

FORM REFERENCE

Version 2.20

(C) Copyright Psion PLC 1990-97

All rights reserved. This manual and the programs referred to herein are copyrighted works of Psion PLC, London, England. Reproduction in whole or in part, including utilization in machines capable of reproduction or retrieval, without express written permission of Psion PLC, is prohibited. Reverse engineering is also prohibited.

The information in this document is subject to change without notice.

Psion and the Psion logo are registered trademarks, and Psion, Psion MC, Psion HC, Psion Series 3, Psion Series 3a, Psion Series 3c, Psion Siena and Psion *Workabout* are trademarks of Psion PLC.

TopSpeed is a registered trademark of Clarion Software Corporation. Intel 8086 and 80286 are registered trademarks of Intel Corporation. IBM, IBM XT and IBM AT are registered trademarks of International Business Machines Corp. Microsoft and MS-DOS are registered trademarks of Microsoft Corporation. Apple and Macintosh are registered trademarks of Apple Computer Inc. VAX and VMS are registered trademarks of Digital Equipment Corporation. Brief is a registered trademark of Underware Inc. Psion PLC acknowledges that some other names referred to are registered trademarks.

CONTENTS

1 Introduction.....	1-1
Using FORM classes	1-1
Measurement units.....	1-2
Notation.....	1-2
Names.....	1-2
Method function prototypes.....	1-2
The 'no leave' symbol	1-3
Long parameters.....	1-3
Class diagrams.....	1-3
Class hierarchy.....	1-3
Structured Error Recovery.....	1-4
Use of the p_leave mechanism.....	1-4
Panic numbers.....	1-4
Call-back methods.....	1-5
Mixin classes	1-5

2 The Formatted Document Content Classes	2-1
The FORMDOC mixin class	2-2
Class diagram	2-2
Class definition	2-2
Property	2-2
FORMDOC methods.....	2-3
Scan to start of paragraph.....	2-3
Provide characters	2-3
Sense paragraph layout data	2-4
Sense paragraph label.....	2-4
Return a page number	2-5
EPDOC.....	2-5
Class diagram	2-6
Pagination	2-6
Document filter	2-6
Class definition	2-7
Property	2-8
EPDOC methods.....	2-8
Initialise.....	2-8
Sense document length.....	2-8
Sense characters forwards.....	2-8
Sense characters backwards.....	2-9
Insert characters	2-10
Copy characters.....	2-10
Delete characters.....	2-11
Clear the document	2-11
Compress allocated storage.....	2-11
Set (or clear) the page list.....	2-11
Return a page number	2-12
Get position at the start of a page.....	2-12
Set (or clear) the filter list.....	2-12
Get unfiltered position.....	2-12

EPDOC call-back methods.....	2-13
Scan to start of paragraph.....	2-13
Provide characters	2-13
EPFDOC.....	2-14
Class diagram	2-15
Class definition	2-15
Property	2-15
EPFDOC call-back methods.....	2-15
Scan to start of paragraph.....	2-15
Provide characters	2-15

3 The Document Layout Classes..... 3-1

Precursors	3-1
Class diagram	3-1
SCRLAY	3-2
Class definition	3-2
Property	3-6
Data structures	3-7
Special characters.....	3-7
Font-width tables.....	3-8
SCRLAY methods.....	3-8
Initialise.....	3-8
Set global layout style.....	3-9
Sense global layout style.....	3-10
Destroy.....	3-10
Convert position to line number and pixel offset.....	3-10
Convert line number and pixel offset to position.....	3-11
Get horizontal pixel offsets of start & end of line.....	3-11
Prepare to read line data	3-11
Read line data	3-12
Format the next line	3-12
Scroll the layout	3-12
View position on given line	3-12
Discard layout	3-13
Adjust screen/printer scaling	3-13
Set number of lines.....	3-13
Discard lines from position.....	3-13
WRAP	3-14
Class definition	3-14
Property	3-14
WRAP methods.....	3-14
Initialise.....	3-14
Queue a line format request	3-15
Format a line.....	3-15
SCRIMG.....	3-15
Class definition	3-16
Property	3-17
SCRIMG methods.....	3-19
Destroy.....	3-19
Initialise.....	3-20
Set view and layout	3-20
Sense window information	3-22
Set emphasis on or off	3-22
Get select region.....	3-22
Scroll the image horizontally.....	3-22
Scroll the image vertically.....	3-23
Show position on given line.....	3-23
Set the cursor position.....	3-23
Draw to given rectangle.....	3-25
Discard screen layout, view and redraw	3-25
Discard layout and redraw.....	3-25
Prepare for a left delete.....	3-26

Draw paragraph to echo content change	3-26
Redraw to echo a style change	3-26
Redraw for a change after the cursor position	3-27

4 The Document Printing Classes..... 4-1

Precursors	4-1
Class diagram	4-2
Measurement units	4-2
PRINTER.....	4-2
Class definition	4-3
Property	4-5
environment variables	4-5
PRINTER methods.....	4-6
Destroy the print manager	4-6
Initialise printer & set default values	4-6
Set serial port characteristics	4-6
Set print file specification	4-6
Set type of printer port.....	4-7
Set printer model no. & WDR file name	4-7
Sense printer port information	4-7
Sense current printer port device information	4-8
Sense printer model number & WDR file name	4-9
Fetch address of printer parameters	4-9
Set top or bottom header text	4-9
Get address of top or bottom header text	4-10
Create WDR object.....	4-10
Destroy WDR object.....	4-10
Open printer port device.....	4-10
Print data source.....	4-11
Paginate data source.....	4-11
Initialise preview.....	4-12
Perform preview	4-13
Terminate preview	4-13
Return handle of preview data	4-14
PAGES	4-14
Class definition	4-15
Property	4-17
Page dimensions.....	4-20
PAGES methods	4-21
Initialise and queue	4-21
Fetch next print element.....	4-23
Handle error	4-24
Translate the print element and print data	4-24
WDR.....	4-25
Class definition	4-26
Property	4-28
WDR methods.....	4-29
Destroy.....	4-29
Initialise WDR	4-29
Return the number of models.....	4-30
Get model name from model number.....	4-30
Set the current model	4-30
Sense current model data.....	4-30
Get typeface by index	4-30
Get typeface by typeface number.....	4-31
Get font height by typeface & height indexes.....	4-31
Get font index given height	4-32
Get a requested font width table.....	4-32
Get printed width of text.....	4-33
Convert twips to printer units	4-33
Create a PDR for printer output	4-33
Load resource record	4-34

PDR.....	4-34
Class definition	4-35
Property	4-36
PDR methods	4-37
Destroy.....	4-37
Initialise PDR.....	4-37
Translate print command	4-38
Add command to buffer.....	4-41
Destroy method, subclassable by DYL	4-42
Start printing.....	4-42
Finish printing	4-42
Start a new page.....	4-43
Print text at current position	4-43
Start a new line	4-43
Position to the right.....	4-44
Set the font.....	4-44
Set the font style.....	4-45
The PAGELAY mixin class	4-46
Class diagram	4-46
Class definition	4-46
Property	4-46
PAGELAY call-back methods.....	4-46
Get next WDR_PRINT element.....	4-46
Handle status messages	4-47
PRNLAY	4-49
Class definition	4-49
Property	4-50
PRNLAY methods	4-52
Read text for printing	4-52
Set start position for printing.....	4-52
Series 3a/Series 3 notes	4-53

5 The Print Preview Class 5-1

Precursors	5-1
Class diagram	5-1
PRVPDR.....	5-2
Class definition	5-2
Property	5-3
PRVPDR.....	5-5
Initialise.....	5-5
Interpret print command.....	5-6
Destroy.....	5-7
Start printing (drawing)	5-7
Start a new page.....	5-8
Set the font.....	5-8
The PRNTPRV mixin class	5-9
Class diagram	5-9
Class definition	5-9
Property	5-9
PRNTPRV call-back methods.....	5-9
Handle status messages	5-9

6 The Calendar Image Class..... 6-1

Precursors	6-1
CALIMG	6-1
Class definition	6-2
Property	6-4
CALIMG methods	6-5
Destroy the calendar.....	6-5
Initialise the calendar	6-5

Set the title.....	6-7
Emphasise the view.....	6-7
Move the cursor.....	6-8
Move the cursor by date.....	6-9
Adjust the current date	6-9
Redraw part of the view.....	6-10
Sense the current date.....	6-10
Draw the view.....	6-10
Move the cursor by position.....	6-10
Update today's date.....	6-11

7 The Polytext Classes..... 7-1

Precursors	7-1
Class diagram	7-2
PTROOT.....	7-2
Class definition	7-2
Property	7-3
PTROOT methods.....	7-4
Add phrase to buffer.....	7-4
Wrap the text	7-4
Draw a line of text.....	7-5
Set font ID.....	7-5
Set font ID and style by phrase	7-6
Set font ID and style by attribute.....	7-7
Find phrase by attribute.....	7-7
Find information about a phrase.....	7-8
PTROOT deferred methods.....	7-8
Initialise.....	7-8
Reset	7-8
Append a record.....	7-8
Store line-length table	7-9
Get address of phrase	7-9
PTFLAT	7-9
Class definition	7-10
Property	7-10
PTFLAT methods	7-10
Destroy the instance	7-10
Initialise.....	7-10
Reset	7-11
Append a phrase.....	7-11
Store line-length table	7-11
Get address of phrase	7-11
PTSEG.....	7-12
Class definition	7-12
Property	7-13
PTSEG methods.....	7-13
Initialise.....	7-13
Reset	7-13
Append a phrase.....	7-13
Store line-length table	7-13
Get address of phrase	7-14

CHAPTER 1

INTRODUCTION

This manual is a reference document for Psion's FORM library. It provides a comprehensive guide to the library and documents the classes, methods, properties, inheritance hierarchies and other information essential for understanding and using the library.

It assumes familiarity with the concepts of Object Oriented Programming.

The *Object Oriented Programming Guide* is useful pre-requisite reading as it provides the necessary background to Object Oriented Programming as implemented at Psion. It can, of course, be read in conjunction with the *FORM Reference* manual.

The FORM library is a collection of classes which provide a range of document formatting and printing services that are independent of the user interface used by an application. The object classes it contains can be used directly or can be subclassed by any application code. Many of the classes inherit methods and property from classes in the OLIB library; the *OLIB Reference* manual is, therefore, a useful pre-requisite.

Use of the FORM library allows complex applications to be built quickly and reliably.

Each chapter in this manual contains a description of a number of closely related classes. For example, the document layout chapter discusses all classes related to the laying out of text on the screen.

The description of each class follows the same format. It includes the purpose of the class, the hierarchical relationship of the class to other classes, the actual class definition, a description of the property and a complete list and discussion of the methods. References to relevant manuals are included where necessary.

The FORM library is supplied as the *form.dyl* dynamic link library in the ROM of all SIBO machines.

All classes in the FORM library are ultimately derived from the `ROOT` class which is described in the *OLIB Reference* manual. It is, therefore, a required component of all object oriented programs.¹

The content of this manual describes the version of FORM as it exists on the Series 3a and *Workabout* (it is identical on these two machines). In general, this is also applicable to the Series 3. However, where behaviour on the Series 3 differs or where certain features, methods or property are not available on the Series 3, then this will be noted at the appropriate points in the text.

Using FORM classes

An application (or DYL) that either subclasses or creates an instance of a FORM class must declare an external reference to the FORM library (and the OLIB library) in its category file. If, for example, an application's category file has the name *myprog.cat*, the content of this category file must start with the following lines:

```
IMAGE myprog

EXTERNAL olib
EXTERNAL form
```

This ensures that, amongst other things, the defined constants representing the external category numbers for the FORM and OLIB categories (in this case, `CAT_MYAPP_FORM` and `CAT_MYAPP_OLIB`, respectively) are available to application code.

¹ It is, however, permissible for a category that has no intrinsic dependence on other OLIB and FORM classes to define its own root class and thereby eliminate all dependency on OLIB and FORM. See, for example, the *Building a Dynamic Library* chapter of the *Object Oriented Programming Guide*.

In the source code of the MYPROG application, an instance of an FORM class - say, of PTSEG - would be created with `p_new` (or `f_new`) as follows:

```
p_new(CAT_MYPROG_FORM, C_PTSEG);
```

.If *myprog.cat* defines a subclass of a FORM class (say, the class SUBPTSEG) this would exist in the local category. An instance is created using the local category number CAT_MYPROG_MYPROG, as follows:

```
p_new(CAT_MYPROG_MYPROG, C_SUBPTSEG);
```

Similar considerations apply to instances created by means of `f_newsend`.

Measurement units

Measurements are generally presented to the user in inches, centimetres or points (there are 72 points per inch).

Internally, these measurements are stored either in twips or printer units. A twip is a twentieth of a point, so that there are 1440 twips per inch.

Printer units are defined to be the natural units associated with a particular printer. The size of the unit thus varies from printer to printer and is defined in the printer driver file (see *WDR Printing* in the *Additional System Information* manual). The unit may be one tenth of an inch for a printer that has a single monospaced font, whereas a typical value for a laser printer is one three hundredth of an inch (corresponding to a printer resolution of 300 dots per inch). The `wdr_twips_to_xy` method of the `WDR` class, described later in this manual, converts a measurement in twips to the equivalent in printer units for a particular printer.

Notation

Throughout this manual, all references to the Series 3 should be taken to refer to the Series 3a, unless otherwise explicitly stated.

Names

Except in class diagrams a class name is always given in upper case, for example SCRLAY.

The method name in the title line of the description of each method is the defined symbol for the method number, without its leading `o_`. In the body of the text this name, in lower case letters, is used to refer to the *method function* (or more simply, the *method*) whereas the upper case name refers to the corresponding *message*. Thus, an object's `destroy` method function is executed when the object receives a `DESTROY` message.

Method function prototypes

The description of each method contains a function prototype that specifies the nature of any return value and the parameters with which the method is called. The parameters exclude the object handle and the method number.

For example, a method for the class EPDOC with the title line:

EP_SENSE_CHARS

Sense characters forwards

and prototyped as:

```
UINT ep_sense_chars(TEXT **pbuf, UINT pos, UINT n);
```

would be invoked by, for example:

```
UINT n, pos;
TEXT *buf; /* to take pointer to buffer */

n=40;
pos=400;
n=p_send4(hand, 0_EP_SENSE_CHARS, &buf, pos, n);
```

where `hand` is the handle of an instance of the `EPDOC` class.

This corresponds to a method function declared in C source code as:

```
METHOD UINT epdoc_ep_sense_chars(PR_EPDOC *self, TEXT **pbuf, UINT pos, UINT n)
{
    ...
}
```

The ☛ symbol

The enter and leave mechanism (which uses `p_enter` and `p_leave`) is commonly used to implement structured error recovery. See the *Error Handling and Error Recovery* chapter of the *Object Oriented Programming Guide* and the *Error Handling* chapter of the *PLIB Reference* manual.

Some methods (the vast majority of destroy methods, for example) can never fail and will therefore never call `p_leave`. The title line of a number of the more significant methods of this type are marked with a leading ☛ symbol.

With the enter and leave mechanism, a call to `p_leave` should only occur within the protection of a `p_enter` harness. If `p_leave` is called outside a `p_enter` harness, the process will be panicked with panic number 47.

The `p_leave` mechanism and its use in method functions is discussed briefly in the section on *Structured Error Recovery* later in this chapter.

Long parameters

A small number of FORM class methods require a `LONG` or a `ULONG` parameter. For the reasons explained in the *Introduction* chapter of the *Object Oriented Programming Guide*, the message-sending mechanism in TopSpeed C does not support such parameters and they should be passed as two `INT` (or `UINT`) parameters, where the first is the least significant word and the second is the most significant word of the data. In such a case the actual method prototype is always followed by a conceptual form, illustrating the intent of the parameters.

Class diagrams

To illustrate the inheritance and using relationships between classes, most chapters will contain at least one class diagram.

The notation is a subset of that used by Grady Booch and described in his book *Object-oriented Analysis and Design with applications* (2nd edition) with two minor changes;

- classes which are referenced, but not described, within a chapter (i.e. classes whose full description lies in other chapters of this manual or in a different manual), are underlined,
- the diagrams do not distinguish between 'has' (aggregation) and 'using' (client/supplier) relationships.

Also note that ultimate inheritance from the root class is assumed and is not shown.

Class hierarchy

In understanding the structure of a specific class, remember that methods and property are often inherited from a superclass (or superclasses).

While a class may contain new methods and property, it may also re-define methods inherited from a superclass (or superclasses). Note that methods in a superclass can be what are known as *deferred* methods.

To help illustrate these relationships, each class description in this manual is accompanied by a diagram which shows that class and its superclass(es) in hierarchical order. This diagram is placed at the beginning of the class description.

The diagram consists of a series of adjacent columns. The rightmost column represents the class being described and will be marked by a double line border while the column to its left represents its immediate superclass (if any) marked by a single line border and so on up the hierarchy. Each column is headed by the class name followed by two boxes; the first lists that class's property and the second lists its methods.

Deferred methods are separated from the preceding methods by a blank line and are printed in italics. If a class re-defines an inherited method, the method name in the appropriate superclass is written with a line through it.

For example, the following diagram would be included in a description of class CCCC subclassed from BBBB which itself is subclassed AAAA.

AAAA	BBBB	CCCC
property_1 property_2	property_3	property_4 property_5
method_a ~~method_b~~	~~method_b~~ method_c method_d method_e	method_b method_c method_x method_y method_z method_u

In this illustration, method_a is supplied by the superclass AAAA, while method_b is replaced in class BBBB and further replaced in class CCCC. Another method, method_c, is introduced in class BBBB but replaced in CCCC, and so on. Note that method_u is a deferred method.

The ROOT class from which all classes are derived is assumed and will not be shown in the diagrams.

Methods and property inherited from a superclass will be described in the appropriate class description.

Structured Error Recovery

As mentioned earlier, the enter and leave mechanism is commonly used to implement structured error recovery.

Use of the p_leave mechanism

In general, you should assume that all methods NOT marked with the ☼ symbol (as discussed in the section on Notation) are capable of calling p_leave, even if this is not explicitly mentioned in the method description. In some cases, such as where a method calls, directly or indirectly, a deferred method (which is supplied by a subclasser) it is not possible to specify whether the method may result in p_leave being called.

In the event of an error (such as out of system memory) occurring a method may:

- call p_leave, passing the (negative) error number,
- return the error number,
- either call p_leave or return an error number, depending on the nature of the error.

Some methods call p_leave(0), which has the effect of returning from the p_enter harness (with the return value zero) without signalling an error. This is used, for example, to provide a normal exit from a deeply nested function call, without the need for a zero return value to be passed back through the chain of calls. Intermediate functions in the chain may then be declared as VOID.

Some method functions that may call p_leave are declared as VOID. One reason for this may be that the method forms part of a chain, as described in the preceding paragraph. If user code were to send such a message within a p_enter harness, the value returned from p_enter would be indeterminate if no error arose. The solution is to construct a shell function which sends the message and then returns zero, and call this shell within a p_enter harness. The call to p_enter will then return either zero (if the method calls p_leave(0) or it executes to completion) or a negative error number.

Panic numbers

See the *Error Handling and Error Recovery* chapter of the *Object Oriented Programming Guide* and the *Error Handling* chapter of the *PLIB Reference* manual for a discussion of panics and panic numbers.

XADD does not have its own unique panic numbers, but panics a client that attempts an illegal operation using PLIB, OLIB and Window Server panic numbers, defined in the respective SDK manuals.

Call-back methods

In general, an object passes a message to another object by calling the appropriate method using the relevant method number. Within the code, the method number is usually a symbolic constant generated at category translation time.

However, an alternative is to define suitable property within the calling object and set the property to contain the method number. The code can then be constructed to use the value in the property, rather than use the symbolic constant. For example, in a class ABC, a call to such a method would take the form:

```
p_send(self->abc.handle, self->abc.methnum, ...);
```

where the object's handle is assumed to have been written to `self->abc.handle` and `self->abc.methnum` has been previously set by, say:

```
self->abc.methnum = O_METHOD_NUMBER;
```

Clearly this is slightly less efficient, both in terms of memory usage and speed of execution, than the more usual:

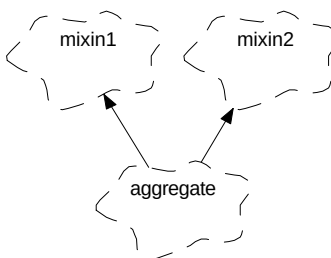
```
p_send(self->abc.handle, O_METHOD_NUMBER, ...);
```

It does, however, offer a number of advantages that, in certain circumstances, can prove to be of use:

- it allows the message being sent to be changed dynamically during the lifetime of the calling object
- the method number can be passed to the calling object, avoiding the need for the calling object to have any knowledge of the class to which the message is being sent - this can be of value in terms of design
- it allows the service to be supplied by *any* class that supports a suitable method function
- it effectively provides a multiple inheritance mechanism for the inheritance of class behaviour (but not of class property)

Mixin classes

A *mixin* class is a class that is defined for the sole purpose of being combined (or mixed in) with other classes to provide more sophisticated behaviour. Such a class encapsulates a single aspect of behaviour for the inheriting class and is not intended to be instantiated in its own right. The following figure illustrates a typical situation and shows that mixin classes are associated with multiple inheritance.



For further discussion of mixin classes see, for example, *Object Oriented Analysis and Design with Applications* by Grady Booch, published by The Benjamin/Cummings Publishing Company, Inc.

Although Psion's object oriented programming system does not support multiple inheritance, a mixin class is an ideal way of formally specifying the required functionality of call-back methods. In application code terms the mixin class itself has no physical existence, but its method functions will be implemented as part of some other 'real' class.

