query tags for the menu type are also accepted. They will return the data for the items menu in the listselect object.

change

PW_LISTSELECT_ADD
: Add an extra choice to the listselect menu. If the menu is not sorted, the item will be added at the end. The parameter should be of type "char *". The pointer to the string is copied, so the memory which contains the string should persist.

PW_LISTSELECT_ADD_COPY
: Add an extra choice to the listselect menu. If the menu is not sorted, the item will be added at the end. The parameter should be of type "char *". The string is copied in some memory allocated by the menu.

PW_LISTSELECT_ADD_ARRAY
: Add all strings from an array of string to the listselect menu. If the menu is not sorted, then the items are added at the end, in the same order as in the array. This tag requires three parameters, the number of strings in the array, the length of each string, and the base of the array (note that the first two parameters are the indexes used when declaring the array). The items in the menu will just point to the position in the array, so the array should persist.

PW_LISTSELECT_CLEAR
: Clear the choices for the listselect menu, however, the current value is not affected. All the items will be removed from the list. This tag does not need a parameter.

PW_LISTSELECT_SORT_COMPARE
: This tag allows you to pass a compare routine to the listselect menu, for example, `STRCompareCI` or `STRCompareCD`. It requires one parameter of type "Error *rout(char *str, char *with, int *result)". If the parameter is NULL then the listselect menu is not sorted.

PW_LISTSELECT_SIZE

Set the minimum width and height of the listselect item. The tag needs two parameters, the x and y size, both in PROforma coordinates (pt). Setting the size automatically also sets autosize to FALSE.

PW_LISTSELECT_XSIZE

Set the minimum width of the listselect item. The tag needs a PROforma point as parameter. Setting the size automatically also sets autosize to FALSE.

PW_LISTSELECT_YSIZE

Set the minimum height of the listselect item. The tag needs a PROforma point as parameter. Setting the size automatically also sets autosize to FALSE.

PW_LISTSELECT_AUTOSIZE

The parameter is either TRUE or FALSE. By default, the value is TRUE. When autosize is TRUE, then the size of the listselect item will automatically be redetermined when the value of the item has changed. If autosize is FALSE and no size is set explicitly, then the size of the text at the time when the window is first activated is used.

PW_LISTSELECT_CURRENT

Set the current value for the listselect object. When there is no current value when the listselect object is displayed, then the first choice from the listselect menu will become the current value. It is preferred that the value is one of the choices for the object, but this is not obligatory. The parameter is of type "char *". however, the contents of the string should be persistent (the value is not copied).

PW_LISTSELECT_ACTION_SELECT

Set an action routine which should be called when the value for the listselect object changes. The parameter is of type "Error (*)(PWObject object, char *item)". this routine can be called more than once in succession for the same item, and the item can also be `NULL`, indicating that the listselect object has no current value.

PW_LISTSELECT_TITLE

When you indicate a listselect item, a menu of choices is displayed. This menu by default does not have a title, but if you want, you can set the title with this tag. the parameter is of type "char *".

PW_LISTSELECT_COMMENT

The listselect object displays a choice menu when it is indicated. If you want, this menu can contain an infostring object at the top. This tag

allows you to specify the text which should be displayed in that item. The parameter is of type "char *". The text can contain several (newline separated) lines.

PW_LISTSELECT_KEYPRESS

Set the keypress which allows immediate selection of the listselect object. When that key is pressed, the menu which allows you to display a new value is displayed.

---

query

PW_LOOSE_STATUS

Get the current status of the loose item. The status can be either PW_STATUS_AVAILABLE, PW_STATUS_UNAVAILABLE or PW_STATUS_SELECTED.

PW_LOOSE_TEXT

Get a pointer to the text which is displayed inside the loose item. This text is read only !

change

PW_LOOSE_STATUS

Set the current status of the loose item. The parameter can be either PW_STATUS_AVAILABLE, PW_STATUS_UNAVAILABLE or PW_STATUS_SELECTED. If the new status if different from the old, then the item will be redrawn when control is handed back to ProWesS. By default, an item is PW_STATUS_AVAILABLE.