This manual contains formal descriptions of three mixin classes:

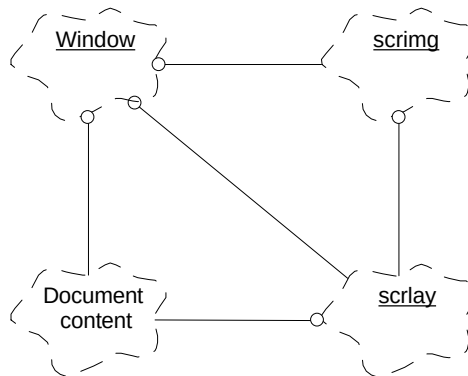
- the FORMDOC mixin class, described in the *Formatted Document Content Classes* chapter;
- the PAGELAY mixin class, described in the *Document Printing Classes* chapter;
- the PRNTPRV mixin class, described in the *Print Preview Class* chapter.

Each of the chapters referred to above contain 'real' classes which implement the respective mixin class method functions.

CHAPTER 2

THE FORMATTED DOCUMENT CONTENT CLASSES

A window that is to display formatted editable text will own a class that contains the document text. The display of the text is managed by a further two classes - an imager class (for example, `SCRIMG`) and a layout class (for example, `SCRLAY`). These two additional classes (described in the *Document Layout Classes* chapter of this manual) are both formally owned by the window, but interact with each other and the document content class to provide displayable formatted text.



During its initialisation, an instance of either the `SCRLAY` class (described in the *Document Layout Classes* chapter) or the `PRNLAY` class (used when preparing printable output, and described in the *Document Printing Classes* chapter) must be provided with the handle of an instance of a suitable document content class.

The document content class must support up to five specific services (two of which are mandatory) by means of up to five *call-back* methods, whose method numbers are also passed to `SCRLAY` or `PRNLAY`.

This chapter specifies the nature of the services that must be supported and describes the document content classes that are supplied in the FORM library.

The FORMDOC mixin class

FORMDOC
para_start sense_chars sense_plabel sense_pdata enq_page

The FORMDOC mixin class provides the *formal specification* for the call-back methods that must be supported by any class that provides access to the text content of a formatted document. These methods may be called by the SCRLAY and PRNLAY document layout classes.

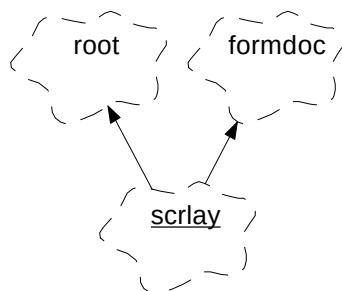
For a general discussion on call-back methods and mixin classes, see the *Introduction* chapter in this manual.

The FORMDOC class does not appear in the FORM library and an instance of FORMDOC will never be created. The FORM library supplies the two classes EPDOC and EPFDOC to encapsulate document text and provide the behaviour to manipulate and query it. These two classes, described later in this chapter, supply the minimum set of FORMDOC call-back methods required by SCRLAY and PRNLAY.

Application programmers are, however, free to supply the methods in either a separate user-defined class, or by subclassing EPDOC or (more rarely) EPFDOC; any such methods must follow the general specification prescribed by this description of the FORMDOC class.

Class diagram

The following class diagram formally illustrates the relationship between the FORMDOC mixin class and the SCRLAY layout class (see the *Document Layout Classes* chapter in this manual). Exceptionally, this diagram shows the ROOT class in order to emphasise the multiple inheritance aspect of mixin classes.



Class definition

CLASS formdoc root		
{		
DEFER para_start	scan to start of paragraph,	mandatory
DEFER sense_chars	sense character content,	mandatory
DEFER sense_plabel	sense paragraph 'label',	optional
DEFER sense_pdata	sense layout data for paragraph	optional
DEFER enq_page	sense a page number	optional
}		

Property

None.

FORMDOC methods

FORMDOC_PARA_START

Scan to start of paragraph

```
VOID formdoc_para_start(UWORD *ppos);
```

Find the position of the start of a paragraph.

The parameter `ppos` should point to a `UWORD` value which specifies a character position within the document text; the method should scan the text to find the position of the start of the paragraph containing the specified position.

The position of the start of the paragraph should be written back to `*ppos`.

A method that performs this function is mandatory; it must be supplied by the object designated to be the supplier of document text to an instance of `SCRLAY` or `PRNLAY`. Versions of this method are supplied by the `EPDOC` and `EPFDOC` classes.

FORMDOC_SENSE_CHARS

Provide characters

```
INT formdoc_sense_chars(SCRLAY_SENSECHARS *sense, SCRLAY_FONT **pf, UBYTE **pfw);
```

Sense a block of up to `WS_MAX_PRINT_BOX_TEXT_LEN` contiguous characters of data of the same style from the document text.

The parameter `sense` must point to a data structure of type `SCRLAY_SENSECHARS`. This structure, which is included as part of the `SCRLAY` class definition, is as follows:

```
typedef struct
{
    UWORD pos;           document position to sense
    WORD printer;        TRUE for printer data, FALSE for screen data
    TEXT *buf;           address of character block
    WORD blen;           length of character block
} SCRLAY_SENSECHARS;
```

The value in `sense->pos` specifies the document position where sensing is to start. The address of a buffer containing the first character should be written to `sense->buf` and the number of contiguous characters available in this buffer should be written to `sense->blen`.

The value written to `sense->blen` must be less than or equal to `WS_MAX_PRINT_BOX_TEXT_LEN`, even if a greater number of contiguous characters are actually present in the buffer. There are two reasons why fewer than this number of characters may be available:

- the characters terminate at a physical boundary within a segmented buffer, that is, at the edge of an individual buffer segment
- the characters terminate at a logical boundary where, for example, there is a change of font or of text attributes

If content-specific layout (i.e. line segments with individual font and style information) is not supported, the parameters `pf` and `pfw` may be ignored and the method should return `FALSE`. This is the case with the document classes `EPDOC` and `EPFDOC`.

If content-specific layout is supported, the parameters `pf` and `pfw` should *not* be ignored.

If `pf` is not `NULL`, the method should write into `pf` the address of a pointer to a `SCRLAY_FONT` data structure. This structure, which is included as part of the `SCRLAY` class definition, is as follows:

```
typedef struct
{
    UWORD fid;           font ID for screen or typeface no. for printer
    UWORD style;         font style (e.g. bold)
    UWORD height;        height of printer font
} SCRLAY_FONT;
```

This data structure contains information that describes the font to be applied to the sensed characters. The font descriptor should relate to either a printer font or the corresponding screen font, depending on whether `sense->printer` is `TRUE` or `FALSE`.

If `pfw` is not `NULL`, the method should write into `pfw` the address of a pointer to a font width table for the font that is to be applied to the sensed characters. The font width table should relate to either a printer font or the corresponding screen font, depending on whether `sense->printer` is `TRUE` or `FALSE`.

The method should return `TRUE`, if the characters terminate at a logical boundary, otherwise it should return `FALSE`.

A method that performs this function is mandatory; it must be supplied by the object designated to be the supplier of document text to an instance of `SCRLAY` or `PRNLAY`. Versions of this method (which do not support content-specific layout) are supplied by the `EPDOC` and `EPFDOC` classes.

FORMDOC_SENSE_PDATA **Sense paragraph layout data**

```
VOID formdoc_sense_pdata(UINT pos, INT printer, SCRLAY_PDATA *p);
```

Sense layout data for a specific paragraph.

The paragraph is that which contains the character position specified by the parameter `pos`.

The parameter `p` should point to a data structure of type `SCRLAY_PDATA`. The structure, which is included as part of the `SCRLAY` class definition, is as follows:

```
typedef struct
{
    SCRLAY_MARGINS *margins;    Paragraph margins
    SCRLAY_TABS *tabs;         Paragraph tabs
    SCRLAY_SPACING *spacing;    Paragraph spacing
} SCRLAY_PDATA;
```

If the parameter `printer` contains the value `TRUE`, the method should write the address of *three* data structures of type `SCRLAY_MARGINS`, `SCRLAY_TABS` and `SCRLAY_SPACING` into `p->margins`, `p->taps` and `p->spacing` respectively. The information in the three data structures will describe the *printer* layout for the paragraph.

If the parameter `printer` contains the value `FALSE`, the method should write the address of *two* data structures of type `SCRLAY_MARGINS` and `SCRLAY_TABS` into `p->margins` and `p->taps` respectively. The information in the two data structures will describe the *screen* layout for the paragraph. Since vertical spacing is not represented on the screen, `p->spacing` can be ignored.

A method that performs this function is optional and need not be supplied if there is no paragraph-specific layout. If, however, it is supplied, it must be by the object designated to be the supplier of document text to an instance of `SCRLAY` or `PRNLAY`.

FORMDOC_SENSE_PLABEL **Sense paragraph label**

```
VOID formdoc_sense_plabel(UINT pos, INT printer, PRNLAY_PLABEL **p);
```

Sense the label data for a specific paragraph.

The paragraph is that which contains the character position specified by the parameter `pos`.

The method should write into the parameter `p`, the *address* of a pointer to a `PRNLAY_PLABEL` data structure. This structure, which is included as part of the `PRNLAY` class definition, is as follows:

```
typedef struct
{
    SCRLAY_PLABEL s;    as for the screen
    UBYTE *wid;         the font width table
    UWORD margin;       margin for paragraph labels in printer units
    UWORD gutter;       gutter between label and para margin
} PRNLAY_PLABEL;
```

If the parameter `printer` contains the value `TRUE`, the method must specify all members of this data structure.

If the parameter `printer` contains the value `FALSE`, the method need only specify the first (i.e. the `SCRLAY_PLABEL`) member.

A method that performs this function is optional and need not be supplied if paragraph labels are not supported. If, however, it is supplied, it must be by the object designated to be the supplier of document text to an instance of `SCRLAY` or `PRNLAY`.

FORMDOC_ENQ_PAGE**Return a page number**

```
INT formdoc_enq_page(UINT pos, UINT len);
```

Return a page number.

The method should expect the following two parameters:

- pos, specifies a character position within the document text.
- len, specifies a character count, defining the number of contiguous characters starting at position pos.

If len is zero, the method should return the page number of the page that contains character position pos.

If len is *non-zero* and there is *no* page break between character position pos and the character position pos+len, then the method should return a zero.

If len is *non-zero* and there is *at least one* page break between character position pos and the character position pos+len, then the method should return the page number of the page that follows the *first* page break in the range pos to pos+len.

In any event, the method should always return zero if no page information is currently available.

It is worth noting that page numbers start from one; in other words, the first page is designated as page 1 and *not* as page 0.

A method that performs this function is optional and need not be supplied if the display of page breaks is not supported. If, however, it is supplied, it must be by the object designated to be the supplier of document text to an instance of SCRLAY or PRNLAY.

EPDOC

EPR00T	EPDOC
maxlen	b pages filter
ep_set_text ep_scan_word ep_word_count ep_scan_para ep_para_count ep_scan_block ep_add_para ep_copy_indent ep_copy_to_front ep_copy_to_back ep_paste ep_mod_chars ep_sense_text ep_capacity ~~ep_compress~~ ~~ep_init~~ ~~ep_sense_len~~ ~~ep_sense_chars~~ ~~ep_back_chars~~ ~~ep_insert~~ ~~ep_delete~~ ~~ep_extract~~ ~~ep_clear~~	ep_init ep_sense_len ep_sense_chars ep_back_chars ep_insert ep_extract ep_delete ep_clear ep_compress epdoc_para_start epdoc_sense_chars epdoc_set_pages epdoc_enq_page epdoc_goto_page epdoc_set_filter epdoc_pos_filter

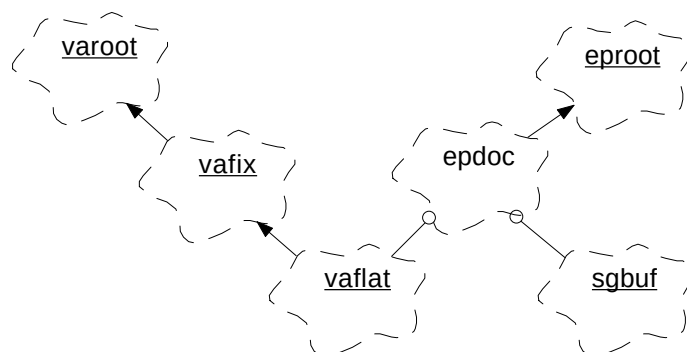
The EPDOC class encapsulates formatted document text. It provides the property for storing and the methods for manipulating the text and is referenced by instances of the SCRLAY and SCRIMG classes. Further, it supplies the two mandatory call-back methods: the 'scan to start of paragraph' method and 'the sense character content' method (also known as 'the provide characters' method).

The fundamental characteristics and behaviour of editable documents are described in the *Editable Documents* chapter of the *OLIB Reference* manual. The `EDOC` class is designed for the efficient storage and manipulation of large quantities of dynamically changing text. The text itself is contained in an `SGBUF` segmented buffer component.

For small quantities of text, the class `EPFDOC`, described later, is more efficient.

Class diagram

The following class diagram shows the relationship between the document content class `EPDOC` and other classes. The underlined classes are imported from `OLIB` and they are *all* discussed in the *OLIB Reference manual*.



Pagination

Within its property, EPDOC contains pagination information on the document.

This information is held in the form of a `UWORD` array. The array itself is a `VAFLAT` object whose handle is held in the property `epdoc.pages`.

Each entry in the array represents a single page and contains a count of the number of characters fitting into that page. It is worth pointing out that the first entry in the array represents the first page and this is deemed to be page 1 (*not* page 0). The array itself is built by an instance of the `PAGES` active object, *not* by `EPDOC` (the pagination process is *relatively* time consuming and is best done as a low priority background task).

A number of EPDOC's methods change the content of the document text, for example `ep_insert` and `ep_delete`. To avoid the overhead of re-paginating the document every time text is inserted or deleted, EPDOC modifies the character count of the appropriate page (or pages) in such a way that the position of the page break relative to the existing text remains unchanged.

Clearly, following any insertion or deletion, the calculated position of a page break may no longer be strictly accurate. This situation can only be corrected when the owning application schedules a re-pagination operation by sending a PR_PAGINATE message to the PRINTER class.

Document filter

Very often, there is a need to work with a *subset* of the whole document text. The precise meaning of a subset varies from application to application. A situation that is very common occurs in word processor applications where, often, only *outline* text needs to be displayed and manipulated. For example, outline text may consist simply of the headings in the document. In other words, text which is not part of the outline must be *logically* deleted.

To handle this kind of situation, EPDOC embraces the concept of a document filter.

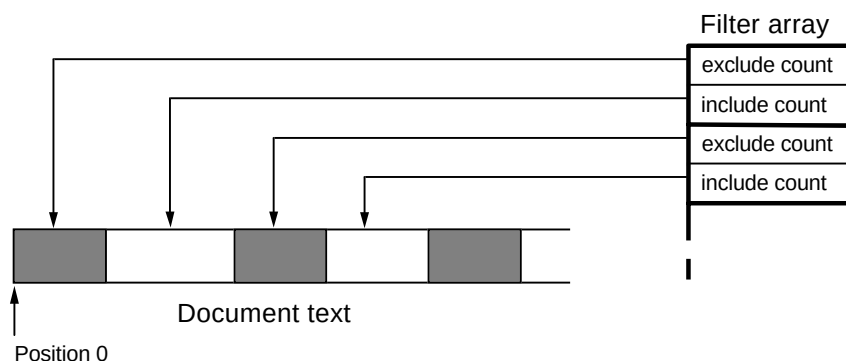
In essence, a filter is a map of the document indicating which sections of text are included in the subset and which sections are excluded (or logically deleted). When the map exists, the document is said to be *filtered*.

The document filter is implemented by means of a `UWORD` array. The array itself is a `VAFLAT` object whose handle is held in the property `epdoc.filter`.

The entries in the array are logically grouped into consecutive pairs.

Starting from the first position in the document, the first entry in the array contains a count of the number of characters which are to be *excluded* or *logically deleted* from the document; the second entry contains a count of the number of *following* characters which are to be *included* in the filtered document. This pattern is repeated for rest of the document.

The idea is more easily understood by looking at the schematic diagram below. The horizontal bar represents document text where the shaded sections represent text which is to be *excluded* or *logically deleted* from the document while the non-shaded sections represent text which is to be *included* as part of the filtered document. Position zero is on the left hand side. The vertical column represents the filter array with the individual elements marked each containing the length of the corresponding section of text.



The sum of the values contained in each element of the array should be the same as the length of the unfiltered document.

It should also be noted that both the first and the last entries in the array are often zero.

Class definition

The EPDOC class subclasses the OLIB class EPROOT and is defined in the sub-category file *epdoc.cl* (with generated header file *epdoc.g*).

```

CLASS epdoc eproot
{
  REPLACE ep_init
  REPLACE ep_sense_len
  REPLACE ep_sense_chars
  REPLACE ep_back_chars
  REPLACE ep_insert
  REPLACE ep_extract
  REPLACE ep_delete
  REPLACE ep_clear
  REPLACE ep_compress
  ADD epdoc_para_start      Scan to start of paragraph
  ADD epdoc_sense_chars     Provide characters
  ADD epdoc_set_pages       Set the page list
  ADD epdoc_enq_page        Enquire page break position
  ADD epdoc_goto_page       Get pos at start of specified page
  ADD epdoc_set_filter       Set (or clear) the filter list
  ADD epdoc_pos_filter       Convert filtered pos to unfiltered pos

  PROPERTY 3
  {
    PR_SGBUF *b;           handle of buffers data
    PR_VAFLAT *pages;      number of characters in each page
    PR_VAFLAT *filter;     filter (e.g for outline mode)
  }
}

```

Property

<code>epdoc.b</code>	The handle of an instance of SGBUF containing the document text; this is a component of EPDOC.
<code>epdoc.pages</code>	The handle of an instance of VAFLAT, containing a UWORD array. Each consecutive entry in the array contains a value giving the number of characters in consecutive pages. The first entry in the array refers to page 1. See the <i>Pagination</i> section above.
<code>epdoc.filter</code>	The handle of an instance of VAFLAT, containing a UWORD array. The array contains the document filter information as described in the <i>Document filter</i> section above.

EPDOC methods

EP_INIT	Initialise
---------	------------

```
VOID ep_init(UINT maxlen);
```

Initialise the instance of EPDOC.

The maximum length of the document (the superclass property `eproot.maxlen`) is set to the value contained in the parameter `maxlen`. Note that this length *includes* the terminating NULL.

An instance of the segmented buffer SGBUF is created and its handle stored in the property `epdoc.b`. At the same time, the instance is initialised by sending a `SB_INIT` message specifying a segment length of 128 bytes.

The buffer itself is seeded with a single NULL character. This is the delimiter which marks the end of the text and, in effect, creates an empty document.

EP_SENSE_LEN	Sense document length
--------------	-----------------------

```
UINT ep_sense_len(VOID);
```

Sense the current length of the document text.

The method returns the number of bytes of text stored; note that this *excludes* the terminating NULL.

If the document is filtered (i.e. `epdoc.filter` is not NULL), the reported length is that of the filtered document; again the length *excludes* the terminating NULL.

EP_SENSE_CHARS	Sense characters forwards
----------------	---------------------------

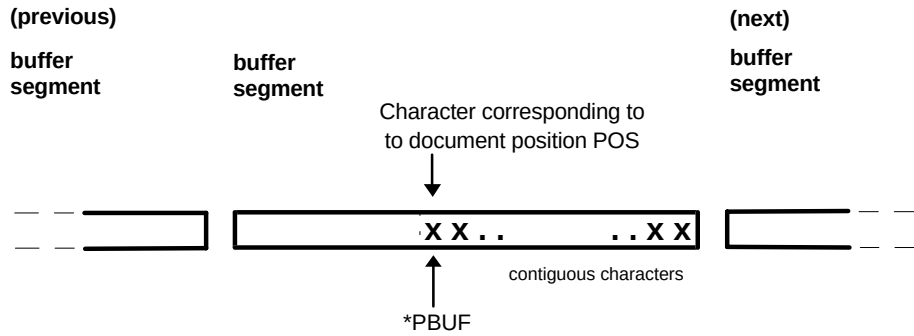
```
UINT ep_sense_chars(TEXT **pbuf, UINT pos, UINT n);
```

Find the address of the character within the segmented buffer (containing the document) whose position within the document is given by the parameter `pos`.

The method places the address of the character into an area whose *address* is passed in the parameter `pbuf`; i.e. the address of the character is set into `*pbuf`.

In addition, it returns either the value in the parameter `n` or the number of characters stored contiguously at `*pbuf`, whichever is the smaller. As the document text is contained in a segmented buffer, the number of contiguous characters at the given position will never be greater than the maximum number of characters within a data segment. The number of contiguous characters will *include* the NULL that terminates the document, if it happens to be in that particular segment.

The following figure illustrates the situation for a non-filtered document.



If the document is filtered (i.e. `epdoc.filter` is not `NULL`), the character position, address and character count are all with respect to the filtered document.

More information on segmented buffers can be found in *The SGBUF Segmented Buffer Class* chapter in the *Olib Reference* manual.

EP_BACK_CHARS

Sense characters backwards

```
UINT ep_back_chars(TEXT **pbuf, UINT pos, UINT n);
```

Find the address of a character within the segmented buffer (containing the document) which is a number of bytes *in front* of the character whose position is specified by the parameter `pos`. The method places the required address into an area whose *address* is passed in the parameter `pbuf`; i.e. the required address is set into `*pbuf`.

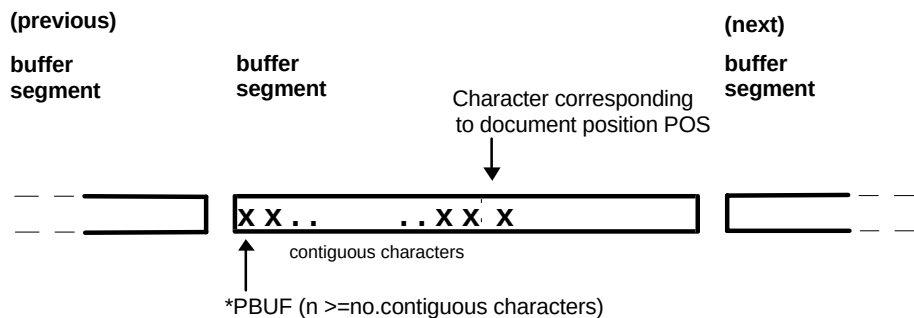
In general, the required address is calculated by taking the address of the character whose position is given by `pos` and subtracting *either* the value in the parameter `n` *or* the number of contiguous characters stored *in front* of that character, whichever is the smaller.

If the character specified by `pos` lies at the very beginning of a buffer, then the required address will lie in the *previous* buffer segment.

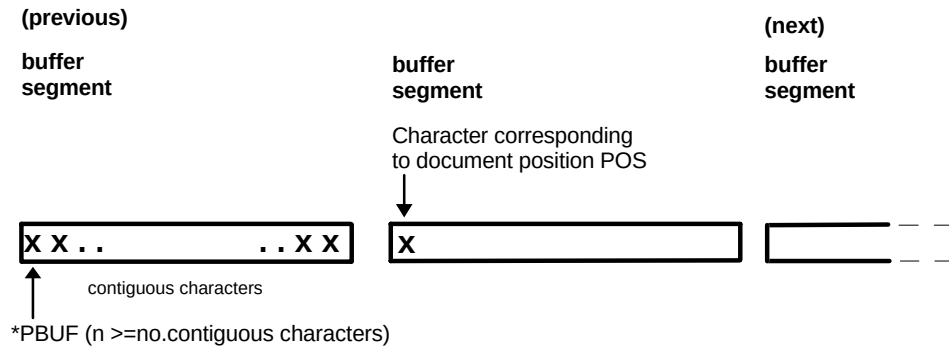
In addition, the method returns either the value in the parameter `n` or the number of contiguous characters stored in front of that character, whichever is the smaller.

As the document text is contained in a segmented buffer, the number of contiguous characters will never be greater than the maximum number of characters which can be fitted in a data segment.

The following figure illustrates the situation for a non-filtered document where the character corresponding to document position `pos` lies wholly within the buffer segment. `*pbuf` is shown pointing to the lowest possible address in the buffer segment (a situation when $n \geq \text{number of contiguous characters}$).



The following figure illustrates the situation for a non-filtered document where the character corresponding to document position `pos` lies at the *beginning* of the buffer segment. `*pbuf` is shown pointing to the lowest possible address in the *previous* buffer segment (a situation when $n \geq$ number of contiguous characters).



In both cases, if $n < \text{number of contiguous characters}$, then `*pbuf` will point to a position which is (number of contiguous characters - n) bytes higher than (to the right of) that shown.

If the document is filtered (i.e. `epdoc.filter` is not NULL), the character position, address and character count are all with respect to the filtered document.

More information on segmented buffers can be found in *The SGBUF Segmented Buffer Class* chapter in the *Olib Reference* manual.

EP_INSERT

Insert characters

```
INT ep_insert(UINT pos, VOID *buf, UINT len);
```

Insert characters into the (assumed unfiltered) document.

The source for the characters is the buffer whose address is passed in the parameter `buf`. The parameter `len` specifies the number of characters, while the parameter `pos` indicates the position within the document where the characters are to be inserted.

If an attempt to insert the characters were to cause the size of the document to exceed its maximum permitted length (i.e. the value in the property `eproot.maxlen`), then no insertion would be attempted and `p_leave` would be called with an `E_GEN_OVER` error.

If the document has been paginated, the character count for the page containing document position `pos` is incremented by `len`, so that page break positions, relative to the document text, do not move. This is achieved by incrementing the appropriate array entry in the component object `epdoc.pages`. Re-pagination may well be desirable after the insertion of text but is not done here.

If there is insufficient memory to perform the insertion `p_leave` is called with an `E_GEN_NOMEMORY` error.

The method returns zero if the insertion is successful, and is thus suitable for being called under the protection of `p_enter`.

EP_EXTRACT

Copy characters

```
VOID ep_extract(UINT pos, TEXT *buf, UINT len);
```

Copy characters from the document into a buffer.

The parameter `pos` specifies the position within the document *from* where copying is to start. The parameter `buf` points to a buffer supplied by the caller *into* which the characters are to be placed while the parameter `len` specifies *how many* characters are to be copied.

The caller is responsible for supplying a buffer of sufficient length to contain the copied text.

The document may be filtered or unfiltered. If it is filtered, the position and extracted characters are with respect to the filtered document.

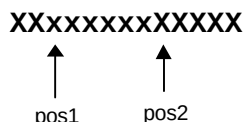
EP_DELETE**Delete characters**

```
VOID ep_delete(UINT pos1, UINT pos2);
```

Delete characters lying between two specified positions within the (assumed unfiltered) document.

The parameter `pos1` specifies the start document position while the parameter `pos2` specifies the end document position.

All characters starting at (and including) `pos1` and ending at (but excluding) `pos2`, are to be deleted. The method deletes (`pos2 - pos1`) characters beginning with the character at `pos1` by sending a `SB_DELETE` message to the `SGBUF` object containing the document text. The following figure illustrates the situation; in this example, the characters in lower case are the ones which are deleted.



Recall that the last addressable position lies immediately before the final paragraph delimiter (often referred to as the terminating `NULL`); consequently, the final paragraph delimiter *cannot* be deleted.

If the document is paginated (i.e. `epdoc->pages` is not `NULL`), the character count for the page containing position `pos1` (and, if necessary, subsequent pages) is decremented by an amount equal to `pos2-pos1`. This may result in one or more pages containing zero characters. Page break positions for the remaining characters, from `pos2` onwards, occur between the same characters as before.

Re-pagination may well be desirable after the deletion of text but is not done here.

EP_CLEAR**Clear the document**

```
VOID ep_clear(VOID);
```

Delete the entire document content.

The method deletes all of the text but leaves the terminating `NULL` which marks the end of the document by sending a `SB_DELETE` message to the `SGBUF` object containing the text.

Any existing document filter is destroyed by calling the `epdoc_set_filter` method, directly.

EP_COMPRESS**Compress allocated storage**

```
VOID ep_compress(VOID);
```

Compress the allocated storage containing the document text.

The text itself is held in the `SGBUF` (segmented buffer) component of `EPDOC` whose handle is held in the property `epdoc.b`. The segmented buffer is compressed by sending it a `SB_COMPRESS` message.

The buffer always contains at least one cell and guarantees sufficient space to contain, as a minimum, the document's terminating `NULL`.

EPDOC_SET_PAGES**Set (or clear) the page list**

```
VOID epdoc_set_pages(PR_VAFLAT *pages);
```

Clear an existing page list and/or set a new one.

The method destroys the current `epdoc.pages` component, if it exists, and resets the property `epdoc.pages` to `NULL`.

The parameter `pages` is expected to be either `NULL` or the handle of a `VAFLAT` object containing page information (as described in the section on `EPDOC` property) and is copied into `epdoc.pages`.

EPDOC_ENQ_PAGE**Return a page number**

```
INT epdoc_enq_page(UINT pos, UINT len);
```

Return a page number.

The parameter `pos` specifies a document position while the parameter `len` specifies the number of characters starting at `pos`.

If the parameter `len` is zero, the method returns the page number of the page that contains the character at position `pos`; recall that page numbers start at one.

If `len` is non-zero, the method returns:

- the page number of the page that contains the character at `pos`, if there is a page break between positions `pos` and `pos+len`.
- zero, if there is *no page break* between positions `pos` and `pos+len`.
- zero, if `pos` is zero

The method always returns zero if there is no `epdoc.pages` component (i.e. the property `epdoc.pages` is `NULL`).

EPDOC_GOTO_PAGE**Get position at the start of a page**

```
UINT epdoc_goto_page(UINT num);
```

Return the document character position corresponding to the start of a given page.

The parameter `num` specifies the page number.

Page numbers always start at one. If `num` is zero, a value of one will be assumed. If `num` is greater than the maximum number of pages, the maximum value will be assumed.

The method always returns zero if there is no `epdoc.pages` component (i.e. the property `epdoc.pages` is `NULL`).

EPDOC_SET_FILTER**Set (or clear) the filter list**

```
VOID epdoc_set_filter(PR_VAFLAT *filter);
```

Clear an existing filter list and/or set a new one.

The method destroys the current `epdoc.filter` component, if it exists, and sets the property to `NULL`.

The parameter `filter` is expected to be either `NULL` or the handle of a `VAFLAT` object containing filter information (as described in the *Document filter* section) and is copied into the property `epdoc.filter`.

Because it is no longer valid, the current `epdoc.pages` component, if it exists, is also destroyed, setting its property to `NULL`.

EPDOC_POS_FILTER**Get unfiltered position**

```
UINT epdoc_pos_filter(UINT pos);
```

Retrieve the unfiltered document position corresponding to a filtered document position.

The parameter `pos` is assumed to contain the position within the filtered document. The method converts this position into the corresponding position in the unfiltered document.

If no filter exists (i.e. the property `epdoc.filter` is `NULL`), the value in `pos` is returned.

EPDOC call-back methods

EPDOC supplies the two mandatory call-back methods required to implement the FORMDOC mixin class. It does *not* supply the other three optional call-back methods.

EPDOC_PARA_START

Scan to start of paragraph

```
VOID epdoc_para_start(UWORD *ppos);
```

Find the position of the start of a paragraph.

The parameter `ppos` points to a `UWORD` value which specifies an (unfiltered) position within the document text; the method scans backwards to find the start of the paragraph which contains this position by sending an `EP_SCAN_PARA` message (see the description of `EPROOT` in the *Olib Reference* manual). The position of the start of the paragraph is written back to `*ppos`.

If `*ppos` is already at the start of a paragraph boundary, no scanning will be done.

EPDOC_SENSE_CHARS

Provide characters

```
INT epdoc_sense_chars(SCRLAY_SENSECHARS *sense, SCRLAY_FONT **pf, UBYTE **pfw);
```

Sense a block of up `WS_MAX_PRINT_BOX_TEXT_LEN` contiguous characters from the document text.

The parameter `sense` must point to a data structure of type `SCRLAY_SENSECHARS`. The structure, which is included as part of the `SCRLAY` class definition, is as follows:

```
typedef struct
{
    UWORD pos;           document position to sense
    WORD printer;        TRUE for printer data, FALSE for screen data
    TEXT *buf;           address of character block
    WORD blen;           length of character block
} SCRLAY_SENSECHARS;
```

The value in `sense->pos` specifies the document position where sensing is to start. The address of the first character is written to `sense->buf` and the number of contiguous characters available is written to `sense->blen`. The content of `sense->printer` is not used by this method and its value is, therefore, irrelevant.

The number of contiguous characters available will be less than or equal to `WS_MAX_PRINT_BOX_TEXT_LEN` for reasons stated in the description of the `ep_sense_chars` method. The sensing is done by sending this instance of EPDOC (i.e. itself) an `EP_SENSE_CHARS` message.

The method *always* returns a value of `FALSE`.

The parameters `pf` and `pfw` are not relevant here and can be ignored. They are included in the function prototype because this method is a special case of a more general design.

In calling this method, the `SCRLAY` object passes the parameters `pf` and `pfw` which, in general, the call-back method *could* modify in order to provide font and style information for a line segment.

EPDOC, however, does not support line segments with individual font and style information and, therefore, has no need to reference the parameters `pf` and `pfw` - which is why they can be safely ignored.

For the same reason, the method always returns the value `FALSE` to indicate that there is no change of font or style in the characters sensed.

EPFDOC

EPROOT	EPFLAT	EPFDOC
maxlen	alen gran len buf	
ep_set_text ep_scan_word ep_word_count ep_scan_para ep_para_count ep_scan_block ep_add_para ep_copy_indent ep_copy_to_front ep_copy_to_back ep_paste ep_mod_chars ep_sense_text ~~ep_capacity~~ ~~ep_compress~~ ~~ep_init~~ ~~ep_sense_len~~ ~~ep_sense_chars~~ ~~ep_back_chars~~ ~~ep_insert~~ ~~ep_delete~~ ~~ep_extract~~ ~~ep_clear~~	destroy ep_init ep_sense_len ep_sense_chars ep_back_chars ep_insert ep_extract ep_delete ep_clear ep_compress ep_capacity ef_granularity ef_sense_buf	epfdoc_para_start epfdoc_sense_chars

The EPFDOC class is, in many ways, similar to EPDOC. However, it is useful and indeed more efficient for *very small* documents such as those containing the text for edit boxes in dialogs.

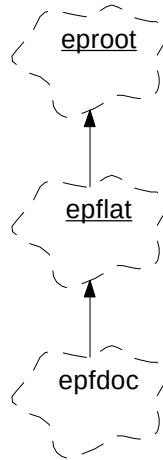
In contrast to the EPDOC class, the text is held contiguously in a single allocated cell.

The methods and property provided by its superclass(es) EPFLAT and EPROOT are sufficient for the behaviour required of EPDOC, although the class makes use of EPFDOC's epfdoc_para_start and epfdoc_sense_chars as the mandatory call-back methods (as required by instances of SCRLAY).

Because the class is designed for handling small amounts of text, no methods comparable to EPDOC's epdoc_set_pages method, for example, are needed.

Class diagram

The following class diagram shows the relationship between the document content class EPFDOC and other classes. The underlined classes are imported from OLIB and they are *all* discussed in the *OLIB Reference manual*.



Class definition

The EPFDOC class subclasses the OLIB class EPFLAT and is defined in the sub-category file *epdoc.cl* (with generated header file *epdoc.g*).

```

CLASS epfdoc epflat
{
  ADD epfdoc_para_start=epdoc_epdoc_para_start
  ADD epfdoc_sense_chars=epdoc_epdoc_sense_chars
}
  
```

Property

None.

EPFDOC call-back methods

EPFDOC supplies only the two mandatory call-back methods required to implement the FORMDOC mixin class. It does *not* supply the other three optional call-back methods.

EPFDOC_PARA_START Scan to start of paragraph

```
VOID epfdoc_para_start(UWORD *ppos);
```

This method is exactly the same as the `epdoc_para_start` call-back method discussed in the EPDOC class description.

EPFDOC_SENSE_CHARS Provide characters

```
INT epfdoc_sense_chars(SCRLAY_SENSECHARS *sense, SCRLAY_FONT **pf, UBYTE **pfw);
```

This method is exactly the same as the `epdoc_sense_chars` call-back method discussed in the EPDOC class

CHAPTER 3

THE DOCUMENT LAYOUT CLASSES

The document layout classes supply a flexible means to display formatted text. On the Series 3, the range of applications using or subclassing these classes varies from simple edit boxes to the Data, Agenda and Word applications.

An application must supply an instance of a (machine-specific) window class that supplies the user interface for text editing and in which is to appear a view of the text. This edit window will create and initialise document layout class components. The `EDWIN` edit windows class as described in the *HWIM Reference* manual is a good example and is included in the class diagram below.

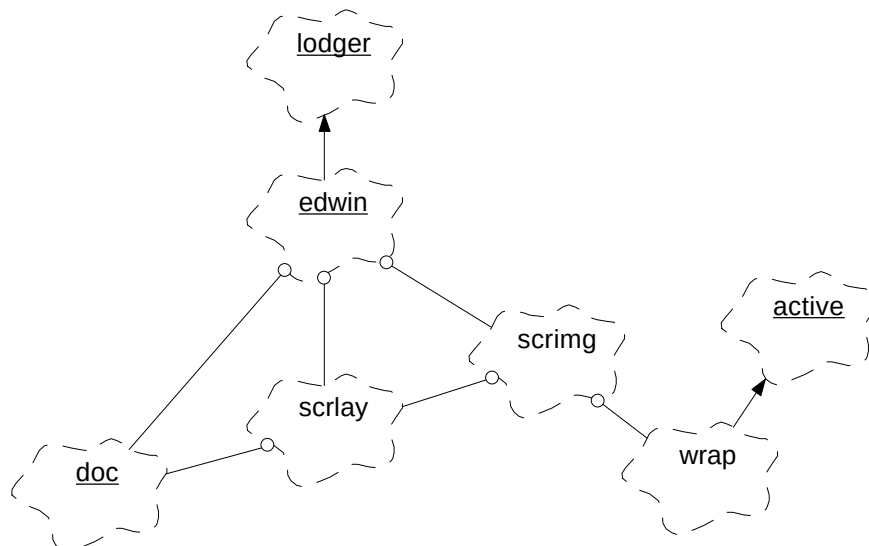
Precursors

An understanding of the document layout classes will be helped by a knowledge of:

- the `p_enter` and `p_leave` error handling services
- the OLIB editable document classes, `EPROOT` and `EPFLAT`
- the OLIB active object class, `ACTIVE`

Class diagram

The following diagram shows the relationships between the classes involved in document layout and are discussed in detail in this chapter. The `ACTIVE` class is imported from OLIB and is discussed in the *OLIB Reference* manual while `EDWIN` is the *HWIM* window class mentioned in the introduction.



SCRLAY

SCRLAY
<code>paras</code> <code>first</code> <code>nomemory</code> <code>adjust</code> <code>scan</code> <code>rd</code> <code>fmt</code> <code>doc</code> <code>slines</code> <code>spadjust</code> <code>st</code>
<code>destroy</code> <code>sl_init</code> <code>sl_set</code> <code>sl_sense</code> <code>sl_line_ends</code> <code>sl_pos_to_xl</code> <code>sl_xl_to_pos</code> <code>sl_begin_read</code> <code>sl_read</code> <code>sl_format_line</code> <code>sl_scroll</code> <code>sl_view</code> <code>sl_discard_layout</code> <code>sl_set_lines</code> <code>sl_para_changed</code> <code>sl_rescale</code>

The SCRLAY class provides services to lay out paragraphs, lines and line segments for display on the screen, from a document that is composed of a sequence of paragraphs.

In effect, SCRLAY provides property and structures which *model* the layout of a document on the screen. The methods supplied by this class allow the layout *model* to be manipulated.

An instance of SCRLAY is normally referenced by two other objects: its creator (normally a window class) and a SCRIMG screen imaging class. SCRLAY itself references a document object which contains the character data to be formatted and supplies any content-specific layout data.

SCRLAY may be used to lay out the text for display in a window in two modes:

- *screen layout*, where text is word-wrapped to the window boundary. Page break and margin positions and the effects of tabs are shown only approximately but the display is well suited to document editing.
- *printer layout*, where line breaks and page breaks occur in the exact positions that they will occur in the printed document, and the effects of margin indents and tabs are shown with greater accuracy. The positioning of text is calculated in terms of the widths of characters in the current printer's fonts but is, of course, drawn on the screen in the corresponding screen fonts. Queries to the document content class, therefore, request a mixture of screen- and printer-related data (see also the FORMDOC notional class and the idea of call-back methods).