PW_LOOSE_TEXT

Set the text which should be displayed inside the loose item. The parameter is of type "char *". The pointer is copied by the object, so the memory which contains the text should be retained ! When autosize is true, then the size of the loose item will be redetermined when control is handed back to ProWesS.

PW_LOOSE_TEXT_COPY

Set the text which should be displayed inside the loose item. The parameter is of type "char *". The text is copied into a piece of memory which is allocated (and released) by the loose items itself. When autosize is true, then the size of the loose item will be redetermined when control is handed back to ProWesS.

PW_LOOSE_CHANGE_STATUS

The parameter is either TRUE or FALSE. By default, the value is TRUE. The status of the loose item is only changed when change status is TRUE. In this case a PW_EVENT_HIT will switch between PW_STATUS_AVAILABLE and PW_STATUS_SELECTED. In the case of a PW_EVENT_DO, the status will change to PW_STATUS_SELECTED. When the loose item is PW_STATUS_UNAVAILABLE, the status is not changed

automatically (actually, not even a border will be drawn around the item).

PW_LOOSE_WINDOW_DO
: The parameter is either TRUE or FALSE. By default it is FALSE. When the window do status is TRUE, then the keypress is also handled by the system (thus a keypress object can react to it).

PW_LOOSE_CENTER_ITEM
: The parameter is either TRUE or FALSE. By default, the value is TRUE. If the value is TRUE, then the text will be draw at the centre of the loose item, else it is draw at the top left corner.

PW_LOOSE_AUTOSIZE
: The parameter is either TRUE or FALSE. By default, the value is TRUE. When autosize is TRUE, then the size of the loose item will automatically be redetermined when the text inside the item is changed. If autosize is FALSE and no size is set explicitly, then the size of the text at the time when the window is first activated is used.

PW_LOOSE_ACTION_HIT
: Set the routine which should be called when the loose item reacts to a PW_EVENT_HIT. The parameter should be of type "Error (*) (PWObject object)".

PW_LOOSE_ACTION_DO
: Set the routine which should be called when the loose item reacts to a PW_EVENT_HIT. The parameter should be of type "Error (*) (PWObject object)". If no do action exists for the loose item (or it is NULL), the hit action will be called.

PW_LOOSE_ACTION_DRAW
: Set a draw action for the loose item, the parameter is of type "Error (*) (PWObject object, Gstate gstate, pt xsiz, pt ysiz)". As usual the SubWindow is set to cover the hit area of the loose item. The text will already be drawn. This makes it possible (in combination with the tags to set the size) to draw icons in loose items.

PW_LOOSE_SIZE
: Set the minimum width and height of the loose item. The tag needs two parameters, the x and y size, both in PROforma coordinates (pt). Setting the size automatically also sets autosize to FALSE.

PW_LOOSE_XSIZE
: Set the minimum width of the loose item. The tag needs a PROforma

point as parameter. Setting the size automatically also sets autosize to FALSE.

PW_LOOSE_YSIZE

Set the minimum height of the loose item. The tag needs a PROforma point as parameter. Setting the size automatically also sets autosize to FALSE.

PW_LOOSE_KEYPRESS

Attach a keypress with the loose-item. The pressing of the key will be equivalent with a PW_EVENT_HIT on the item. The parameter is the primary keypress, which is of type "char". When a keypress is assigned to a loose item, than that character will be underlined in the text. When possible, the character with the same case is underlined, or otherwise a character with differing case.

PW_LOOSE_AUTOREPEAT

When tag tah has been passed to the object, then autorepeat will be activated on the *HIT* and *DO* as well. This tag does not have a parameter.

PW_LOOSE_OFFSET

This tag allows you to define an offset for the display of the text in the loose item. It is only valid when the text is not centered. The default is to leave one pixel at the left. This tag requires two parameters, the offset from the left, and the offset from the top. The parameters are in PROforma points (device independent).