Class definition

The SCRLAY class subclasses ROOT and is defined in the sub-category file *scrlay.cl* (with generated header *scrlay.g*)

```

CLASS scrlay root
{
    REPLACE destroy
    ADD sl_init          Set document and initial global layout style
    ADD sl_set           Set global layout style
    ADD sl_sense         Sense global layout style
    ADD sl_line_ends     Get x positions of line ends
    ADD sl_pos_to_xl     Convert document pos to line number & horizontal
                        pixel offset
    ADD sl_xl_to_pos     Convert line number & horizontal pixel offset
                        to document position
    ADD sl_begin_read    Set up for a read of line data
    ADD sl_read          Read line data
    ADD sl_format_line   Format a line from a given pos
    ADD sl_scroll        Scroll the layout by lines
    ADD sl_view          View document position on specified line
    ADD sl_discard_layout Discard layout
    ADD sl_set_lines     Set number of lines
    ADD sl_para_changed  Cause lines from pos to be discarded
    ADD sl_rescale       Adjust screen/printer scaling
}

CONSTANTS
{
    SCRLAY_SYM_HARD_HYPHEN    7    Not a word delimiter
    SCRLAY_SYM_SOFT_HYPHEN   14    Also called potential hyphen
    SCRLAY_SYM_HARD_SPACE    15    Not a word delimiter
    SCRLAY_SYM_SHOW_SPACE    8     A visible space

    SCRLAY_SHOW_TABS         0x01
    SCRLAY_SHOW_SPACES       0x02
    SCRLAY_SHOW_CR           0x04
    SCRLAY_SHOW_HYPHENS     0x08    Show optional hyphens
    SCRLAY_SHOW_LFS          0x10
    SCRLAY_WIDOW_ORPHAN     0x20    Set to enable widow & orphan suppression

    SCRLAY_ALIGN_LEFT        0
    SCRLAY_ALIGN_RIGHT       1
    SCRLAY_ALIGN_CENTRE      2
    SCRLAY_ALIGN_JUSTIFY     3

    SCRLAY_REPEAT_TAB        3
    SCRLAY_NTABS_MAX         8

    SCRLAY_SCAN_POS          0
    SCRLAY_SCAN_XY           1
    SCRLAY_SCAN_LINE         2

    SCRLAY_SPACING_KEEP_NEXT 0x01
    SCRLAY_SPACING_KEEP_TOGETHER 0x02
    SCRLAY_SPACING_NEW_PAGE  0x04

    SCRLAY_TBOX_TAB          0x8000
    SCRLAY_TBOX_TAB_USED     0x4000
    SCRLAY_TBOX_TAB_LEFT     0x2000
    SCRLAY_TBOX_NO_STYLE     0x1000
    SCRLAY_TBOX_MASK_LEN     0xff
}

TYPES
{
    typedef struct
    {
        UWORD x;          tab position
        UWORD type;       tab type (left, right, centre or repeated)
    } SCRLAY_TABSTOP;

    typedef struct
    {
        UWORD ntab;       number of tabs
        SCRLAY_TABSTOP tab[SCRLAY_NTABS_MAX];
    } SCRLAY_TABS;
}

```

```
typedef struct
{
    UWORD left;           Left margin
    UWORD right;          Right margin
    UWORD indent;         Left margin of first line in para
    UWORD align;          Alignment (left, right, centre or justified)
} SCRLAY_MARGINS;

typedef struct
{
    UWORD line;           Space between paragraph lines
    UWORD above;          Space above paragraph
    UWORD below;          Space below paragraph
    UWORD flags;          Keep together/next and start new page
} SCRLAY_SPACING;

typedef struct
{
    UWORD fid;            font ID for wserv or typeface for printer
    UWORD style;          font style (eg bold)
    UWORD height;         height of printer font
} SCRLAY_FONT;

typedef struct
{
    UWORD pos;            document position to sense
    WORD printer;         TRUE for printer data else screen data
    TEXT *buf;           address of character block
    WORD blen;           length of character block
} SCRLAY_SENSECHARS;

typedef struct
{
    SCRLAY_MARGINS *margins;    Paragraph margins
    SCRLAY_TABS *tabs;         Paragraph tabs
    SCRLAY_SPACING *spacing;    Paragraph spacing
} SCRLAY_PDATA;

typedef struct
{
    SCRLAY_FONT font;          font ID, height (printer only) and style
    UWORD align;              alignment
    TEXT *buf;                address of character block
    WORD blen;                length of character block
} SCRLAY_PLABEL;

typedef struct
{
    UWORD len;                Length of doc (one greater than max position)
    VOID *content;            Object containing document content
    WORD sensechars;          Method to sense character segments
    WORD sensepdata;          Method to sense paragraph layout data
    WORD senseplabel;         Method to sense paragraph label
    WORD toparst;             Method to scan start of paragraph
    WORD enqpage;             Method to enquire for a page break
} SCRLAY_DOC;

typedef struct que_tbox
{
    struct scrlay_tbox *next;
    struct scrlay_tbox *prev;
} QUE_TBOX;

typedef struct scrlay_tbox
{
    QUE_TBOX hd;
    WORD width;               width of box in pixels
    UWORD tlen;               number of doc positions, with mask info
} SCRLAY_TBOX;
```

```

typedef struct que_line
{
    struct scrlay_line *next;
    struct scrlay_line *prev;
} QUE_LINE;

typedef struct scrlay_line
{
    QUE_LINE hd;
    QUE_TBOX tboxes;
    WORD indent;           x pixel position of left of 1st tbox
    UWORD len;             number of addressible content positions
    UBYTE islast;          TRUE if last line in paragraph
    UBYTE new_page;        TRUE if start of page
} SCRLAY_LINE;

typedef struct que_para
{
    struct scrlay_para *next;
    struct scrlay_para *prev;
} QUE_PARA;

typedef struct scrlay_para
{
    QUE_PARA hd;
    QUE_LINE lines;
} SCRLAY_PARA;

typedef struct
{
    UWORD pos;             document position
    WORD line;             line number
    WORD x;                x offset
} SCRLAY_PLX;

typedef struct
{
    WORD scan;             Criterion for scan
    SCRLAY_PLX test;       Test position for scan
    SCRLAY_PLX pbeg;       Beginning of para from scan
    SCRLAY_PLX lbeg;       Beginning of line from scan
    SCRLAY_PLX tbeg;       Beginning of tbox from scan
    SCRLAY_TBOX *pt;       Current TBOX from scan
    SCRLAY_LINE *pl;       Current LINE from scan
    SCRLAY_PARA *pp;       Current PARA from scan
} SCRLAY_SCAN;

typedef struct
{
    SCRLAY_PLABEL *label;  To write in label margin
    SCRLAY_FONT f;         Font data for TBOX
    UWORD width;           Pixel width of TBOX
    UWORD indent;          Indent in pixels before tbox
    UBYTE blen;            Length of text written to buf ...
                           ... (See sl_read method)
    UBYTE isfirst;         TRUE if first TBOX in line
    UBYTE islast;          TRUE if last TBOX in line
    UBYTE new_page;        TRUE if page break before line
} SCRLAY_READ;

typedef struct
{
    SCRLAY_PARA *pp;       The next paragraph to read
    SCRLAY_LINE *pl;       The next line to read
    SCRLAY_TBOX *pt;       The next tbox to read
    UWORD pos;             The next document pos to read
} SCRLAY_RD;

```

```
typedef struct
{
    SCRLAY_PARA *pp;      The next paragraph to format
    UWORD pos;            The next document pos to format
    WORD line;            The next line to format
    UWORD pend;           Pos at beginning of existing layout
                          when background formatting
    WORD lend;            Line number of beginning of existing layout
} SCRLAY_FMT;

typedef struct
{
    UBYTE options;        Layout options
    UBYTE printer;        TRUE if printer layout
    SCRLAY_PDATA pd;      Global margins, tabs, spacing
    SCRLAY_FONT *font;     Global font ID and style
    UBYTE *fwtab;         Global font table or NULL
    UWORD scrpwidth;       Width of screen (240 pixls) in printer units
    SCRLAY_FONT *sfont;    Global screen font ID and style
} SCRLAY_STYLE;
}

PROPERTY
{
    QUE_PARA paras;       List of paragraphs
    SCRLAY_PLX first;     Screen/document position of 1st SCRLAY_TBOX
    UBYTE nomemory;       TRUE if failed to allocate memory
    UBYTE adjust;         TRUE if spadjust has been changed
    SCRLAY_SCAN *scan;    Scan context
    SCRLAY_RD rd;         Read context
    SCRLAY_FMT fmt;       Format line context
    SCRLAY_DOC doc;       Describes document object
    WORD slines;          Number of lines in screen or height of page
    WORD spadjust;        Adjustment to scrpwidth
    SCRLAY_STYLE st;      Global layout style
}
}
```

Property

<code>scr lay .paras</code>	The head of a list (a doubly-linked queue) of items representing the layout of the data. Each data item in the list represents a paragraph; from each of these is queued a list of data items representing lines within the paragraph; from each of these is queued a list of data items representing individual phrases within the line (i.e. sections of text with a different font or with different attributes from its neighbours). See the figure in the section titled <i>Data structures</i> for more detail.
<code>scr lay .first</code>	The position of the first text box. This is described in terms of both a document position and a screen position (the line number and the horizontal pixel offset from the beginning of the line). It is important to note that the line number can be negative; in a situation where only part of a paragraph is visible at the top of the screen, layout will have been generated for the whole paragraph. Consequently, those lines which are not visible (i.e. above the screen) will have negative line numbers.
<code>scr lay .nomemory</code>	Used to indicate an out of memory error. Set to <code>TRUE</code> if an out of memory condition occurs.
<code>scr lay .adjust</code>	Used to indicate whether the property <code>scr lay .spadjust</code> has been changed. Set to <code>TRUE</code> if it has been changed.
<code>scr lay .scan</code>	Used to keep track of the current scan position and the general context when scanning.
<code>scr lay .rd</code>	This is a <code>SCRLAY_RD</code> data structure wick is used to record the current read context when reading through the layout.

<code>scrlay.fmt</code>	Used to keep track of format data when generating layout.
<code>scrlay.doc</code>	Document content information. This data structure holds the call-back methods and the handle of the document object; this property is set on initialisation by the <code>SL_INIT</code> method.
<code>scrlay.slines</code>	The number of lines in screen or height of page.
<code>scrlay.spadjust</code>	Adjustment to the <code>scrwidth</code> member of the global layout style data in the <code>scrlay.structure</code> property. The <code>scrwidth</code> member contains the width of the screen in printer units.
<code>scrlay.st</code>	A data structure that contains the global font ID, style information etc, set during initialisation. In general, this information applies to the whole document unless "overridden" by content specific information. This data may subsequently be set using the <code>sl_set</code> method and sensed using the <code>sl_sense</code> method.

Data structures

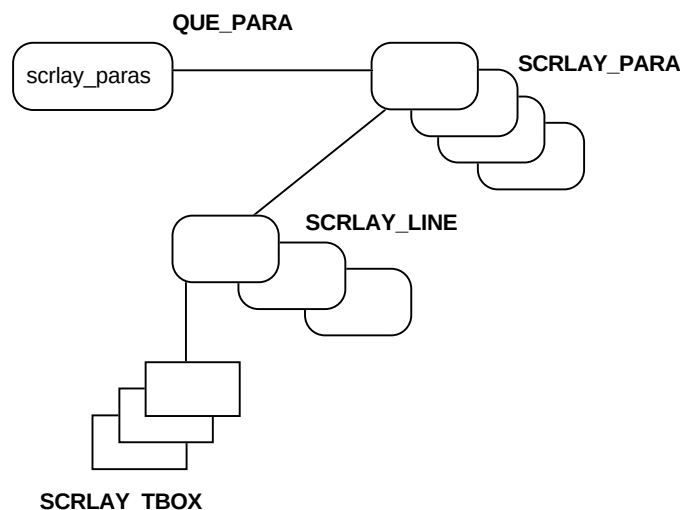
The data structure of prime importance to SCRLAY is that which is anchored in the property `scrlay.paras`. It is a model of the layout of that part of the document text which is visible. It contains sufficient information to allow the text to be drawn correctly.

The figure below illustrates the situation. `scrlay.paras` points to the head of a queue of SCRLAY_PARA type data structures each of which corresponds to a paragraph in the layout. From each of these is queued a number of SCRLAY_LINE type data structures each of which corresponds to an individual line on the screen.

A displayed line may be drawn using one or more text boxes. Typically, more than one textbox is used when a section of text within the line is displayed with a different font, emphasis or style to the neighbouring text in the line. Data structures of type SCRLAY_TBOX are used to *represent* these textboxes and are queued from the SCRLAY_LINE data structures.

It is important to note that the layout is built in whole paragraphs. Thus, where a paragraph is only partly on the screen, some of the corresponding SCRLAY_LINE and SCRLAY_TBOX data structures will represent lines and text boxes which are not visible.

In general, the methods of this class build or modify this data structure. For example, the `SL_SCROLL` method can have the effect of removing SCRLAY_PARA items from the queue at one end and adding new SCRLAY_PARA items at the other (depending on the direction of the scroll).



Special characters

SCRLAY treats certain characters in the content as special:

- character 0 marks a paragraph end.
- character 7 (`SCRLAY_SYM_HARD_HYPHEN`) represents a 'hard' hyphen, that is not treated as a word delimiter.

- character 9 (`w_KEY_TAB`) represents a tab character.
- character 10 (`LF`) represents a forced line break.
- character 14 (`SCRLAY_SYM_SOFT_HYPHEN`) represents a 'soft' hyphen (or potential hyphen).
- character 15 (`SCRLAY_SYM_HARD_SPACE`) represents a space that is not treated as a word delimiter.

To support the display of special symbols, all screen fonts must include distinctive graphics for the following characters:

- character 0 as a paragraph end symbol.
- character 7 as a hard hyphen symbol.
- character 8 (`SCRLAY_SYM_SHOW_SPACE`) as a 'shown' space (the same width as character 32)
- character 9 as a tab symbol.
- character 10 as a forced line break symbol.
- character 14 as a soft hyphen symbol.

Font-width tables

Font width tables are required when `SCRLAY` is in printer layout mode. There are two types of table:

- monospace fonts
- proportional fonts

A table for a monospace font just consists of two bytes - the first byte is `NULL` while the second contains the width of each character.

A table for proportional fonts contains 256 bytes; the first contains the value 1, to distinguish the table from a monospace font width table, while the remaining 255 bytes contain the widths of each of the 255 possible characters

SCRLAY methods

Most of these methods are used by an associated instance of the `SCRIMG` class. Instances of the `SCRLAY` and `SCRIMG` classes work in close co-operation. It is not expected that the creator of an instance of `SCRLAY` will directly access any methods other than `sl_init`, `sl_set` and `sl_sense`.

SL_INIT

Initialise

```
VOID sl_init(SCRLAY_DOC *doc, SCRLAY_STYLE *pstyle);
```

Initialise a newly created instance of `SCRLAY` by copying the supplied document content information and the global layout style data into `scr lay.doc` and `scr lay.st` respectively.

`scr lay.paras` which is the anchor point for a list (a doubly linked queue) of data items representing the layout of the data, is initialised.

The document content information is specified by the parameter `doc`, a pointer to a structure of type `SCRLAY_DOC`. The structure which is included as part of the `SCRLAY` class definition, is as follows:

```
typedef struct
{
    UWORD len;
    VOID *content;
    WORD sensechars;
    WORD sensepdata;
    WORD senseplabel;
    WORD toparst;
    WORD enqpage;
} SCRLAY_DOC;
```

The individual fields in the `SCRLAY_DOC` structure have the following meaning:

- `doc->len` is the length of the document in characters, including the terminating zero byte.
- `doc->content` is the handle of the object that contains the text - typically an instance of (or a subclass of) `EPDOC` or `EPFDOC`.
- the remaining fields specify the method numbers of call-back methods used to query the document content. Two of these methods - `doc->sensechars` and `doc->toparst` - are mandatory and are supplied by both `EPDOC` and `EPFDOC`.

The remaining three methods are optional. If they are not supplied, the corresponding fields of the `SCRLAY_DOC` structure (i.e. `doc->sensepdata`, `doc->senseplabel` and `doc->enqpage`) should be set to zero.

The requirements of all five of these methods are specified in the `FORMDOC` notional mixin class, described in *The Formatted Document Classes* chapter of this manual.

The global layout style is specified by the parameter `pstyle`, a pointer to a structure of type `SCRLAY_STYLE`.

The global layout style is the style applied to the whole document in the absence of any content-specific style supplied by the document content call-back methods. See the description of the `sl_set` method for a fuller discussion.

Finally, the method sets the `pend` member of the `scr lay .fmt` property to the highest possible value, i.e. the value `0xffff`.

SL_SET

Set global layout style

```
VOID sl_set(SCRLAY_STYLE *pstyle);
```

Set the global layout style (i.e. the style that is applied to the whole document in the absence of any content-specific style supplied by the document content call-back methods) and reset the property `scr lay .adjust` to zero.

The style is specified by the parameter `pstyle`, a pointer to a structure of type `SCRLAY_STYLE`.

The entire content of this data structure is copied into the object's `scr lay .st` property.

The `SCRLAY_STYLE` structure which is included as part of the `SCRLAY` class definition, is as follows:

```
typedef struct
{
    UBYTE options;
    UBYTE printer;
    SCRLAY_PDATA pd;
    SCRLAY_FONT *font;
    UBYTE *fwtab;
    UWORD scrpwidth;
    SCRLAY_FONT *sfont;
} SCRLAY_STYLE;
```

`pstyle->options` contains a set of flags which control whether certain optional features are to be included. Amongst these are flags which control whether special symbols, such as paragraph ends, are to be displayed.

The flags (which can be ored together) and their meaning are as follows:

<code>SCRLAY_SHOW_TABS</code>	if set, display tabs on screen
<code>SCRLAY_SHOW_SPACES</code>	if set, display spaces on screen
<code>SCRLAY_SHOW_CRS</code>	if set, show paragraph ends on screen
<code>SCRLAY_SHOW_HYPHENS</code>	if set, show all soft (potential) hyphens on screen
<code>SCRLAY_SHOW_LFS</code>	if set, show forced line breaks on screen
<code>SCRLAY_WIDOW_ORPHAN</code>	if set, enable widow and orphan suppression

The value of `pstyle->printer` is used to decide whether screen or printer layout is to be generated:

- `TRUE` generates printer layout
- `FALSE` generates screen layout

The `pstyle->pd` structure specifies global values for paragraph margins, tab positions and spacing. The three pointers `pstyle->pd.margins`, `pstyle->pd.tabs` and `pstyle->pd.spacing` must point to data structures of type `SCRLAY_MARGINS`, `SCRLAY_TABS` and `SCRLAY_SPACING` respectively. These data structures must remain in existence while being referenced by an instance of `SCRLAY`. Typically, this is for the lifetime of the `SCRLAY` object and it is common for the three data structures to form part of the property of the creator of the `SCRLAY` object.

The global settings can be overridden for any individual paragraph by content-specific paragraph layout specified by the optional call-back method whose method number can be found in the document content information, i.e. `scr lay.doc->sensepdata`.

The three pointers may be set to `NULL` if *all* of the corresponding paragraph layout is content-specific.

Each of `pstyle->font` and `pstyle->sfont` contain pointers to `SCRLAY_FONT` data structures that respectively specify the global printer font and the corresponding global screen font. If `pstyle->printer` is `FALSE`, both pointers may indicate the global screen font. The global settings can be overridden for any individual line segment by means of the `scr lay.doc->sensechars` call-back method.

If `pstyle->printer` is `TRUE`, `pstyle->fwtab` must contain a pointer to a global *printer* font width table. This only needs to be set in printer layout mode. The global setting can be overridden for any individual line segment by means of the `scr lay.doc->sensechars` call-back method.

The content of `pstyle->scrpwidth` is relevant only when generating printer layout, that is, when `pstyle->printer` is `TRUE`. In this case it defines the basis for the accurate on-screen representation of printer tab positions.

SL_SENSE

Sense global layout style

```
VOID sl_sense(SCRLAY_STYLE *pstyle);
```

Sense the global layout style, i.e. the style that is applied to the whole document in the absence of any content-specific style supplied by the document content call-back methods.

The global layout style data is written to a data structure of type `SCRLAY_STYLE` pointed to by the parameter `pstyle` and supplied by the caller.

DESTROY

Destroy

```
VOID destroy(VOID);
```

Destroy the instance.

The method destroys the instance by sending an `SL_DISCARD` message to free all memory used by the layout model before supersending a `DESTROY` message.

SL_POS_TO_XL Convert position to line number and pixel offset

```
INT sl_pos_to_xl(SCRLAY_PLX *plx);
```

Convert a character position within a document to the corresponding line number on the screen and the horizontal pixel offset of that character position within the line.

A data structure of type `SCRLAY_PLX` pointed to by the parameter `plx` is supplied by the caller; `plx->pos` contains the character position. The corresponding line number is written to `plx->line` and the corresponding horizontal pixel offset of that character within the line is written to `plx->x`.

If the character position is on the screen, the method returns 0.

If the character position is above the screen, the method writes a value of -30000 to `plx->line` and returns -1.

This method is used by the screen image class `SCRIMG`, and is not normally accessed directly by users of `FORM`.

SL_READ**Read line data**

```
INT sl_read(SCRLAY_READ *pr, TEXT *buf);
```

Read a line or part of a line from the currently generated layout and write information about it to a structure of type SCRLAY_READ pointed to by the parameter *pr.

The first call to this method will have been preceded by a call to the sl_begin_read method to identify the line.

If the parameter buf is not NULL, up to WS_MAX_PRINT_BOX_TEXT_LEN bytes of text are written to the buffer pointed to by buf. The actual number of bytes of text read by this method can be found in pr->blen.

When the method reads the *first* tbox in the line, it sets pr->isfirst to TRUE and sets the appropriate value into pr->indent, otherwise pr->isfirst is set to FALSE and pr->indent is set to zero. Similarly, when the last text box in the line is read, pr->islast is set to TRUE.

If there is more data to read, the method returns TRUE, otherwise it returns FALSE.

This method is used by the screen image class SCRIMG, and is not expected to be accessed directly by users of FORM.

SL_FORMAT_LINE**Format the next line**

```
INT sl_format_line(UWORD *ppos);
```

Format the next line using the document text belonging to the document object whose handle can be found in the property scr lay.doc. The status of the current format position is maintained in the property scr lay.fmt.

Provided ppos is not NULL, the character position at the start of this next line is written to the UWORD pointed to by ppos.

The method returns TRUE if there are more lines to format, otherwise it returns FALSE.

This method is used by the screen image class SCRIMG, and is not expected to be accessed directly by users of FORM.

SL_SCROLL**Scroll the layout**

```
INT sl_scroll(INT d1);
```

Scroll the screen layout by the number of lines specified by the parameter d1.

If d1 > 0 then the data moves *down* the screen, possibly generating additional layout for previous paragraphs and deleting layout for complete paragraphs which are now "below" the screen.

If d1 < 0 then the data moves *up* the screen, possibly generating additional layout for following paragraphs and deleting layout for complete paragraphs which are now "above" the screen.

This method should only be called when the absolute value of d1 is *less* than the number of lines displayed on the screen.

The method returns the actual number of lines by which the screen has scrolled (as limited by the bounds of the document). Note that the return value is signed; a value > 0 for the number of lines scrolled down the screen and a value < 0 for the number of lines scrolled up the screen.

This method is used by the screen image class SCRIMG, and is not expected to be accessed directly by users of FORM.

SL_VIEW**View position on given line**

```
INT sl_view(UINT pos, INT line);
```

Generate layout for sufficient paragraphs to enable document position pos to appear on the specified screen line.

The screen layout may be scrolled if the position is already on the screen; new layout will be generated as required.

Returns either the number of lines to scroll, or 0x7fff if the screen should be completely redrawn.

This method is used by the screen image class SCRIMG, and is not expected to be accessed directly by users of FORM.

SL_DISCARD_LAYOUT**Discard layout**

```
VOID sl_discard_layout(UINT doclen);
```

Set new document length and discard the whole layout.

If the parameter `doclen` is non-zero, then this is the new document length to be recorded. This value is set into the property `scr lay.doc.len`. If `doclen` is zero, the existing document length is not to be changed and `scr lay.doc.len` remains unaltered.

All currently generated layout is discarded by freeing all data items in the list anchored in the property `scr lay.paras`.

This method is used by the screen image class `SCRIMG`, and is not expected to be accessed directly by users of `FORM`.

SL_RESCALE**Adjust screen/printer scaling**

```
INT sl_rescale(VOID);
```

Adjust the screen/printer scaling.