PW_LOOSE_OFFSET_PIX

This tag allows you to define an offset for the display of the text in the loose item. It is only valid when the text is not centered. The default is to leave one pixel at the left. This tag requires two parameters, the offset from the left, and the offset from the top. The parameters are in pixels (integer).

---

create

PW_MENU_KEYPRESSES

Set the keypresses which should be used to allow the user to scroll the menu without indicating the scroll items. This tag needs four parameters, first the keypresses for scrolling a line up and down, then the keypresses for scrolling a page up and down. If this tag is not passed, then the standard keypresses for scrolling are defined (<ALT up/down> and <ALT SHIFT up/down>).

query

PW_MENU_FIRST

Get the first item in the menu. This query should be used to initialise cycling over all items in the menu. The first item will become the current item.

PW_MENU_NEXT

Get the next item in the menu. This query should only be used if the current item was initialised using a PW_MENU_FIRST query. The returned item will become the current. If all items in the menu have been returned, then a NULL will be returned and the current item is undefined (can be anything).

PW_MENU_AVAILABLE_FIRST

Get the first available item in the menu. This query should be used to initialise cycling over all available items in the menu. The returned item will become the current item.

PW_MENU_AVAILABLE_NEXT

Get the next available item in the menu. This query should only be used if the current item was initialised using a PW_MENU_AVAILABLE_FIRST query. The returned item will become the current. If all available items in the menu have been returned, then a NULL will be returned and the current item is undefined (can be anything).

PW_MENU_SELECTED_FIRST

Get the first selected item in the menu. This query should be used to initialise cycling over all selected items in the menu. The returned item will become the current item.

PW_MENU_SELECTED_NEXT

Get the next selected item in the menu. This query should only be used if the current item was initialised using a PW_MENU_SELECTED_FIRST query. The returned item will become the current. If all selected items in the menu have been returned, then a NULL will be returned and the current item is undefined (can be anything).

PW_MENU_STATUS_CURRENT
: Get the status of the current item in the menu.

PW_MENU_FIRST_NUMBER
PW_MENU_NEXT_NUMBER
PW_MENU_AVAILABLE_FIRST_NUMBER
PW_MENU_AVAILABLE_NEXT_NUMBER
PW_MENU_SELECTED_FIRST_NUMBER
PW_MENU_SELECTED_NEXT_NUMBER
: Similar to the query tags which do not end in "_NUMBER", except that they do not return a pointer to the item, but the number of the item in the menu. The numbering starts at zero. This can be usefull especially when `PW_MENU_ADD_ARRAY` was used, to just get the index of the item in the array (if the menu is *not* sorted that is).

PW_MENU_AVAILABLE_NUMBER
PW_MENU_UNAVAILABLE_NUMBER
PW_MENU_SELECTED_NUMBER
: Count the number of items in the menu with the given status. The result is of type `int`.

change

PW_MENU_ADD
: Add an item inside the menu. If the menu is not sorted, the item will be added at the end. The parameter should be of type "char *". The pointer to the string is copied, so the memory which contains the string should persist. The current item, as last returned with one of the queries, is reset when adding something to the menu.

PW_MENU_ADD_COPY
: Add an item inside the menu. If the menu is not sorted, the item will be added at the end. The parameter should be of type "char *". The string is copied in some memory allocated by the menu. The current item, as last returned with one of the queries, is reset when adding something to the

menu.

PW_MENU_ADD_ARRAY

Add all strings from an array of string to the menu. If the menu is not sorted, then the items are added at the end, in the same order as in the array. This tag requires three parameters, the number of strings in the array, the length of each string, and the base of the array (note that the first two parameters are the indexes used when declaring the array). The items in the menu will just point to the position in the array, so the array should persist. The current item, as last returned with one of the queries, is reset when adding something to the menu.

PW_MENU_ITEMWIDTH

Specify the width of each item in the menu. The parameter is of type "pt", the width in PROforma coordinates.

PW_MENU_ITEMWIDTH_PIX

Specify the width of each item in the menu. The parameter is of type "int", the width in pixels.

PW_MENU_ITEMWIDTH_FS

Specify the width of each item in the menu. The parameter is of type "pt", which is interpretted as a factor, the base unit being the fontsize used.

PW_MENU_ITEMWIDTH_MAX

Specify that the itemwidth which should be used inside the menu should be the maximum width of all the items in the menu (this is the default). This tag requires no parameters. An attempt is made to prevent that the size of all objects in the window are redetermined when the itemwidth changes because a larger item appears. This cand influence the appearance of the window (compared with the other case) if items are added while the window is visible !

PW_MENU_ITEMWIDTH_COLUMNS

The width of the items will be such that the requested number of columns can be displayed in the menu. This tag requires one integer parameter, the number of columns.

PW_MENU_ACTION_SELECT

Specify a routine which should be called each time an item is selected. The parameter should be of type "Error (*)(PWObject, char *)", with the "char *" being the item which is being selected. This routine is not called when a `DOSELECT` routine is set and the item was indicated using a

DO.

PW_MENU_ACTION_DOSELECT

Specify a routine which should be called each time an item is selected using a *DO*. The parameter should be of type "Error (*)(PWObject, char *)", with the "char *" being the item which is being selected. If this is not defined, and the item is selected, then the normal select routine will be called. Please note that the PW_MENU_WINDOW_DO tag has no effect if a DOSELECT routine is set.

PW_MENU_ACTION_DESELECT

Specify a routine which should be called each time an item is deselected. The parameter should be of type "Error (*) (PWObject, char *)", with the "char *" being the item which is being deselected.

PW_MENU_UNIQUE

Tell the menu that at most one item can be selected at any instant. This tag needs no parameters. When this tag is given, all selected items in the menu will be deselected.

PW_MENU_SORT_COMPARE

This tag allows you to pass a compare routine to a menu, for example, STRCompareCI or STRCompareCD. It requires one parameter of type "Error *rout(char *str, char *with, int *result)". If the parameter is NULL then the menu is not sorted.

PW_MENU_VISIBLE_LINES

Specify the minimum number of lines which should be visible in the menu. Each line could (if the menu is wide enough) contain more than one item. The parameter is of type "int".

PW_MENU_CLEAR

Clear the menu. All the items will be removed from the menu. The current itemwidth is reduced to zero if the maximum width is the current option. This tags does not need a parameter. The current item, as last returned with one of the queries, is reset when adding something to the menu.

PW_MENU_WINDOW_DO

The parameter is either TRUE or FALSE. By default it is FALSE. When the window do status is TRUE, then the <enter> keypress is also handled by the system (thus a keypress object can react to it). However, if there is a special DOSELECT routine for the menu, then this tag has no effect !

PW_MENU_NONE_SELECTED

This tag is similar to the PW_MENU_UNIQUE tag, except that in this case, a menu item will never be displayed as selected. However, when a menu item is indicated, the menu Select routine will still be called. This tag does not need a parameter.

PW_MENU_STATUS

Change the status of an item in the menu. This tag needs two parameters, the new status and the item. The new status should be one of PW_STATUS_AVAILABLE, PW_STATUS_UNAVAILABLE or PW_STATUS_SELECTED. The item is passed as a "char *", the position of the item. Please notice that the pointers are compared, and therefore you whould be ware when doing this for copied items. When changing the status of an item, the Select or DeSelect routines are not called, except when it causes an item to be deselected (because PW_MENU_UNIQUE). When PW_MENU_NONE_SELECTED, this tag can only be used to make an item available or unavailable.

PW_MENU_STATUS_ALL

Set the status of all the item in the window to the status which is given as parameter. This will do nothing when you can't select more than one item at a time.

PW_MENU_STATUS_CURRENT

Change the status of the current item, being the last one which was returned by one of the queries. This tag requires only one parameter, the new status, either PW_STATUS_AVAILABLE, PW_STATUS_UNAVAILABLE or PW_STATUS_SELECTED. When changing the status of an item, the Select or DeSelect routines are not called, except when it causes another item to be deselected (because PW_MENU_UNIQUE). When PW_MENU_NONE_SELECTED, this tag can only be used to make an item available or unavailable.