If the property `scr lay.adjust` is `FALSE`, indicating no change to the width of the screen/printer, then the method does nothing but returns `FALSE`.

If the property `scr lay.adjust` is `TRUE`, indicating a change to the width of the screen:

- The existing layout is discarded and the document length is left unchanged.
- The layout is re-built so that the document position defined by `first.pos` will appear on the line defined by `first.line`.
- The method returns `TRUE`.

This method is used to rescale the screen layout as necessary to ensure that printer tab positions are displayed accurately, despite differences between the screen and printer fonts. It is called by the screen image class `SCRIMG`, and is not expected to be accessed directly by users of `FORM`.

SL_SET_LINES**Set number of lines**

```
VOID sl_set_lines(INT slines);
```

Set the number of displayable lines in the layout to the value in the parameter `slines`.

The method simply takes the value in the parameter `slines` and sets it into the property `scr lay.slines`.

This method is used by the screen image class `SCRIMG`, and is not expected to be accessed directly by users of `FORM`.

SL_PARA_CHANGED**Discard lines from position**

```
INT sl_para_changed(INT code, SCRLAY_PLX *plx, WORD *lgood);
```

Discard one or more lines of layout as a result of the change in the document content specified by the parameter `code` at document position `plx->pos`.

The change may be a left delete of one character (`code` is `W_KEY_DELETE_LEFT`) a right delete of one character (`code` is `W_KEY_DELETE_RIGHT`) or the insertion of a single content character, for which `code` is one of :

- a printable character code
- zero (paragraph end)
- `W_KEY_TAB`
- `'\n'`

The value of `code` is used to infer the new document length.

In preparation for a following sequence of calls to `scrlay_sl_format_line`, the method updates `plx->line` to the line number of the first line that needs to be reformatted and sets `plx->pos` to the character position at the start of this line. The line number of the first following line that does not need to be reformatted is written to the word pointed to by the parameter `lgood`.

Returns `TRUE` if a paragraph end was deleted, otherwise returns `FALSE`.

This method is used by the screen image class `SCRIMG`, and is not expected to be accessed directly by users of `FORM`.

WRAP

ACTIVE	WRAP
q priority isactive pcb stat	scrimg
destroy ao_init ao_cancel ao_abrun ao_queue ao_run	ao_init ao_queue ao_run

An instance of the `WRAP` active object class is created and initialised by `SCRIMG`, which uses it to reformat lines of text in background.

Background formatting is restarted (from the `SCRIMG si_para_changed` method) each time a key is pressed to insert or delete a character, but the reformatted text is only redrawn when such formatting runs to completion. This means that the response to keypresses is not degraded by the whole of the affected text being redrawn for each keypress.

The expectation is that `WRAP` will not be subclassed, and that it will only be used by `SCRIMG`.

Class definition

`WRAP` subclasses the `OLIB` class `ACTIVE` and is defined in the sub-category file *scrimg.cl* (with generated header file *scrimg.g*).

```
CLASS wrap active
{
  REPLACE ao_init
  REPLACE ao_queue
  REPLACE ao_run
  PROPERTY
  {
    VOID *scrimg;
  }
}
```

Property

<code>wrap.scrimg</code>	The handle of the owning instance of the <code>SCRIMG</code> class.
--------------------------	---

WRAP methods

AO_INIT

Initialise

```
VOID ao_init(VOID *scrimg);
```

Initialise this instance of WRAP.

The method takes one parameter; `scrimg` contains the handle of the instance of the SCRIMG class which has created this WRAP object.

The method does the following:

- Sets the handle of the creating instance of the SCRIMG class into the property `wrap.scrimg`.
- Adds the WRAP object to the active object task queue with a priority of `PRIORITY_ACTIVE_COMPUTE`. This is a low priority so that the it will only run if there are no higher priority tasks ready to run.

AO_QUEUE

Queue a line format request

```
VOID ao_queue(VOID);
```

If the WRAP object is not currently active, it supersends an `AO_QUEUE` message to queue a request to format a line.

AO_RUN

Format a line

```
INT ao_run(VOID);
```

Re-format a line.

The method limits itself to re-formatting a single line. If there are more lines to be formatted, an `AO_QUEUE` message is sent to schedule the re-format of the next line. By doing this, the WRAP object allows other (higher priority) active objects to run.

Once formatting has finished, all reformatted lines are re-drawn and *no* further `AO_QUEUE` messages are sent.

SCRIMG

SCRIMG	
<code>wrap</code>	<code>select</code>
<code>lay</code>	<code>formatting</code>
<code>win</code>	<code>isredraw</code>
<code>anc</code>	<code>GCcreated</code>
<code>crs</code>	<code>emphasised</code>
<code>oldcrs</code>	<code>plabchange</code>
<code>txwidth</code>	<code>nopan</code>
<code>mrwidth</code>	<code>flags</code>
<code>xleft</code>	<code>lfmt</code>
<code>xright</code>	<code>lgood</code>
<code>lcwidth</code>	<code>xo</code>
<code>lcline</code>	<code>gc</code>
<code>updownx</code>	
<code>destroy</code>	<code>si_move_cursor</code>
<code>si_init</code>	<code>si_redraw</code>
<code>si_set</code>	<code>si_doc_changed</code>
<code>si_sense</code>	<code>si_doc_reset</code>
<code>si_emphasize</code>	<code>si_para_changed</code>
<code>si_get_select</code>	<code>si_style_changed</code>
<code>si_pan</code>	<code>si_delprep</code>
<code>si_scroll</code>	<code>si_fwd_change</code>
<code>si_view</code>	

The SCRIMG class provides the means of displaying formatted, editable text in a window. The class provides methods that manipulate the screen display including methods to perform scrolling and re-drawing.

SCRIMG works in close co-operation with an instance of the SCRLAY class, which formats the text that SCRIMG is to display.

Although SCRIMG relies on the presence of a document content object - generally an instance of (a subclass of) EPDOC or EPFDOC - it has no direct knowledge whatsoever about the nature or interpretation of the document content that is being displayed.

In particular, SCRIMG has no concept that the whole of the document may be already divided up into lines - lines for SCRIMG mean nothing more than lines within the screen area that SCRIMG draws to. In illustration of this point, it is worth noting that in the Word application, formatting information - including the locations of line breaks - only exists for the portion of the document that is displayed on the screen. Formatting information for other regions of the document is discarded as soon as it is no longer required for display purposes.

The area drawn to by SCRIMG is divided into three vertical regions:

- an optional label margin
- an optional line cursor region
- an area for formatted text; this includes the text cursor

This division of the drawing area is illustrated in the figure shown after the description of the property, below.

The Series 3a built-in word processor provides a good illustration of this. Setting the "Show style bar" option in the "Set preferences" menu item of the "Special" menu to YES shows the word processor window divided into the three vertical regions.

Class definition

The SCRIMG class subclasses ROOT and is defined in the sub-category file *scrimg.cl* (with generated header file *scrimg.g*).

```
CLASS scrimg root
{
  REPLACE destroy          Remove any text cursor
  ADD si_init              Optional set then builds layout
  ADD si_set               Set win and layout
  ADD si_sense             Sense win setup data
  ADD si_emphasize         Set emphasis on/off
  ADD si_get_select        Return selection as pos,len
  ADD si_pan               Horizontally scroll the image by pixels
  ADD si_scroll            Scroll the image by lines
  ADD si_view              Show pos on specified line
  ADD si_move_cursor       Set the cursor position
  ADD si_redraw            Draw to given rectangle
  ADD si_doc_changed       Discard layout, cancel select and redraw
  ADD si_doc_reset         Discard layout, cancel select, view and redraw
  ADD si_para_changed      Background reformat of para from cursor pos
  ADD si_style_changed     Re-evaluate layout
  ADD si_delprep           Prepare for a left delete
  ADD si_fwd_change        Like si_doc_changed except pivot from screen top
}

CONSTANTS
{
  ! si_move_cursor actions
  SCRIMG_LINEDN 0x00
  SCRIMG_LINEUP 0x01
  SCRIMG_PAGEDN 0x02
  SCRIMG_PAGEUP 0x03
  SCRIMG_LINBEG 0x04
  SCRIMG_LINEND 0x05
  SCRIMG_SETPOS 0x06
}
```

```

! States for background formatting
SCRIMG_FORMAT_DELETE_LEFT    1
SCRIMG_FORMAT_DELETE_RIGHT   2
SCRIMG_FORMAT_TYPING         3

! si_pan horizontal scroll (panning) modes
SCRIMG_PAN_SETNOPAN         0
SCRIMG_PAN_DELTA            1
SCRIMG_PAN_ABS               2

! si_style_changed qualifiers
SCRIMG_STCHNG_DOC           0      Reformat whole document
SCRIMG_STCHNG_PARA          1      Reformat from paragraph
SCRIMG_STCHNG_LINE          2      Reformat from previous line

SCRIMG_FLAGS_PAGEBREAK      0x01
SCRIMG_FLAGS_S3_COMPAT      0x02
}

```

TYPES

```

{
typedef struct
{
    UWORD wid;          window ID
    P_POINT tl;         top left corner of area being drawn to
    WORD nlines;        number of text lines to display
    UBYTE lheight;      line height in pixels
    UBYTE lascent;      distance from top of line to text base line
    WORD width;         total width in pixels (margin,line cursor,text)
    WORD margin;        width of label margin in pixels
    WORD lcfont;        line cursor font (or zero for no line cursor)
    UBYTE cwidth;       text cursor width
    UBYTE lcstyle;      line cursor style
    UBYTE lcode;        line cursor character code
    UBYTE hscrlx;       horizontal scroll x jump
    UBYTE hscrlm;       horizontal scroll margin
    UBYTE drawplabs;    draw trailing para labels after formatting if set
} SCRIMG_WIN;
}

```

PROPERTY 1

```

{
    PR_WRAP *wrap;      background word wrap active object
    PR_SCLAY *lay;      the document screen layout
    SCRIMG_WIN win;     describes the window to be drawn to
    SCLAY_PLX anc;      select anchor position
    SCLAY_PLX crs;      text cursor position
    SCLAY_PLX oldcrs;   old text cursor position
    WORD txwidth;       width of text area in pixels
    WORD mrwidth;       width of margin plus line cursor
    WORD xleft;         left clip margin
    WORD xright;        right clip margin
    UBYTE lcwidth;      width of line cursor area
    UBYTE lcline;       current line for line cursor
    WORD updownx;       latent x for cursor up/down movement
    UBYTE select;       TRUE if there is a selection
    UBYTE formatting;   TRUE if background formatting
    UBYTE isredraw;     TRUE if performing a redraw
    UBYTE GCcreated;    TRUE if a temp GC has been created
    UBYTE emphasised;   TRUE if emphasis is on
    UBYTE plabchange;   TRUE if para labels may have changed
    UBYTE nopan;        don't pan to expose cursor if TRUE
    UBYTE flags;        SCRIMG_FLAGS_PAGEBREAK, SCRIMG_FLAGS_S3_COMPAT
    WORD lfnt;          Next line to be replaced by background format
    WORD lgood;         First line not requiring to be formatted
    WORD xo;            x postion of left of layout for horiz scroll
    G_GC gc;           current graphics context
}

```

}

Property

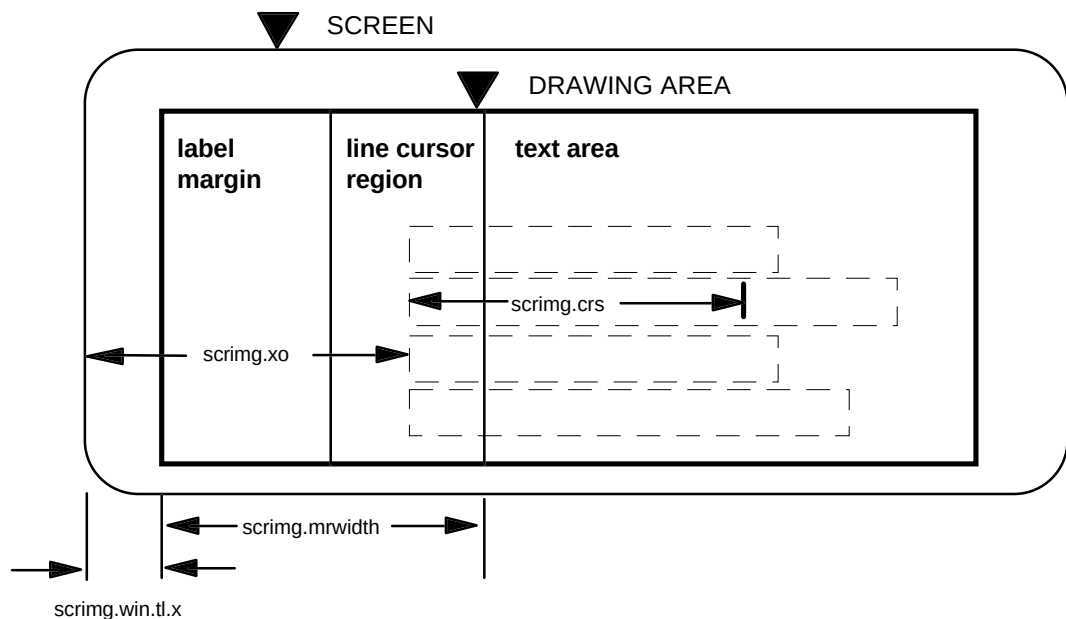
<code>scrimg.wrap</code>	The handle of a background word-wrapping active object, expected to be an instance of <code>WRAP</code> . This object is created and owned by <code>SCRIMG</code> .
<code>scrimg.lay</code>	The handle of an instance (or a subclass) of <code>SCRLAY</code> . This instance is created by the creator of <code>SCRIMG</code> and its handle passed as a parameter to the <code>si_init</code> or <code>si_set</code> methods.
<code>scrimg.win</code>	A data structure of type <code>SCRIMG_WIN</code> containing information on the window within which the document content is displayed. The data structure is created by the creator of <code>SCRIMG</code> and its handle passed as a parameter to the <code>si_init</code> or <code>si_set</code> methods.
<code>scrimg.anc</code>	The anchor position for a select region, that is, a region of highlighted text, stored as a character offset from the start of the document and as a screen line number and x-offset in that line.
<code>scrimg.crs</code>	The current position of the text cursor, stored as a character offset from the start of the document and as a screen line number and screen x-offset in that line.
<code>scrimg.oldcrs</code>	A copy of a previous cursor position, described in terms of both a document and a screen position, as for <code>scrimg.crs</code> . This property is used by the <code>si_move_cursor</code> and <code>si_scroll</code> methods.
<code>scrimg.txwidth</code>	The width, in pixels, of the <i>text area</i> . This is set by the <code>si_set</code> method and is calculated as the width of the drawing region <i>minus</i> the combined width of the label margin and the line cursor region.
<code>scrimg.mrwidth</code>	The sum of the widths, in pixels, of the label margin and the line cursor region. This is calculated and set by the <code>si_set</code> method.
<code>scrimg.xleft</code>	This defines the left hand horizontal pixel position for clipping.
<code>scrimg.xright</code>	This defines the right hand horizontal pixel position for clipping. In general terms, those parts of a rectangle which extend outside a region bounded by <code>scrimg.xleft</code> and <code>scrimg.xright</code> , are clipped.
<code>scrimg.lcwidth</code>	The width, in pixels, of the line cursor region. This is set by the <code>si_set</code> method and is calculated as the sum of the width of the line cursor character plus two (pixels).
<code>scrimg.lcline</code>	The number of the line on the screen containing the <i>line cursor</i>
<code>scrimg.updownx</code>	The "latent" horizontal pixel offset of the text cursor. It records the default horizontal pixel position to which the cursor is moved during <i>vertical</i> scrolling and cursor movement operations. For example, when the text cursor is moved up one line, it defines where on that line the cursor should be placed. This property can be changed by a number of methods.
<code>scrimg.select</code>	Set to <code>TRUE</code> if there is currently a select region.
<code>scrimg.formatting</code>	Set to <code>TRUE</code> if background formatting is in progress.
<code>scrimg.isredraw</code>	Set to <code>TRUE</code> if redrawing is in progress.
<code>scrimg.GCcreated</code>	Set to <code>TRUE</code> if a temporary graphics context exists.
<code>scrimg.emphasised</code>	Set to <code>TRUE</code> if the window containing the drawing area has the emphasis. This is set and unset by the <code>si_emphasize</code> method.
<code>scrimg.plabchange</code>	Set to <code>TRUE</code> if paragraph labels have (or might have) changed.
<code>scrimg.nopan</code>	Set to <code>TRUE</code> if horizontal scrolling is disabled; while this is set, horizontal scrolling is prevented - in particular, during attempts to make the cursor visible.

<code>scrimg.flags</code>	This property contains a number of flags which can be a combination of the following: <code>SCRIMG_FLAGS_PAGEBREAK</code> If set, at least one pagebreak line has been drawn on the screen during the lifetime of the <code>SCRIMG</code> object. Once set, it is never unset. <code>SCRIMG_FLAGS_S3_COMPAT</code> If set, the Series 3a is running in Series 3 compatibility mode.
<code>scrimg.lfmt</code>	The number of the next line on the screen to be formatted when background formatting.
<code>scrimg.lgood</code>	After a character has been inserted or deleted, a number of lines may need reformatting; this is done in background. This property will contain the number of the first following line on the screen that does not require reformatting.
<code>scrimg.xo</code>	The horizontal pixel position of the left hand edge of the text relative to the window. This is set initially by the <code>si_set</code> method to be co-incident with the left hand edge of the <i>text area</i>. (i.e the horizontal pixel position of the drawing region <i>plus</i> the width of the label margin <i>plus</i> the width of the line cursor region). As the text is scrolled horizontally, the position of the left hand edge of the text moves; this property is changed accordingly to reflect the new position of the left hand edge of the text which may or may not be visible. This value can be negative. This is graphically illustrated in the figure below
<code>scrimg.gc</code>	The current graphics context.

The following diagram illustrates the meaning of some of the property items discussed above.

It shows a typical situation where the drawing area contains a label margin, a line cursor region and a text area. The dotted lines represent lines of text which are shown as having been scrolled. Text which is "outside" the text area is not visible.

All the measurements depicted are in units of 1 pixel.



SCRIMG methods

DESTROY

Destroy

```
VOID destroy(VOID);
```

Destroy the instance of SCRIMG.

If the window containing the drawing area has the emphasis, then the text cursor is removed by calling the window server function `wEraseTextCursor`. The method concludes by supersending a `DESTROY` message.

SI_INIT

Initialise

```
VOID si_init(SCRIMG_WIN *win, VOID *lay);
```

Initialise SCRIMG ready to view text.

The method takes two parameters:

- `win` points to a data structure of type `SCRIMG_WIN` which contains information on the window within which the document content is to be displayed.
- `lay` is the handle of an associated instance of the `SCRLAY` class.

If the processor is running in compatibility mode, the flag `SCRIMG_FLAGS_S3_COMPAT` is set in the property `scrimg.flags`.

The `si_set` method is called to copy the content of the window information data structure and the handle of the `SCRLAY` object into the property `scrimg.win` and `scrimg.lay` respectively. One or both of the parameters `win` and `lay` may be `NULL` but, if so, then `si_set` must have been called previously with valid non `NULL` values for `win` and `lay`.

`SL_DISC_RD_LAYOUT` and `SL_VIEW` messages are sent to the screen layout (`SCRLAY`) object to prepare the layout for the document text so that the start of the document (character position zero) will be on the first line (line zero) of the window.

This method does not draw anything except for the text cursor - the text will normally be drawn by a subsequent call to the `si_redraw` method.

SI_SET

Set view and layout

```
INT si_set(SCRIMG_WIN *win, VOID *lay);
```

Set the window information and the handle of the `SCRLAY` object.

The method takes two parameters:

- `win` points to a data structure of type `SCRIMG_WIN` which contains information on the window within which the document content is to be displayed. This can be a `NULL` value.
- `lay` is the handle of an associated instance of the `SCRLAY` class. This can be a `NULL` value.

The method waits for any background formatting to complete, before doing anything else.

If the parameter `lay` is not `NULL`, its value is copied into the property `scrimg.lay`. Similarly, if the parameter `win` is not `NULL`, the entire content of `*win` is copied into the property `scrimg.win`.

If, at this stage, `scrimg.win` is not `NULL` then the following settings and calculations are done:

- The font ID in the current graphics context is set to the default value (`WS_FONT_BASE`) and the style is set to normal (`G_STY_NORMAL`).
- If a line cursor font ID is supplied (i.e. `win->lcfont` is non-zero), the method calculates the width of the line cursor region and sets the value into the property `scrimg.lcwidth`.
- The width of the line cursor region *plus* the label margin is calculated and set into the property `scrimg.mrwidth`.
- The horizontal pixel position of the text area is calculated and set into the property `scrimg.xo`.
- The width of the text area is calculated and set into the property `scrimg.txwidth`.

An `SL_SET_LINES` message is sent to the screen layout object, passing the value of `win.nlines` so that it knows the maximum number of text lines that are to be displayed on the screen.