---

query

PW_OUTLINE_BOX_LEFT

To allow the user to modify the behaviour and look of the outline object, there is always an empty box at the left in the outline. The user may put some objects in this box. The object identifier is returned by this query, so the value filled in is of type "PWObject".

PW_OUTLINE_BOX_RIGHT

To allow the user to modify the behaviour and look of the outline object, there is always an empty box at the right in the outline. The user may put some objects in this box. The object identifier is returned by this query, so the value filled in is of type "PWObject".

PW_OUTLINE_OBJECT_QUIT

Query the object identifier for the quit object in the outline. The value which is filled in is of type "PWObject". If there is no quit object, then the value filled in will be `NULL`.

PW_OUTLINE_OBJECT_INFO

Query the object identifier for the info object in the outline. The value which is filled in is of type "PWObject". If there is no info object, then the value filled in will be `NULL`.

PW_OUTLINE_OBJECT_DO

Query the object identifier for the do object in the outline. The value which is filled in is of type "PWObject". If there is no do object, then the value filled in will be `NULL`.

PW_OUTLINE_OBJECT_WAKE

Query the object identifier for the wake object in the outline. The value which is filled in is of type "PWObject". If there is no wake object, then the value filled in will be `NULL`.

PW_OUTLINE_OBJECT_HELP

Query the object identifier for the help object in the outline. The value which is filled in is of type "PWObject". If there is no help object, then the value filled in will be `NULL`.

change

PW_OUTLINE_SLEEP

Make sure that the outline contains a sleep item. This tag needs no parameters. The sleeping program will display its name. You should

only use this tag in the primary window of your application.

PW_OUTLINE_SLEEP_TEXT
: Make sure that the outline contains a sleep item. This tag needs one parameter of type "char *", the text which is displayed by the sleeping application. You should only use this tag in the primary window of your application.

PW_OUTLINE_QUIT
: Make sure the outline has a quit item. By default, the action of the quit item depends on the quit confirm status. If it is FALSE, the window will be exited, otherwise, a window will pop up to query whether the user is really sure he/she wants to quit the window. This tag has no parameter.

PW_OUTLINE_QUIT_ACTION
: Attach a user defined action to the quit item. The outline should already have a quit item. The parameter is of type "Error (*)(PWObject)".

PW_OUTLINE_QUIT_CONFIRM
: Set the quit confirm status. The parameter is either TRUE or FALSE. The default quit action uses this status to determine whether a confirmation request should be asked for.

PW_OUTLINE_QUIT_KEYPRESS
: Attach a keypress to the quit item. The outline should already have a quit item. The paremeter should be of type "char". By default the quit item has no keypress attached to it.

PW_OUTLINE_INFO_TEXT
: Make sure an info item is included in the outline. The parameter is of type "char *" and is the text which will be displayed in the item.

PW_OUTLINE_ACTION_INFO
: Make sure an info item is included in the outline. If no info item existed, the text in it will be "info". The parameter is of type "Error (*)(PWObject)", and is the action routine for the info item.

PW_OUTLINE_ACTION_DO
: Make sure the outline contains a do item, which can be activated also by a do keypress. The parameter is of type "Error (*)(PWObject)", and is the action routine for the do item.

PW_OUTLINE_ACTION_WAKE
: Make sure the outline contains a wake item, which can be activated also by a keypress. The parameter is of type "Error (*)(PWObject)", and is the action routine for the do item.

PW_OUTLINE_TITLE_TEXT
: Set the title for the outline. The parameter is of type "char *". By default the title will be the program name.

PW_OUTLINE_HELP
: Make sure a help item is included in the window. The default action for the help item is to execute the ProWesS reader (which should be loaded as resident extension - to make it into an executable thing). The file which has to be displayed (and the directory where it can be found and the position in the file can be specified by the PW_OUTLINE_HELP_xxx tags). This tag requires no parameters.

PW_OUTLINE_ACTION_HELP
: Assign your own action routine to the help item in the outline. If there was no help item yet, then it will be created. The parameter is of type "Error (*)(PWObject)".

PW_OUTLINE_HELP_FILE
: Specify which help file should be loaded when the help item is indicated. This will automatically reset the position in the file (so the file will be displayed from the start). The parameter is of type "char *".

PW_OUTLINE_HELP_POSITION
: Specify the position in the current help file which should be displayed when the help item is indicated by the user. The parameter is of type "char *".

PW_OUTLINE_HELP_DIRECTORY
: Specify the directory where the help file should be searched. The parameter is of type "char *".

---

create

PW_SCROLL_NOBAR

When this tag is encountered on creation of the scroll object, then only the arrows will exist, and no scrollbar. In the case, the arrow items may be bigger.

change

PW_SCROLL_CALCSIZE

Set the routine which can be used to calculate the size of the visible area (the bar), to draw the scrollbar. The parameter should be of type "Error (*)(PWObject, pt *)", and should convert the second parameter (which is the size of the canvas), into the size in the metric and type as used as paramter of PW_SCROLL_MINIMUM and PW_SCROLL_MAXIMUM.

PW_SCROLL_CANVAS

Set the canvas to which this scroll object is linked. The parameter should be of type "PWObject" and should be a canvas object.

PW_SCROLL_MINIMUM

Set the minimum value for the scrolling. The parameter is int sized and can be in any chosen metric. The scrollbar will not allow you to scroll further back than this minimum. The metric used should match the metric used as origin of the canvas. The default minimum is 0.

PW_SCROLL_MAXIMUM

Set the maximum value for the scrolling. The parameter is int sized and can be in any chosen metric. The scrollbar will not allow you to scroll further than this minimum. The metric used should match the metric used as origin of the canvas. The default maximum is 0.

PW_SCROLL_MINDIST

Set the distance to scroll when the scroll arrow is activated with a PW_EVENT_HIT. If the maxdist is not set, it will default to this value as well. If the distance is negative, the scrolling distance will be (size+mindist).

PW_SCROLL_MAXDIST

Set the distance to scroll when the scroll arrow is activated with a PW_EVENT_DO. If the mindist is not set, it will default to this value as well. If the distance is negative, the scrolling distance will be

(size+maxdist).

PW_SCROLL_SCROLL

Force a scroll without event on the scroll arrows. The scrolling distance is passed as parameter. The direction is right/down for positive, left/up for negative distance. Particularly useful to normalise the scrollbar after a window scale operation (in this case distance=0).

---

create

PW_POSITION_RIGHT_OF
: One parameter, the object which should be to the left of the newly created object. This tag is only valid for region objects.

PW_POSITION_LEFT_OF
: One parameter, the object which should be to the right of the newly created object. This tag is only valid for region objects.

PW_POSITION_ABOVE
: One parameter, the object which should be below the newly created object. This tag is only valid for region objects.

PW_POSITION_BELOW
: One parameter, the object which should be above the newly created object. This tag is only valid for region objects.

PW_POSITION_NEXT_ROW
: The newly created object will be the first object in a new row, which is positioned at the bottom inside the parent object.

PW_POSITION_NEXT_COLUMN
: The newly created object will be the first object in a new column, which is positioned at the right inside the parent object.

PW_SLEEP_OBJECT
: The newly created object is the object which should be displayed when the window is put to sleep. This tag is only valid for region objects.

query

PW_OBJECT_AUXILIARY
: Each object can have an auxiliary (int sized) value, which is returned by this query. The auxiliary is normally used to store data which is needed for the specific case of some of the action routines. These auxiliary variables are important to write code without global variables, so that the code is re-entrant. When the auxiliary value for the queried object is NULL, then the auxiliary value of the owner is returned (recursively until either no more owner or a non NULL value is encountered).

PW_GLOBAL_AUXILIARY
: The system can also have an auxiliary (int sized) value, which is returned by this query. The auxiliary is normally used to store (a pointer to) the global data structure. This auxiliary variable is important to write

code without global variables, so that the code is re-entrant.