SCRIMG_WIN which is included as part of the SCRIMG class definition, is as follows:

```
typedef struct
{
    UWORD wid;
    P_POINT tl;
    WORD nlines;
    UBYTE lheight;
    UBYTE lascent;
    WORD width;
    WORD margin;
    WORD lcfont;
    UBYTE cwidth;
    UBYTE lcstyle;
    UBYTE lccode;
    UBYTE hscr1x;
    UBYTE hscr1m;
    UBYTE drawplabs;
} SCRIMG_WIN;
```

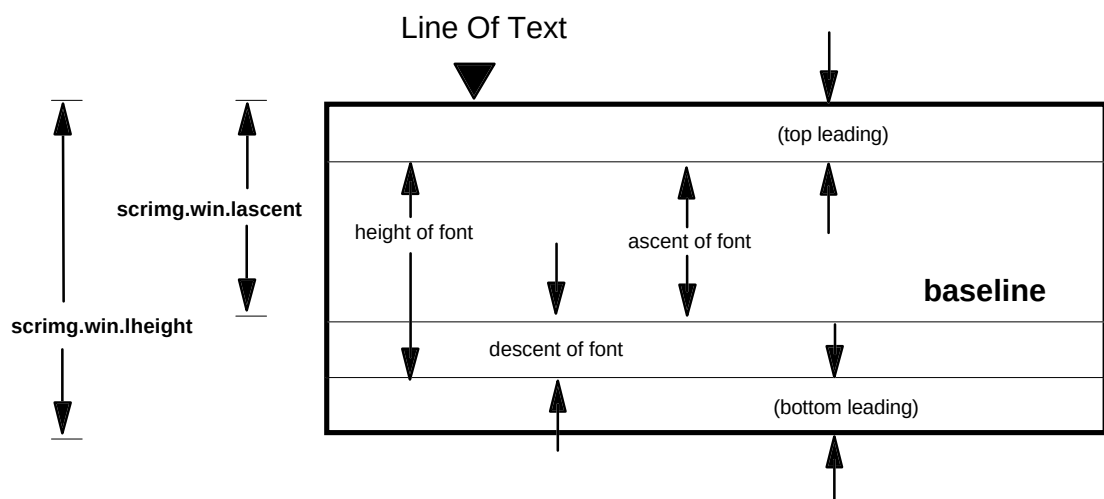
The `win->wid` element specifies the window ID (as returned by a call to `wCreateWindow`) of the window to which drawing is done. This will normally relate to the window object that creates and initialises the `SCRIMG` object.

The area within this window within which `SCRIMG` draws is defined by:

- the coordinates of its top-left hand corner, given by `win->tl.x` and `win->tl.y`
- the total width, in `win->width`
- the total height, calculated from the number of lines, `win->nlines`, multiplied by the line height, `win->lheight`.

These, and all other dimensions in the `SCRIMG_WIN` struct, are specified in pixels.

The line height will normally be the height of the screen font in which the text is displayed, plus one or two pixels of additional space, known as *leading*. The base line for drawing characters within a line of text is defined by `win->lascent`, which will normally be equal to the ascent of the screen font plus the *leading*. This is illustrated below. More information on fonts can be found in the section on *Text output functions* in the *Window Server Reference* manual.



In addition to the text region, the drawing area may contain a label margin and a line cursor margin, as described earlier in this chapter.

The width of the label margin, used to display paragraph labels, is specified by `win->margin`. Labels will only be drawn if `win->drawplabs` is TRUE (in this case `SCRLAY` will need to be supplied with a `senseplabel` call-back method). If `win->drawplabs` is FALSE, `win->margin` may be zero and a `senseplabel` call-back method need not be specified.

A line cursor will be displayed in the line cursor margin if `win->lcfont` is non-zero. Its value should be the font ID of the screen font that contains the line cursor character. The character code of the line cursor character itself, is supplied in `win->lccode`. The required line cursor style attribute, for example BOLD, is specified by `win->lcstyle`.

A text cursor will be displayed in the text area if `win->cwidth` is non-zero. Its value should be the required width of the text cursor, normally one or two pixels.

Automatic horizontal scrolling of text within the text area, provided lines of text are longer than the width of the text area, is controlled by `win->hscr lx` and `win->hscr lm`. The value of `win->hscr lx` specifies the unit of horizontal scrolling motion (a unit being some number of pixels). The document will scroll, if necessary, when the text cursor reaches the right hand edge of the text area or when the text cursor moves to within `win->hscr lm` pixels of the left hand edge of that area.

A call to `si_set`, other than one that precedes a call to `si_init` or the one that is made by `si_init` itself, should be followed by a call to `si_doc_changed` to force the layout to be rebuilt and the content to be redrawn.

The method returns the width, in pixels, of the text area.

Note that this method offers a simple way of waiting for the completion of background formatting, by calling it with both parameters set to NULL.

SI_SENSE **Sense window information**

```
VOID si_sense(SCRIMG_WIN *win);
```

Write a copy of SCRIMG's window information data structure, as contained in the property `scrimg.win`, to the location pointed to by the parameter `win`; this is expected to point to a structure of type `SCRIMG_WIN` supplied by the caller of the method.

SI_EMPHASIZE **Set emphasis on or off**

```
VOID si_emphasize(INT on);
```

Turn emphasis on or off.

The method takes a single parameter; `on` has the value TRUE or FALSE.

When the window containing the drawing area has the emphasis, this is indicated by setting the property `scrimg.emphasised` to TRUE.

If the parameter `on` has the value TRUE, the method indicates that the emphasis is on by setting the property `scrimg.emphasised` to TRUE. If the parameter `on` has the value FALSE, the method indicates that the emphasis is off by setting the property `scrimg.emphasised` to FALSE. Repeated calls with the same value of `on` do nothing.

This method is often called from the `wn_emphasis` method of an instance of the `WIN` class (or more likely, a subclass of `WIN`).

Turning the emphasis off causes the text cursor to be removed and the highlight of any selected text to be removed.

Turning the emphasis on causes the text cursor to be drawn and any selected text to be highlighted.

SI_GET_SELECT **Get select region**

```
UINT si_get_select(UWORD *ppos);
```

Write, to `*ppos`, the document position of the first character (the character nearest to the beginning of the document) of the select region. This will be either the anchor position or the cursor position, depending on the direction in which the selection was made.

The method returns the length of the select region.

If there is no select region, the current cursor position is written to `*ppos` and the method returns zero.

SI_PAN**Scroll the image horizontally**

```
VOID si_pan(INT func, INT par);
```

Scroll the image horizontally after any background formatting is completed.

This method has three modes of operation depending on the value of the parameter `func`. The interpretation of the parameter `par` depends on the mode of operation.

`func` can take one of the following values:

SCRIMG_PAN_SETNOPAN	The method either disables or enables horizontal scrolling depending on the value of <code>par</code> . If <code>par</code> has the value <code>TRUE</code> , horizontal scrolling is <i>disabled</i> ; a value of <code>FALSE</code> <i>enables</i> horizontal scrolling. The value of <code>par</code> is set into the property <code>scrimg.nopan</code> .
SCRIMG_PAN_DELTA	Scroll the image horizontally by <code>par</code> pixels. <code>par</code> may be negative or positive. If <code>par</code> is positive, the image is scrolled to the <i>left</i> by <code>par</code> pixels; if negative, the image is scrolled to the <i>right</i> by <code>par</code> pixels.
SCRIMG_PAN_ABS	Scroll the image such that the position <code>par</code> is at the <i>left</i> of the view. The scroll is limited to reasonable limits. Scrolling past the right hand end of the longest visible line is prevented.

SI_SCROLL**Scroll the image vertically**

```
INT si_scroll(INT dl);
```

Scroll the image vertically by the number of lines specified by the parameter `dl`.

The direction of scroll depends on the sign of `dl`. If positive (i.e. `dl>0`), the image moves down, bringing in new paragraphs from above; if negative (i.e. `dl<0`), the image moves up, bringing in new paragraphs from below. The method should only be called when the absolute value of `dl` is *less* than the number of lines displayed on the screen.

After any background formatting is complete, a `SL_SCROLL` message is sent to the `SCRLAY` object to scroll the screen layout by `dl` lines. The `sl_scroll` method returns the number of lines actually scrolled and this value is used to update:

- the number of the line on which the cursor is displayed (a component of `scrimg.crs`)
- the line number for any previous text cursor (a component of `scrimg.oldcrs`)
- the line number of the anchor position for any select region (a component of `scrimg.anc`)

The screen display itself is scrolled by the number of lines returned by the `sl_scroll` method, i.e. the number of lines by which the screen layout was scrolled.

Note that the amount scrolled is limited by the bounds of the document.

The method returns the actual number of lines scrolled, positive if scrolled down or negative if scrolled up.

SI_VIEW**Show position on given line**

```
VOID si_view(UINT pos, INT line);
```

Draw a view of the document such that, subject to limitations imposed by the bounds of the document, document position `pos` is on screen line number `line`.

Any background formatting is allowed to complete first. A `SL_VIEW` message to the `SCRLAY` object to arrange the screen layout to satisfy this request.

If the document position is already on the screen, the screen display is scrolled, otherwise it is completely re-built.

SI_MOVE_CURSOR**Set the cursor position**

```
INT si_move_cursor(INT select, INT type, UWORD *ppos);
```

Set the cursor position.

Any background formatting is allowed to complete first.

The movement of the cursor is controlled by the value of the parameter `type` which can take one of the following values:

SCRIMG_SETPOS The cursor is moved to the document position specified by the value pointed to by the parameter `ppos`.

If the position is already visible, no scrolling or re-building of the screen display is done.

If the position is "above" the current display, the screen content is scrolled or re-built so that the line containing the specified position lies at the top of the screen.

If the position is "below" the current display, the screen content is scrolled or re-built so that the line containing the specified position lies at the bottom of the screen.

SCRIMG_LINEDN The cursor is moved down by one line.

If the resulting line is "below" the current display, the screen content is scrolled or re-built so that this line lies at the bottom of the screen.

The horizontal pixel position of the cursor is set to the latent value as recorded in `scrimg.updownx`. However, if the cursor was already on the last displayable line, it will be positioned at the end of the line.

SCRIMG_LINEUP The cursor is moved up by one line.

If the resulting line is "above" the current display, the screen content is scrolled or re-built so that this line lies at the top of the screen.

The horizontal pixel position of the cursor is set to the latent value as recorded in `scrimg.updownx`. However, if the cursor was already on the first displayable line, it will be positioned at the beginning of the line.

SCRIMG_PAGEDN The cursor is moved down by a number of lines equal to the number of lines displayed in the text area *minus* one.

If the resulting line is "below" the current display, the screen content is scrolled or re-built so that this line lies at the bottom of the screen.

The horizontal pixel position of the cursor is set to the latent value as recorded in `scrimg.updownx`. However, if the cursor was already on the last displayable line, it will be positioned at the end of the line.

SCRIMG_PAGEUP The cursor position is moved up by a number of lines equal to the number of lines displayed in the text area *minus* one.

If the resulting line is "above" the current display, the screen content is scrolled or re-built so that this line lies at the top of the screen.

The horizontal pixel position of the cursor is set to the latent value as recorded in `scrimg.updownx`. However, if the cursor was already on the first displayable line, it will be positioned at the beginning of the line.

SCRIMG_LINBEG The cursor is moved to the beginning of the line on which it is currently positioned. The horizontal pixel offset of the text cursor is taken as the new "latent" value and is recorded in the property `scrimg.updownx`.

SCRIMG_LINEND The cursor is moved to the end of the line on which it is currently positioned. The horizontal pixel offset of the text cursor is taken as the new "latent" value and is recorded in the property `scrimg.updownx`.

The new document position is written to *ppos.

The select region is modified if the parameter `select` is `TRUE`, otherwise any selection is cancelled and the method returns `FALSE`.

If there is a selected region after the movement of the cursor, the method returns `TRUE`.

SI_REDRAW

Draw to given rectangle

```
VOID si_redraw(P_RECT *prect);
```

Redraw the region specified by the rectangle whose address is given by the parameter `prect`. The method assumes that a temporary or permanent graphics context has already been set up. The graphics context is frequently modified by calls to `gsetgc` during drawing.

The method may be called:

- from within a redraw performed in response to a window server `WM_REDRAW` message. Such a message may be ignored by a window with a backup bitmap.
- from code that draws directly to the view.

If `prect` is `NULL` the whole window is redrawn, otherwise those lines that intersect with the rectangle specified by `prect` are redrawn. If the specified rectangle does not intersect with the window, no re-drawing is done.

Any background formatting is allowed to complete before any re-drawing is attempted.

While re-drawing is in progress, the property `scrimg.isredraw` is set to `TRUE`; this is re-set to `FALSE` when re-drawing is complete.

On the Series 3, and on other machines when running in Series 3 compatibility mode, `prect` is ignored, and the whole display area is always redrawn, but a value (`NULL` if necessary) should always be supplied for `prect`.

SI_DOC_RESET

Discard screen layout, view and redraw

```
VOID si_doc_reset(UINT doclen, UINT pos, INT line);
```

Cancel any existing select region and discard any current layout. Rebuild the screen layout as necessary and redraw the view subject to the limitations imposed by the bounds of the document.

The parameter `doclen` should contain the new length of the document (which should include the terminating `NULL`); this may be zero if the length is unchanged.

The view is rebuilt such that document position `pos` is visible on the screen on line number `line`. The value of `line` may be `-1`, in which case document position `pos` should, if possible, be displayed on the screen line currently containing the cursor.

This method is intended to be used to when a new document has replaced the original one; typically, it is used after the application has opened or created a new document.

The method is also used if there has been a sufficiently large change to the document content to justify re-building the view from first principles.

SI_DOC_CHANGED

Discard screen layout and redraw

```
VOID si_doc_changed(UINT doclen);
```

Cancel any existing select region and discard any current layout. Rebuild the screen layout as necessary and redraw the view subject to the limitations imposed by the bounds of the document.

The parameter `doclen` should contain the new length of the document (which should include the terminating `NULL`); this may be zero if the length is unchanged.

The view is rebuilt such that whatever character now occupies the current cursor position appears on the same screen line as the current cursor.

The method is intended to be used if there has been a sufficiently large change to the document content to justify re-building the view from first principles.

This method is almost identical to `si_doc_reset` except that it does not allow the document position and line number to be changed.

SI_DELPREP Prepare for a left delete

```
VOID si_delprep(SCRLAY_PLX *old);
```

Provide useful performance-enhancing information before performing a left delete. This method is usually called prior to calling the `si_para_changed` method with a `W_KEY_DELETE_LEFT` character code (see later for a description of this method).

The parameter `old` must point to a `SCRLAY_PLX` type data structure.

The method writes the document position of the character which is to the left of the current cursor to `old->pos` and writes the corresponding screen position to `old->x` and `old->line`. If this position is off-screen or not on same line as the cursor, a value of -1 is written to `old->line`.

SI_PARA_CHANGED Draw paragraph to echo content change

```
VOID si_para_changed(INT code, SCRLAY_PLX *old);
```

Immediately echo the content change, indicated by the parameter `code`, to the screen display and then reformat and redraw in background. Any selected region is cancelled.

The change indicated by `code` may be a left delete of one character (where `code` has the value `W_KEY_DELETE_LEFT`), a right delete of one character (where `code` has the value `W_KEY_DELETE_RIGHT`) or the insertion of a single content character, for which `code` is one of :

- a printable character code
- zero (paragraph end)
- `W_KEY_TAB`
- `'\n'`

The parameter `old` is only relevant when `code` has the value `W_KEY_DELETE_LEFT`; it should point to a `SCRLAY_PLX` type data structure and should contain the document position and corresponding screen position of the character which is to the left of the current cursor (as returned by the method `si_delprep`).

If `code` has any value other than `W_KEY_DELETE_LEFT`, `old` can be set to `NULL`.

This method supplies responsiveness to the most common keyboard operations used when editing a document.

SI_STYLE_CHANGED Redraw to echo a style change

```
VOID si_style_changed(INT type);
```

Rebuild the screen layout and redraw one or more lines as appropriate, following a style change. The current cursor position and any select region are maintained and, in general, the cursor remains on the same line of the screen.

The value of the parameter `type` specifies the nature of the change and can be one of the following values:

<code>SCRIMG_STCHNG_DOC</code>	A style change has occurred that potentially affects the whole document. The screen layout is rebuilt and the whole display redrawn. A typical use would be following a change in the base font used for the document
<code>SCRIMG_STCHNG_PARA</code>	A style change has occurred in the paragraph containing the cursor or the range of paragraphs that contain the select region, and affecting only that paragraph or paragraph range. The screen layout is rebuilt, from the start of the first paragraph in the range (excluding paragraphs that are entirely invisible) and the display is redrawn as appropriate. A typical use would be following a change in the margin positions of a single paragraph.

SCRIMG_STCHNG_LINE A style change has occurred in the line containing the cursor or the range of lines that contain a select region. The screen layout is rebuilt, from the beginning of the line before that in which the change starts, and the display is redrawn as appropriate. A typical use would be following a change in emphasis of one or more words.

Note that on the Series 3 the whole view is redrawn in all cases, regardless of the value of type.

SI_FWD_CHANGE Redraw for changes beyond cursor position

`VOID si_fwd_change(UINT doclen);`

Redraw the screen forward of the current cursor position and record a new document length.

The parameter `doclen` should contain the new length of the document (which should include the terminating `NULL`); this may be zero if the length is unchanged.

Any existing select region is cancelled.

The existing screen layout is discarded and re-built afresh such that the character originally on the first line of the screen and occupying the first position on that line, retains that position.

All lines from:-

 the line above that which contains the cursor

to:-

 the last line

are redrawn; in effect, the top of the screen is frozen while those lines from the cursor downwards are redrawn.

This method may be used following any change to the document content that does not affect the content before the current cursor position.

CHAPTER 4

THE DOCUMENT PRINTING CLASSES

The document printing classes are a set of classes which, co-operatively, permit documents to be printed, previewed and paginated. While this is true for the Series 3a, previewing is not available on the Series 3.

The `PRINTER` class is the main interface to an application. It acts as a high level manager, being also a repository of useful information such as the printer model number, port characteristics and so on.

The actual process of printing is delegated to the `PAGES active` class which, amongst other duties, handles pagination, builds the command sequences specific to individual printers and schedules the printing process. `PAGES` itself uses the services of other `FORM` and `OLIB` classes to achieve this behaviour.

Information about specific printer models is held in a WDR printer resource file which can be access by an instance of the `WDR` class. A `PRINTER` object always contains a `WDR` component object.

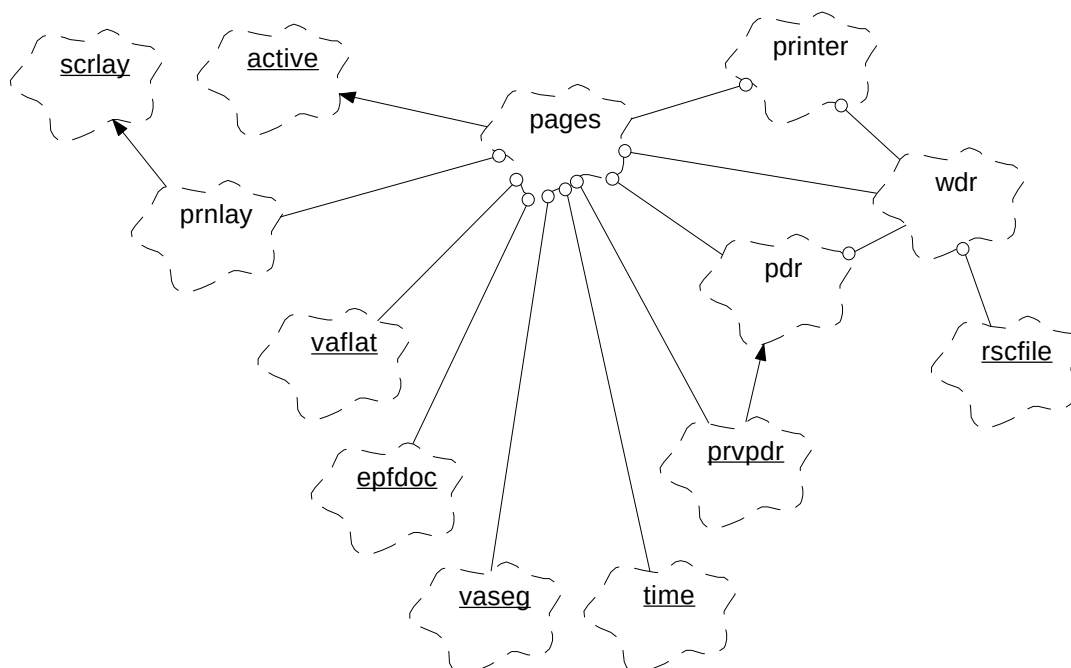
Precursors

An understanding of the document printing classes will be helped by a knowledge of:

- the description of WDR printing and *.wdr* files in the *WDR Printing* chapter of the *Additional System Information* manual
- the *Printing* chapter of the *Object Oriented Programming Guide*
- the *Document Layout Classes* chapter of this manual
- the *Formatted Document Content Classes* chapter of this manual
- the `OLIB active` object class, `ACTIVE`
- the `OLIB` variable array classes, `VASEG` and `VAFLAT`
- the `p_enter` and `p_leave` error handling services

Class diagram

The following diagram covers the relationships between the classes involved in document printing and are discussed in detail in this chapter. The underlined classes are either discussed in another chapter of this manual or they refer to OLIB classes in which case they are *all* discussed in the *OLIB Reference* manual.



Measurement units

In addition to the standard inches and centimetres, this chapter will often refer to other measurement units which are in common use. These are as follows:

- point - defined as 1/72 of an inch; this gives 72 points per inch.
- twip - defined as 1/1440th of an inch; this gives 1440 twips per inch and, therefore, 20 twips per point.
- printer units - defined as the minimum distance of travel in both horizontal and vertical directions, normally defined in twips. The units are dependent on the printer model. See the description of the `wdr` class for more detail.

PRINTER

PRINTER	
wdr	defbottxt
a	p
port_type	prv
deftoptxt	
destroy	pr_get_hd
pr_init	pr_open_wdr
pr_store_srchar	pr_close_wdr
pr_store_file	pr_open_port
pr_set_port_type	pr_print
pr_set_model	pr_paginate
pr_port_data	pr_preview_start
pr_sense_port	pr_preview
pr_sense_model	pr_preview_end
pr_get_params	pr_preview_data
pr_set_hd	

The print manager class provides an interface to an application for printing, paginating and print preview operations; it acts as a repository of information required to successfully execute an operation and provides the methods for setting and sensing this information.