PW_SYSTEM_GSTATE
: Sometimes, you may need a Gstate to query PROforma about something. As a ProWesS system always own a gstate, there is no need to allocate a new one for that. Therefore, this tag allows you to know the gstate which is currently used by the system. The value filled in is of type "Gstate". Please note that ProWesS may replace it's gstate by another one at any time. Therefore, it can not be guaranteed that the Gstate still exists after the next time that control is given to ProWesS.

PW_DEFAULT_FONT
: Query the ProWesS default font. The value filled in is a string with a maximum length of `PF_MAXFONTNAME`.

PW_DEFAULT_FONTSIZE
: Get the ProWesS default fontsize. This can for example be used to know a good default size to display some text in a canvas. The value filled in is of type "pt".

PW_DEFAULT_FOREGROUND
: Get the ProWesS default foreground colour. The value is of type "ColourRGB".

PW_DEFAULT_BACKGROUND
: Get the ProWesS default background colour. The value is of type "ColourRGB".

PW_DEFAULT_MIDDLEGROUND
: Get the ProWesS default middleground colour. The value is of type "ColourRGB".

change

PW_OBJECT_AUXILIARY
: Each object can have an auxiliary (int sized) value, which is set to the parameter. The auxiliary is normally used to store data which is needed for the specific case of some of the action routines. These auxiliary variables are important to write code without global variables, so that the code is re-entrant.

PW_GLOBAL_AUXILIARY
: The system can also have an auxiliary (int sized) value, which is set to the parameter. The auxiliary is normally used to store (a pointer to) the global data structure. This auxiliary variable is important to write code

without global variables, so that the code is re-entrant.

PW_SYSTEM_BREAKDOWN

This command tells ProWesS to stop the activation of the window of which the object is part. When the control is next returned to ProWesS, it will remove the window from the screen, and the program can continue after the PWActivate call.

PW_POINTER

Set the pointer which should be used in the window. The pointer is normally restored when the current region is exited from. The parameter is of type "Sprite *".

PW_WINDOWFIT

Set whether the object should be windowfitted or not, the parameter should be either TRUE or FALSE. Windowfitting is the action of making the region fit inside the parent in the secondary direction (e.g. if the parent is a row, then the secondary direction is the height). This tag is only valid for region objects.

PW_SCALE_FACTOR

Set the scale factor for the object, the parameter should be a positive integer. The scale factor is used to make the objects fit inside the parent in the primary direction. The remaining unused space is divided among the objects according to the fraction (scale factor for current object / total of scale factor for all children of parent). This tag is only valid for region objects.

PW_WINDOW_SIZE_PIX

Set the preferred size of the window. This has two parameters, the x and y size, both in pixels.

PW_WINDOW_XSIZE_PIX

Set the preferred width of the window. The parameter should be an integer, in pixels.

PW_WINDOW_YSIZE_PIX

Set the preferred height of the window. The parameter should be an integer, in pixels.

PW_WINDOW_SIZE

Set the preferred size of the window. This has two parameters, the x and y size, both in PROforma points.

PW_WINDOW_XSIZE

Set the preferred width of the window. The parameter should be in

PROforma points, with a value between 0 and 720.

PW_WINDOW_YSIZE

Set the preferred height of the window. The parameter should be in PROforma points, with a value between 0 and 540.

PW_WINDOW_ORIGIN_PIX

Set the prefered position of the window on the screen. This has two parameters, the x and y position, both in pixels.

PW_WINDOW_XORIGIN_PIX

Set the prefered horizontal position of the window on the screen. The parameter should be an integer, in pixels.

PW_WINDOW_YORIGIN_PIX

Set the prefered vertical position of the window on the screen. The parameter should be an integer, in pixels.

PW_WINDOW_ORIGIN

Set the prefered position of the window on the screen. This has two parameters, the x and y position, both in PROforma points.

PW_WINDOW_ORIGIN_RESET

Reset the window position. This will make sure that the next time the window is popped up, the position will be based on the pointer position at that moment. Otherwise the window will pop up in the same position as before. This tag has no parameters.