Note that print previewing is *not* available on the Series 3. All references to previewing apply to the Series 3a only.

The class also supplies methods to launch a printing, paginating or previewing operation.

An instance of the `PRINTER` class can be used for a single operation and then be destroyed, or it can be used for multiple operations. A second printing, paginating or previewing operation may be configured in a completely different fashion from the first.

The printing, previewing or paginating process itself is delegated to an instance of the `PAGES` active class; on completion of the process, the `PAGES` class destroys itself.

The `PRINTER` class also provides default values for page dimensions and the positioning of header and footer text which are set up at initialisation time. This information is required by the `PAGES` class. `PRINTER`, however, provides *no* methods to change these values; if they are not suitable, then the `PRINTER` class must be subclassed to provide the required behaviour.

The following configuration parameters can be set and sensed:

- The text of headers and footers.
- The type of port to which printing is to be directed, (i.e. serial or parallel) or whether printing is to be directed to a file or to fax. On the Series 3, printing cannot be directed to fax.
- The characteristics of the serial port.
- The name of the file, if printing is directed to a file.
- The printer resource filename (i.e. the *WDR* resource file name) and the model number of the printer to be used for the next printing operation.

Class definition

The `PRINTER` class subclasses `ROOT` and is defined in the sub-category file *printer.cl* (with generated header file *printer.g*).

```

CLASS printer root
{
  REPLACE destroy          Free alloc cells
  ADD pr_init              Set default values
  ADD pr_store_srchar      Store the serial characteristics
  ADD pr_store_file        Store the spec of the print file
  ADD pr_set_port_type     Set/store printer port type
  ADD pr_set_model         Set/store wdr file & model number
  ADD pr_port_data         Sense printer port data
  ADD pr_sense_port        Sense printer port data of current port type
  ADD pr_sense_model       Sense wdr file & model number
  ADD pr_get_params        Get address of params struct for read/write
  ADD pr_set_hd            Set top or bottom header text
  ADD pr_get_hd            Return address of top or bottom header text
  ADD pr_open_wdr          Create and init wdr object
  ADD pr_close_wdr         Destroy wdr component (to save memory)
  ADD pr_open_port         Open print port device
  ADD pr_print             Print data source
  ADD pr_paginate          Paginate data source
  ADD pr_preview_start     Start preview (i.e. allocate resources)
  ADD pr_preview           Preview data source
  ADD pr_preview_end       End preview (i.e. destroy any resources)
  ADD pr_preview_data      Return pointer to preview data

```

CONSTANTS

```
{
PRINTER_PORT_PARALLEL    0
PRINTER_PORT_SERIAL      1
PRINTER_PORT_FILE        2
PRINTER_PORT_FAX         3
PRINTER_PORT_NOT_SET     255

PRINTER_HDR_TOP          0
PRINTER_HDR_BOT          1

PRINTER_PAGINATE         0
PRINTER_PRINTING         1
PRINTER_PREVIEW          2

PRV_SEG_GRANULARITY      16 paragraphs (256)(power of 2 is ideal)
}
```

TYPES

```
{
typedef struct
{
    TEXT *model;           WDR filename and model number
    TEXT *hdtxt[2];        Header text
} PRINTER_ALLOC;

typedef struct
{
    SCRLAY_FONT f;         Font data for body area text
    UBYTE size_choice;     Paper size index (A4 is zero)
    UBYTE wo_control;      TRUE to disable widows and orphans control
    UWORD spare[2];        Might be useful in future
} PRINTER_DATA;

typedef struct
{
    PAGES_PARAMS p;        Parameters for init of pages object
    PRINTER_DATA d;        Used externally
} PRINTER_PARAMS;

typedef struct
{
    UBYTE *pSegName;
    PR_ROOT *pArray;
    UPOINT PrvSize;
    UWORD BitWidth;        used width of bitmap, use this for scaling
    VOID *hPrvDone;        Call-back handle for %done & completion
    WORD mPrvDone;         Call-back method for %done & completion
} PREVIEW_INIT;

typedef struct
{
    HANDLE SegHandle;      Handle of preview data segment
    PR_ROOT *pArray;
    UPOINT Size;           Size of bitmap (pixels)
    UWORD BitWidth;        used width of bitmap, use this for scaling
    VOID *hPrvDone;        Call-back handle for %done & completion
    WORD mPrvDone;         Call-back method for %done & completion
} PREVIEW_DATA;
}
```

PROPERTY 1

```
{
PR_WDR *wdr;              Printer driver
PRINTER_ALLOC a;          Addresses of allocated cells
WORD port_type;           Port type
TEXT deftoptxt[3];        Default header text (top)
TEXT defbottxt[3];        Default header text (bottom)
PRINTER_PARAMS p;
PREVIEW_DATA prv;
}
```

}

Property

<code>printer.wdr</code>	The handle of the current printer resource object, i.e. the handle of an instance of WDR. The WDR object is created by the <code>pr_open_wdr</code> method.						
<code>printer.a</code>	A data structure of type <code>PRINTER_ALLOC</code> containing three <code>TEXT</code> pointers. Each pointer can contain the address of a single cell of allocated storage as follows: <table><tr><td><code>printer.a.model</code></td><td>if not <code>NULL</code>, points to an allocated cell containing the model number and the filename of the printer resource (i.e. the WDR name) as a zero terminated string. The first byte of the string holds the model number as a numeric character while subsequent bytes hold the zero terminated filename.</td></tr><tr><td><code>printer.a.hdtxt[0]</code></td><td>if not <code>NULL</code>, points to an allocated cell containing the top header text to be used.</td></tr><tr><td><code>printer.a.hdtxt[1]</code></td><td>if not <code>NULL</code>, points to an allocated cell containing the bottom header text to be used.</td></tr></table>	<code>printer.a.model</code>	if not <code>NULL</code> , points to an allocated cell containing the model number and the filename of the printer resource (i.e. the WDR name) as a zero terminated string. The first byte of the string holds the model number as a numeric character while subsequent bytes hold the zero terminated filename.	<code>printer.a.hdtxt[0]</code>	if not <code>NULL</code> , points to an allocated cell containing the top header text to be used.	<code>printer.a.hdtxt[1]</code>	if not <code>NULL</code> , points to an allocated cell containing the bottom header text to be used.
<code>printer.a.model</code>	if not <code>NULL</code> , points to an allocated cell containing the model number and the filename of the printer resource (i.e. the WDR name) as a zero terminated string. The first byte of the string holds the model number as a numeric character while subsequent bytes hold the zero terminated filename.						
<code>printer.a.hdtxt[0]</code>	if not <code>NULL</code> , points to an allocated cell containing the top header text to be used.						
<code>printer.a.hdtxt[1]</code>	if not <code>NULL</code> , points to an allocated cell containing the bottom header text to be used.						
<code>printer.port_type</code>	This contains a value which indicates whether printer output is to be directed to the serial port, the parallel port, fax or to a file. It can take one of the values: <code>PRINTER_PORT_PARALLEL</code> <code>PRINTER_PORT_SERIAL</code> <code>PRINTER_PORT_FAX</code> <code>PRINTER_PORT_FILE</code>						
<code>printer.deftoptxt</code>	The default <i>top</i> header text. This property contains the zero terminated string "%F" and is the text used if top header text has not been explicitly set using the <code>pr_set_hd</code> method.						
<code>printer.defbottxt</code>	The default <i>bottom</i> header text. This property contains the zero terminated string "%P" and is the text used if bottom header text has not been explicitly set using the <code>pr_set_hd</code> method.						
<code>printer.p</code>	This is configuration information consisting of items which can be set and sensed and items such as page dimensions which cannot be altered.						
<code>printer.prv</code>	Preview parameters. These are set by the <code>PR_PREVIEW_START</code> method and are re-set by the <code>PR_PREVIEW_END</code> method.						

Note that the four methods `pr_preview_start`, `pr_preview`, `pr_preview_end` and `pr_preview_data` and the property `printer.prv` are *not* available on the Series 3

Environment variables

The print manager makes use of a number of environment variables in which to store some of its information. When setting configuration information, the environment variables will be created if they do not already exist. If, when sensing configuration information, the appropriate variable does not exist, default values will be returned.

The variable names and their meaning are as follows:

- P\$D** - a zero terminated character string containing the type of port; the type is held as a numeric character.
- P\$F** - a zero terminated character string containing the name of the file to which printing is to be directed. This file is referred to as the *print file*.
- P\$S** - the characteristics of the serial port held as a `P_SRCHAR` data structure.
- P\$M** - a zero terminated character string containing the printer model number as a numeric character followed by the WDR file name (i.e. the printer resource file name).

PRINTER methods

DESTROY

Destroy the print manager

```
VOID destroy(VOID);
```

Destroy the instance of the print manager.

A copy of the address of the instance of the print manager is normally kept in a spare property of the application manager, `appman.spare1`; this method resets this property to `NULL`.

The property `printer.a` is a `PRINTER_ALLOC` data structure which contains three data members each of which *may* contain the handle of an allocated cell. If storage has been allocated, it is freed.

The method supersends a `DESTROY` message to complete the destruction process.

PR_INIT

Initialise printer & set default values

```
VOID pr_init(VOID);
```

Initialise the print manager and set default values for the page dimensions and the positioning of header and footer text.

A copy of the address of this instance of the print manager is set into a spare property of the application manager, `appman.spare1`. This is done for convenience and gives other objects (such as instances of `PDR`) quick and easy access to this instance of `PRINTER`.

Default values are also set for the printer port type and both the header and footer text.

On the Series 3, there is a slight difference to the behaviour of this method. See the Series 3a/Series 3 notes section at the end of this chapter for detail.

PR_STORE_SRCHAR

Set serial port characteristics

```
VOID pr_store_srchar(P_SRCHAR *psc);
```

Set the characteristics of the serial port.

The parameter `psc` points to a data structure of type `P_SRCHAR` which defines the characteristics of the serial port. The method copies the entire content of the data structure pointed to by `psc` into the environment variable `P$S`. The environment variable is created if it does not exist.

The structure `P_SRCHAR` is defined in `p_serial.h` but is shown below for completeness:

```
typedef struct
{
    UBYTE tbaud; /* transmit Baud rate selector */
    UBYTE rbaud; /* receive Baud rate selector */
    UBYTE frame; /* number of data, parity and stop bits */
    UBYTE parity; /* parity selector */
    UBYTE hand; /* handshake flags */
    UBYTE xon; /* XON character */
    UBYTE xoff; /* XOFF character */
    UBYTE flags; /* ignore parity errors/dont drive DTR changing chars */
    ULONG tmask; /* terminator mask */
} P_SRCHAR;
```

PR_STORE_FILE

Set print file specification

```
VOID pr_store_file(TEXT *file);
```

Set the file specification of the file to which printing is to be directed. This file is often referred to as the *print file*.

The parameter `file` points to a character string which contains the (zero terminated) specification of the file to which printing is to be directed. The method copies this string into the environment variable `P$F` which is created if it does not already exist.

If printing is directed to a file and the file specification is not set, then `M:\P.LIS` will be assumed as default.

PR_SET_PORT_TYPE**Set type of printer port**

```
VOID pr_set_port_type(INT store, INT port_type);
```

Set the type of the printer port.

The value passed in the parameter `port_type` specifies the type of the printer port. This can be *one* of the following values:

```
PRINTER_PORT_PARALLEL
```

```
PRINTER_PORT_SERIAL
```

```
PRINTER_PORT_FILE
```

```
PRINTER_PORT_FAX
```

The printer port type is set into *either* the property `printer.port_type` *or* the environment variable `P$D`, not in both. If the parameter `store` contains the value `TRUE`, the port type is set into the environment variable otherwise it is set into the property.

Whichever location is chosen, the printer port type is held as a numeric character.

PR_SET_MODEL**Set printer model no. & WDR file name**

```
VOID pr_set_model(INT store, TEXT *name, INT mnum);
```

Set the printer resource file name (i.e. the *WDR* resource file name) and the printer model number.

The parameter `name` should point to a zero terminated string containing the printer resource file name; the value in the parameter `mnum` should contain the printer model number.

A character string is generated such that the first byte contains the model number as a numeric character; the zero terminated printer resource file name occupies the rest of the string.

A cell of sufficient length is allocated to hold this string and its handle is stored in the property `printer.a.model`.

If the value of the parameter `store` is `TRUE`, the string is also copied into the environment variable `P$M`.

PR_PORT_DATA**Sense printer port information**

```
INT pr_port_data(TEXT *file, P_SRCHAR *ser, INT UseModel);
```

Retrieve information about the printer port.

The method returns the printer port type and retrieves the serial port characteristics and the name of the print file.

The parameter `ser` should point to a data structure of type `P_SRCHAR`. If the environment variable `P$S` can be read, the serial port characteristics are copied from that variable into this `P_SRCHAR` data structure; otherwise, default values are inserted. The default values are as follows:

```
ser->tbaud    P_BAUD_9600
ser->rbaud    P_BAUD_9600
ser->frame    P_DATA_8
ser->parity   0
ser->hand     P_OBEY_XOFF | P_OBEY_DSR | P_IGN_CTS
ser->xoff     0x13
ser->xon      0x11
ser->flags    0
ser->tmask    0
```

For more information on the serial port, see the *Serial Port* chapter of the *I/O Devices Reference* manual.

The parameter `file` should point to a text buffer. If the environment variable `P$F` exists, the name of the print file is copied from that variable into the buffer; otherwise, the default print file name `P.LIS` is copied instead.

The printer port type returned depends on a number of factors and is determined as follows:

- if the parameter `UseModel` contains the value `TRUE` and the first three characters of the fully parsed printer resource file name (i.e. the *WDR* resource file name) are "FAX", the method returns the printer port type `PRINTER_PORT_FAX`.
- if the property `printer.port_type` contains a valid printer port type, then this value is returned.
- if the environment variable `P$D` contains a value, this value is returned.
- if the printer port type cannot be determined by any of the above, a value of `PRINTER_PORT_PARALLEL` is assumed by default.

The effect of the `UseModel` parameter can be neatly summarised:

- if it has the value `FALSE`, the printer port information retrieved is that set by the user of the `PRINTER` object (in an earlier call to the `pr_set_port_type` method).
- if it has the value `TRUE`, it allows the method to return printer port information other than that set by the user. The best example of this is where the printer resource file name begins with the three characters `F A X`. In this case, regardless of the value set in the property `printer.port_type` or the environment variable `P$D`, the printer port type returned is `PRINTER_PORT_FAX`.

On the Series 3, there is a slight difference to the behaviour of this method. See the Series 3a/Series 3 notes section at the end of this chapter for detail.

PR_SENSE_PORT Sense current printer port device information

```
INT pr_sense_port(TEXT *buf, P_SRCHAR *ser);
```

Retrieve information about the current printer port device.

The method returns the printer port type and retrieves the (zero terminated) name of the current printer port device; if the printer port type is `PRINTER_PORT_SERIAL`, the method retrieves the serial port characteristics.

The parameter `buf` should point to a buffer into which the zero terminated name of the current printer port device will be placed. The name written to `*buf` depends on the port type as follows:

<code>PRINTER_PORT_FILE</code>	- the name of the file
<code>PRINTER_PORT_SERIAL</code>	- the serial device name
<code>PRINTER_PORT_PARALLEL</code>	- the parallel device name

On the Series 3 and Series 3a, which only have one port, the serial and parallel device names will always be `TTY:A` and `PAR:A` respectively.

On machines with more than one port, such as the *Workabout*, the port letter will be read from the relevant environment variable, if it exists:

- The serial port letter will be read from the first character of the environment variable `P$SP`, if it exists. Thus, if the first character in this environment variable is 'B' the serial device name will be set to `TTY:B`. If the environment variable does not exist, the serial device name defaults to `TTY:A`.
- The parallel port letter will be read from the first character of the environment variable `P$PP`, if it exists. Thus, if the first character in this environment variable is 'C' the parallel device name will be set to `PAR:C`. If the environment variable does not exist, the parallel device name defaults to `PAR:A`.

These environment variables are not created by system code. It is an application's responsibility to create them if they are needed and do not already exist.

The parameter `ser` should point to a data structure of type `P_SRCHAR`. If the printer port type is `PRINTER_PORT_SERIAL`, a copy of the serial port characteristics will be written to `*ser`.

PR_SENSE_MODEL Sense printer model no. & WDR file name

```
INT pr_sense_model(TEXT *buf);
```

Retrieve the fully parsed printer resource file name (i.e. the *WDR* resource file name) and return the model number.

The parameter *buf* should point to a buffer into which the method can insert the zero terminated printer resource file name. The buffer itself must be capable of holding a fully parsed file name and, therefore, must be at least `P_FNAME_SIZE` bytes long.

The model number and the printer resource file name are held as a character string in an allocated cell pointed to by the property `printer.a.model` and/or in the environment variable `P$M`. The model number itself is held in the form of a numeric character and precedes the file name in the string.

This information is retrieved from the property, if it exists, otherwise it is retrieved from the environment variable. By default, if the information exists in neither location, a value of 0 is returned for the model number and a printer resource file name of `ROM: :BJ.WDR` is written to **buf*.

A check is made to ensure that the printer resource file exists. If the file cannot be found, all of the `LOC: :` drives are searched. If this search is unsuccessful, the `ROM: :` is searched. If, finally, the file has still not been found, then the printer resource file name of `ROM: :BJ.WDR` is assumed together with a model number of zero.

On the Series 3, there is a slight difference to the behaviour of this method. See the Series 3a/Series 3 notes section at the end of this chapter for detail.

PR_GET_PARAMS Fetch address of printer parameters

```
PRINTER_PARAMS *pr_get_params(VOID);
```

Return the address of the `PRINTER` object property `printer.p`.

This property is a data structure of type `PRINTER_PARAMS` and contains information required by the `PAGES` object at its initialisation. As discussed earlier, this is printer configuration information, page dimension information and so on.

See the class definition for more detailed information on `PRINTER_PARAMS`. It may also be useful to refer to the `PAGES` class definition.

PR_SET_HD Set top or bottom header text

```
VOID pr_set_hd(INT htype, TEXT *str);
```

Set *either* the top *or* the bottom header text.

The parameter *str* should be either `NULL` or point to a buffer containing the zero terminated text to be set. The value of the parameter *htype* indicates whether the text is to be set for the top or the bottom header.

Both top and bottom header text are held in cells of allocated storage; the handles of both cells are held in `printer.a.hdtxt[0]` and `printer.a.hdtxt[1]` respectively.

If *htype* contains the value `PRINTER_HDR_TOP`, any existing top header text is discarded by freeing the existing cell. If *str* is *not* `NULL`, a new cell is allocated large enough to contain the new text and its handle is stored in the property `printer.a.hdtxt[0]`; the text is copied into the new cell from **str*. If *str* is `NULL`, any existing top header text is discarded and `printer.a.hdtxt[0]` is set to `NULL`. The effect will be to cause the *default* top header text, as found in the property `printer.deftoptxt`, to be used.

Similarly, if *htype* contains the value `PRINTER_HDR_BOT`, any existing bottom header text is discarded by freeing the existing cell. If *str* is *not* `NULL`, a new cell is allocated large enough to contain the new text and its handle is stored in the property `printer.a.hdtxt[1]`; the text is copied into the new cell from **str*. If *str* is `NULL`, any existing bottom header text is discarded and `printer.a.hdtxt[1]` is set to `NULL`. The effect will be to cause the *default* bottom header text, as found in the property `printer.defbottxt`, to be used.

PR_GET_HD **Get address of top or bottom header text**

```
TEXT *pr_get_hd(INT htype);
```

Return the current address of *either* the top *or* the bottom header text.

The value of the parameter `htype` determines whether the address returned is that of the top or the bottom header text; a value of `PRINTER_HDR_TOP` causes the address of the top header text to be returned while a value of `PRINTER_HDR_BOT` causes the address of the bottom header text to be returned.

If the top header text has been set, the address of the cell containing this text is returned, otherwise the address of the default text is returned. This also applies to the bottom header text.

A note of caution - if the method `pr_set_hd` is called *after* a call to `pr_get_hd`, the address of any header text may be invalid.

PR_OPEN_WDR **Create WDR object**

```
PR_WDR *pr_open_wdr(VOID);
```

Create and initialise a new WDR (printer resource) object and return its handle.

Any existing WDR object is destroyed by calling the `pr_close_wdr` method before attempting to create the new one.

The new WDR object is initialised by sending it a `WDR_INIT` message and passing it both the current model number and the printer resource file name. The `pr_sense_model` method is used to determine the current model number and the printer resource file name.

The handle of the new WDR object is set into the property `printer.wdr`.

The model number set in the WDR object may be changed at any time by sending the object a `WDR_SET_MODEL` message; however, if a different printer resource file name is needed, the existing WDR object *must* be destroyed and a new one created, passing it the new file name.

Note that this method is called by the `pr_print` and `pr_paginate` methods.

PR_CLOSE_WDR **Destroy WDR object**

```
VOID pr_close_wdr(VOID);
```

Destroy the current printer resource (WDR) object, if it exists.

The WDR object is destroyed by sending it a `DESTROY` message. The property `printer.wdr` containing the handle of the object is re-set to `NULL`.

PR_OPEN_PORT **Open printer port device**

```
VOID *pr_open_port(VOID);
```

Open the printer port device.

The name of the printer port device, the printer port type and the serial characteristics (if the port type is `PRINTER_PORT_SERIAL`) are retrieved by calling the `pr_sense_port` method.

The printer port device is opened by calling the Plib function `f_open`, passing it the retrieved device name.

If the port type is `PRINTER_PORT_SERIAL` then `p_iow` is called to *set* the serial port characteristics.

The method returns the handle of the opened port (i.e. the address of the channel control block).

PR_PRINT**Print data source**

```
PR_PAGES *pr_print(PAGES_CALLS *pc);
```

Launch the printing of the current data source and return the handle of the `PAGES` object created to do the printing.