PW_WINDOW_XORIGIN

Set the prefered horizontal position of the window on the screen. The parameter should be in PROforma points, with a value between 0 and 720.

PW_WINDOW_YORIGIN

Set the prefered vertical position of the window on the screen. The parameter should be in PROforma points, with a value between 0 and 540.

PW_KEYPRESS

This command is only valid for keypress objects. The parameter will be the keypress to which the object reacts. The object will also react to the secondary keypress (the different case keypress) if no other keypress object exist which reacts to that keypress as primary.

PW_SYSTEM_SLEEP

Tell ProWesS that the window should be put asleep. The SLEEP_OBJECT will then be activated as current window, and the

window will be put inside the button frame (if that exists). See also PW_SLEEP_OBJECT, PW_WINDOW_BUTTON.

PW_WINDOW_BUTTON

When this tag occurs, the window will automatically position itself inside the button frame (if this is possible). If there is not enough room inside the button frame, then the system will revert back to normal behaviour. Please note that a window which is positioned inside the button frame cannot be moved or scaled. This tag does not require a parameter.

PW_WINDOW_NOSCALE

The passing of this tag, makes sure that the size of the window cannot exceed the minimum size. Scaling thus becomes impossible. This tag does not need a parameter. When no scaling is possible, a DO on the scaleborder will be interpretted as a request to move the window.

PW_WINDOW_NOXSCALE

The passing of this tag, makes sure that the horizontal size of the window cannot exceed the minimum. Horizontal scaling thus becomes impossible. This tag does not need a parameter. When no scaling is possible in either direction, a DO on the scaleborder will be interpretted as a request to move the window.

PW_WINDOW_NOYSCALE

The passing of this tag, makes sure that the vertical size of the window cannot exceed the minimum. Vertical scaling thus becomes impossible. This tag does not need a parameter. When no scaling is possible in either direction, a DO on the scaleborder will be interpretted as a request to move the window.

PW_CURSOR_SEPARATE

This tag needs one parameter, TRUE of FALSE, which determines whether the cursor keys (and space and enter) are treated as separate, or as ordinary mouse moves, hit and do.

PW_CATCH_OBJECT

The ProWesS systems passes keypresses which are not handled by a keypress object to one of the objects in the system. This object can be set with this tag. The parameter is of type "PWObject".

PW_SYSTEM_ACTION_INIT

This tag allows you to pass a routine which has to be executed by ProWesS after the window is initialy drawn, but before the pointer is

displayed (thus before any events are caught). Parameter is of type "Error (*)(PWObject)". The PWObject which is passed as the object which "receives" the event can be any object in the system (an access point to the window is needed).

PW_SYSTEM_ACTION_EVENT1
PW_SYSTEM_ACTION_EVENT2
PW_SYSTEM_ACTION_EVENT3
PW_SYSTEM_ACTION_EVENT4
PW_SYSTEM_ACTION_EVENT5
PW_SYSTEM_ACTION_EVENT6
PW_SYSTEM_ACTION_EVENT7
PW_SYSTEM_ACTION_EVENT8

Pass an action which has to be executed when the given software event is enerated for te job. Parameter is of type "Error (*)(PWObject)". The PWObject which is passed as the object which "receives" the event can be any object in the system (an access point to the window is needed). These events can only be generated and trapped on SMSQ/E systems (v2.71 or later).

PW_OBJECT_POINTER_START

This tag indicates that the pointer has to be centered inside this item when the system is activated. When there is no object in the system which was marked using this tag, then the pointer will be centered in the window. When there are several objects which were marked, the pointer can be centered in any one. This tag has no parameters.

PW_OBJECT_AUTOSIZE

This tag indicates that the object on which it is applied can try to increase its size to make more information visible. Only one object in the system is allowed to increase its size this way. When there are several objects which were marked, it is not decided which one can increase. This tag has no parameters.

---

change
PW_TITLE_TEXT
: Set the text of the title item. The text should not contain '\n'. The parameter is of type "char *".