The parameter `pc` must point to a data structure of type `PAGES_CALLS`. The caller of this method must set the call-back methods needed to handle *read* and *done* messages, into the `PAGES_CALLS` data structure. See the description of the `PAGES` class and the `PAGELAY` notional mixin class in this chapter for more information.

`PAGES_CALLS` can be found in the `PAGES` class definition but is shown below for completeness:

```
typedef struct
{
    VOID *hread;           Call-back handle for reading a line
    WORD mread;            Call-back method for reading a line
    VOID *hdone;           Call-back handle for %done & completion
    WORD mdone;            Call-back method for %done & completion
} PAGES_CALLS;
```

If the printer resource (`WDR`) object does not exist, it is created by calling the `pr_open_wdr` method.

The method creates a `PAGES` object to do the printing and initialises it by sending an `AO_INIT` message.

The `PAGES` object requires information which consists of a copy of `PRINTER`'s property `printer.p.p` (a `PAGES_PARAMS` data structure), a fully completed `PAGES_INIT` data structure and an indication that it is to perform a printing operation.

The information in the `PAGES_INIT` data structure includes a copy of the `PAGES_CALLS` data structure whose address is given in the parameter `pc`, the handle of the `WDR` object, the addresses of both the top and bottom header text and the address of the buffer containing the name of the current data source.

The `PAGES` object destroys itself on completion of printing or if an error occurs.

On the Series 3, there is a slight difference to the behaviour of this method. See the Series 3a/Series 3 notes section at the end of this chapter for detail.

PR_PAGINATE**Paginate data source**

```
PR_PAGES *pr_paginate(PAGES_CALLS *pc);
```

Launch the pagination of the current data source and return the handle of the `PAGES` object created to do the pagination.

The parameter `pc` must point to a data structure of type `PAGES_CALLS`. The caller of this method must set the call-back methods needed to handle *read* and *done* messages, into the `PAGES_CALLS` data structure. See the description of the `PAGES` class and the `PAGELAY` notional mixin class in this chapter for more information.

`PAGES_CALLS` can be found in the `PAGES` class definition (or see the description of `pr_print` earlier).

If the printer resource (`WDR`) object does not exist, it is created by calling the `pr_open_wdr` method.

The method creates a `PAGES` object to do the pagination and initialises it by sending an `AO_INIT` message.

The `PAGES` object requires information which consists of a copy of `PRINTER`'s property `printer.p.p` (a `PAGES_PARAMS` data structure), a *partially* completed `PAGES_INIT` data structure and an indication that it is to perform a pagination operation.

The information required in the `PAGES_INIT` data structure includes a copy of the `PAGES_CALLS` data structure and the handle of the `WDR` object.

The `PAGES` object destroys itself on completion of pagination or if an error occurs.

PR_PREVIEW_START**Initialise preview**

```
HANDLE pr_preview_start (PREVIEW_INIT *pInit);
```

Perform initialisation for previewing a data source.

The parameter `pInit` should point to a data structure of type `PREVIEW_INIT` containing the information required by preview. This structure, shown below, is defined in the class definition:

```
typedef struct
{
    UBYTE    *pSegName;
    PR_ROOT  *pArray;
    UPOINT    PrvSize;
    UWORD    Bitwidth;    used width of bitmap, use this for scaling
    VOID      *hPrvDone;  Call-back handle for %done & completion
    WORD      mPrvDone;   Call-back method for %done & completion
} PREVIEW_INIT;
```

The members have the following meaning:

pSegName	The address of a buffer containing the name to be given to the external data segment.
pArray	The handle of a VAFLAT array object which will be used to hold a series of values giving the position of consecutive compressed bitmaps within the preview data segment. It is designed to hold entries (records) which are the length of a LONG 'C' data type and has a granularity of 16 entries (records).
PrvSize	The dimensions of the bitmap, in <i>pixels</i>, to which a page will be drawn. The x component is the value of <i>Bitwidth</i> <i>rounded</i> up so that it occupies an integral number of bytes. The y component is normally determined by the height of the application's window.
Bitwidth	The number of horizontal bits needed to draw a single line so that it fits into the application's window <i>and</i> the ratio of this value to the height of the page measured in pixels will be the same as the ratio of the width to the height of the page measured in twips. Note that this value will not necessarily be exactly the same as that in <i>PrvSize.x</i> above. This item will be used by the PRVPDR class to calculate a rounding factor for its internal calculations
hPrvDone	The handle of the object which will provide the <i>done</i> callback method.
mPrvDone	The method number of the <i>done</i> callback method. For the general specification of this method, see the <code>pagelay_mdone</code> method in the description of the PAGELAY mixin class in this chapter.

The exact method for calculating the values of `PrvSize` and `Bitwidth` depends on the application. The following sequence shows how a typical application might proceed. However, it should only be used as a guideline:

- Determine the space *available* to display a previewed page; in other words, determine the number of horizontal and vertical pixels available and set *PrvSize.y* to the number of vertical pixels.
- If in landscape mode, calculate *Bitwidth* to be the result of:
page height (in twips) / page width (in twips) * vertical pixels available.
- If in portait mode, calculate *Bitwidth* to be the result of:
page width (in twips) / page height (in twips) * vertical pixels available.

- Check that the resulting value of `Bitwidth` will fit into the number of horizontal pixels available. If it does not fit, re-set `Bitwidth` to the available number of horizontal pixels and adjust the value of `PrvSize.y` to maintain proportions:
 if in landscape mode, set `PrvSize.y` to:
 $\text{page width (in twips)} / \text{page height (in twips)} * \text{horizontal pixels available}$
 if in portrait mode, set `PrvSize.y` to:
 $\text{page height (in twips)} / \text{page width (in twips)} * \text{horizontal pixels available}$
- Set `PrvSize.x` to the value of `Bitwidth` rounded up to a multiple of eight, in other words, ensure that the number of bits represented will fit into an integral number of bytes.

The method creates a dynamic external data segment; the name to be applied to the segment is supplied in a buffer pointed to by `pInit->pSegName`. Recall that an external data segment is one that is not constrained to the 64K limit but can extend to 512k, provided sufficient memory is available.

Although the segment is initially created with a zero length, `p_leave` is called if the allocation fails.

The handle of the successfully created external segment is set into the `SegHandle` member of the property `printer.prv`.

The size of the external segment is adjusted to `PRV_SEG_GRANULARITY` paragraphs. If the adjustment fails (e.g. `E_GEN_NOMEMORY`), the method `pr_preview_end` is called to free any acquired resources and is followed by a call to `p_leave`.

All the remaining initialisation information supplied by `*pInit` is copied into the property `printer.prv`.

It is worth noting that an instance of `PRVPDR` (the print preview class) requests the address of the `printer.prv` property during its initialisation by sending a `PR_PREVIEW_DATA` message to this instance of `PRINTER`. The `pr_preview_data` method is described later.

The method returns the handle of the dynamic external data segment.

PR_PREVIEW

Perform preview

```
PR_PAGES *pr_preview (PAGES_CALLS *pc);
```

Launch the preview process of the current data source and return the handle of the `PAGES` object created to do the previewing.

The parameter `pc` must point to a data structure of type `PAGES_CALLS`. The caller of this method must set the call-back methods needed to handle *read* and *done* messages, into the `PAGES_CALLS` data structure. See the description of the `PAGES` class and the `PAGELAY` notional mixin class in this chapter for more information.

`PAGES_CALLS` can be found in the `PAGES` class definition (or see the description of `pr_print` earlier).

If the printer driver (`WDR`) object does not exist, it is created by calling the `pr_open_wdr` method.

The method creates a `PAGES` object to perform the previewing and initialises it by sending an `AO_INIT` message.

The `PAGES` object requires information which consists of a copy of `PRINTER`'s property `printer.p.p` (a `PAGES_PARAMS` data structure), a fully completed `PAGES_INIT` data structure and an indication that it is to perform a previewing operation.

The information in the `PAGES_INIT` data structure includes a copy of the `PAGES_CALLS` data structure whose address is given in the parameter `pc`, the handle of the `WDR` object, the addresses of both the top and bottom header text and the address of the buffer containing the name of the current data source.

The `PAGES` object destroys itself on completion of the previewing operation or if an error occurs.

PR_PREVIEW_END

Terminate preview

```
VOID pr_preview_end (VOID);
```

Terminate preview and free any acquired resources.

The method frees the allocated external data segment using the `Plib` function `p_sgclose` and then indicates the end of preview by resetting the whole of property `printer.prv` to binary zero.

PR_PREVIEW_DATA**Return handle of preview data**

```
PREVIEW_DATA *pr_preview_data (VOID);
```

Return the handle of the preview data.

This method returns the address of the `PRINTER` object's `printer.prv` property which contains information required by the preview operation. The content of `printer.prv` is set by the `pr_preview_start` method.

PAGES

ACTIVE	PAGES
q priority isactive pcb stat	pagarr page todo reset_page ephead newdoc prhead started pdr newpage in region par holding last_pos pos flags pr pheight tabs brk_height spacing brk_above margins brk_pos time y date nrec ioclen
destroy ~~ao_init~~ ao_cancel ~~ao_abrun~~ ~~ao_queue~~ ~~ao_run~~	ao_init ao_run ao_queue ao_abrun

A `PAGES` object handles a request to print, paginate or preview one or more documents on behalf of an instance of the `PRINTER` class and is created when the application sends a `PR_PRINT`, `PR_PAGINATE` or `PR_PREVIEW` message to the `PRINTER` object. The `pr_print`, `pr_paginate` and `pr_preview` methods create a `PAGES` object as part of their implementation.

`PAGES` itself is a low priority active object; this allows the printing, previewing or pagination process to be done in discrete chunks, giving other active objects (and, therefore, other applications) the opportunity to run concurrently.

In general, knowledge of the document to be printed, in terms of the text itself, the typefaces, fonts (i.e. height) and styles to be used, lies within other object(s) in the application. In order to print a document, the `PAGES` object must ask the application for the next portion of text to be printed; this is normally represented by a data structure of type `WDR_PRINT`, often referred to as a *print element*. The `PAGES` object must also keep the application informed of the current status of the printing operation. `PAGES` achieves this by means of *callback* methods.

`PAGES` itself uses the services of an instance of the `PDR` (printer driver) class to translate a print element into a sequence of commands suitable for a specific printer. When performing a preview operation, the `PDR` object represents a specialised printer driver.

When paginating, `PAGES` does no printing, instead it uses the print elements to build pagination information.

Page breaks and the printing of headers and footers are done automatically by `PAGES`.

Class definition

The PAGES class subclasses the OLIB class ACTIVE and is defined in the sub-category file *pages.cl* (with generated header file *pages.g*).

```

CLASS pages active
{
  REPLACE ao_init
  REPLACE ao_run
  REPLACE ao_abrun
  REPLACE ao_queue

  CONSTANTS
  {
    PAGES_DONE_PAGE      0
    PAGES_DONE_DOC       1
    PAGES_DONE_END       2
    PAGES_DONE_ERROR     3

    PAGES_DOC_NEW_PAGE    0x01    A new doc starts a new page
    PAGES_DOC_RESET_PAGE_NUM 0x02    A new doc resets the page number

    PAGES_FLAGS_PRINTING  0x01    Printing (or previewing)
    PAGES_FLAGS_NOTFIRST  0x02    Discard down when FALSE (on top of 1st page)
    PAGES_FLAGS_ZERODOWN  0x04    Discard 1st down in todo list if TRUE

    PAGES_REGION_BODY     0
    PAGES_REGION_TOP      1
    PAGES_REGION_BOTTOM   2
    PAGES_REGION_LAST_BOTTOM 3
    PAGES_REGION_END      4
    PAGES_REGION_VERY_END 5
    PAGES_REGION_DOC_END  6

    PAGES_PAGENUM_ARABIC  0
    PAGES_PAGENUM_ROMAN_U 1
    PAGES_PAGENUM_ROMAN_L 2
    PAGES_HEADER_2_COLUMNS SCRLAY_ALIGN_JUSTIFY+1
    PAGES_HEADER_3_COLUMNS SCRLAY_ALIGN_JUSTIFY+2
  }

  TYPES
  {
    typedef struct
    {
      UWORD x;
      UWORD y;
    } UPOINT;

    typedef struct
    {
      UPOINT tl;
      UWORD width;
      UWORD height;
    } UEXTENT;

    typedef struct
    {
      WORD event;          PAGES_DONE_PAGE, _DOC, _END
      UWORD page;          Page number of new page
      PR_VAFLAT *pages;    Page length array
    } PAGES_DONE;

    typedef struct
    {
      SCRLAY_FONT f;       Font data
      UBYTE align;         Header alignment
      UBYTE first_page;    TRUE to emit on first page
    } PAGES_HEADER;
  }
}

```

```
typedef struct
{
    UWORD width;      Width of page
    UWORD height;     Height of page
    UEXTENT body;     Body area
    UWORD hdtop;      body.tl.y-hdtop = Y of top of top header
    UWORD hdbot;      body.tl.y+body.height+hdbot=height = Y of top
                     of bottom header
} PAGES_PAGE;

typedef struct
{
    WORD offset;      Offset for page number
    WORD last;        Last page number for %m
    WORD style;       Style for page
} PAGES_PAGENUM;

typedef struct
{
    PAGES_PAGE pg;    Page dimensions with margins
    WORD pdrflags;    WDR_PDR_LANDSCAPE, _DRAFT etc
    WORD docflags;    PAGES_DOC_NEW_PAGE is set for new page per doc
    UWORD pgbeg;      Page number to start printing (1st page is 1)
    UWORD pgend;      Last page number to print (inclusive)
    PAGES_HEADER top; Top running header
    PAGES_HEADER bot; Bottom running header
    PAGES_PAGENUM pgnum; Page number for %p
} PAGES_PARAMS;

typedef struct
{
    VOID *hread;      Call-back handle for reading a line
    WORD mread;       Call-back method for reading a line
    VOID *hdone;      Call-back handle for %done & completion
    WORD mdone;       Call-back method for %done & completion
} PAGES_CALLS;

typedef struct
{
    PR_WDR *wdr;      Printer driver
    PAGES_CALLS c;    Call-backs
    TEXT *fname;      File name for %f (ZTS) or NULL
    TEXT *toptxt;     Top running header ZTS or NULL
    TEXT *bottxt;     Bottom running header ZTS or NULL
} PAGES_INIT;
}
```

```

PROPERTY 5
{
    PR_VAFLAT *pagarr;    Page break array being built
    PR_VASEG *todo;      Pending print output
    PR_EPFDOC *ephead;   Text for top/bottom header
    PR_PRNLAY *prhead;   Layout for top/bottom header
    PR_PDR *pdr;         Output printer driver
    PAGES_INIT in;       Init parameters
    PAGES_PARAMS par;    User settable init parameters
    WORD last_pos;       Pos of last page break
    UWORD flags;         Printing or paginating 1st page etc
    WORD pheight;        Height of processed lines on current page
    WORD brk_height;     Height of last line break
    WORD brk_above;      At last line break
    UWORD brk_pos;       Document pos at last page break
    WORD y;              Current page y
    WORD nrec;           Record number in todo list
    UWORD page;          Page Number
    UBYTE reset_page;    Reset page number on next footer if TRUE
    UBYTE newdoc;        TRUE if there is another doc
    UBYTE started;       TRUE if printing has really started
    UBYTE newpage;       TRUE if pagination decided on a new page
    WORD region;         PAGES_REGION_TOP, _BODY or _BOTTOM
    WORD holding;        TRUE if holding data in todo list
    UWORD pos;           Document pos
    WDR_PRINT pr;        Being printed
    SCRLAY_TABS tabs;    Header tab settings
    SCRLAY_SPACING spacing; Header spacing
    SCRLAY_MARGINS margins; Header margins
    TEXT time[LN_TIME_DATE_STR];
    TEXT date[LN_TIME_TIME_STR-2];
    UWORD ioclen;
}
}

```

Property

<code>pages.pagarr</code>	The <i>handle</i> of an instance of the VAFLAT class. The array of UWORD elements contains pagination information where each entry contains a count of the number of characters fitting into a page. The entries are in page order.
<code>pages.todo</code>	The <i>handle</i> of an instance of the VASEG class. The array of WDR_PRINT data structures contains a queue of print elements to be dealt with. Queuing print elements is a convenient way of deferring printing. This is important when lines which must be <i>kept together</i> are being processed; in these circumstances it is not possible to know in advance whether a page break will occur after the lines are printed or whether a page break will need to be forced <i>before</i> the first line is printed (with the consequent need to print bottom and top header text first). N.B. This queue is also referred to as the <i>todo list</i>.
<code>pages.ephead</code>	The <i>handle</i> of an instance of the EPFDOC class, used to encapsulate the text of a top or bottom header.
<code>pages.prhead</code>	The <i>handle</i> of an instance of the PRNLAY class, used to encapsulate the layout of a top or bottom header.
<code>pages.pdr</code>	The handle of the printer driver object. This is an instance of the PDR class and is created by the <code>ao_init</code> method.

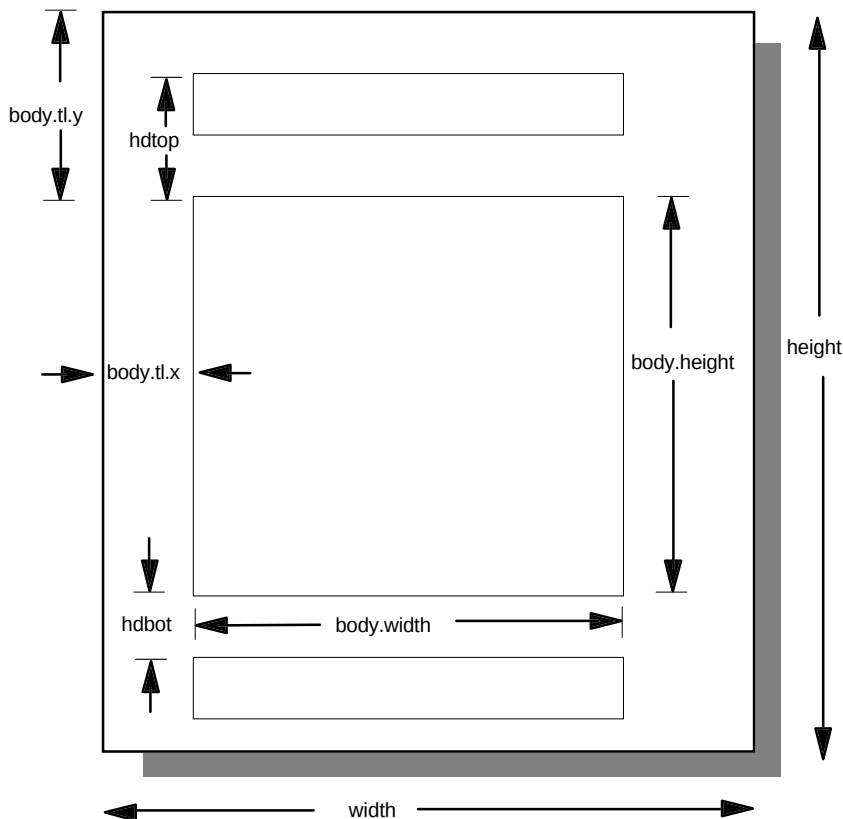
<code>pages.in</code>	Initialisation information, derived from the application, and passed to the <code>PAGES</code> object at initialisation time in a call to its <code>ao_init</code> method; this includes: • the handle of the printer resource (<code>WDR</code>) object • the call-back information • when printing and previewing, the addresses of buffers containing the file name of the current data source, the top header text and the bottom header text.						
<code>pages.par</code>	Initialisation information, set up internally by the <code>PRINTER</code> object, and passed to the <code>PAGES</code> object at initialisation time in a call to its <code>ao_init</code> method. See the description of the <code>ao_init</code> method.						
<code>pages.last_pos</code>	The document position of the previous page break. This is used during pagination; in particular, during the process of generating entries for the <code>pages</code> array whose handle is contained in the property <code>pages.pagarr</code>.						
<code>pages.flags</code>	This is a general flag area; the following values can be <code>ORed</code> into this property in combination: <table><tr><td><code>PAGES_FLAGS_PRINTING</code></td><td>If set, <code>PAGES</code> has been created to perform a printing or previewing operation; if <i>not</i> set, <code>PAGES</code> has been created to perform a paginating operation.</td></tr><tr><td><code>PAGES_FLAGS_ZERODOWN</code></td><td>This flag is only set when a page break occurs. <i>If set</i>, any spacing <i>above</i> the first line of the <i>next</i> page is suppressed (by setting <code>pages.pr.down</code> to zero). At the top of a page, any spacing above a line is redundant.</td></tr><tr><td><code>PAGES_FLAGS_NOTFIRST</code></td><td>This flag is set after the <i>first</i> line on the <i>first</i> page of the <i>first</i> document has been processed. Once set, it remains set for the life of the <code>PAGES</code> object. <i>If not set:</i> • any page break before the first page of the first document is suppressed. • any spacing <i>above</i> the <i>first</i> line of the <i>first</i> page of the <i>first</i> document is suppressed (by setting <code>pages.pr.down</code> to zero). At the top of a page, any spacing above a line is redundant. </td></tr></table>	<code>PAGES_FLAGS_PRINTING</code>	If set, <code>PAGES</code> has been created to perform a printing or previewing operation; if <i>not</i> set, <code>PAGES</code> has been created to perform a paginating operation.	<code>PAGES_FLAGS_ZERODOWN</code>	This flag is only set when a page break occurs. <i>If set</i> , any spacing <i>above</i> the first line of the <i>next</i> page is suppressed (by setting <code>pages.pr.down</code> to zero). At the top of a page, any spacing above a line is redundant.	<code>PAGES_FLAGS_NOTFIRST</code>	This flag is set after the <i>first</i> line on the <i>first</i> page of the <i>first</i> document has been processed. Once set, it remains set for the life of the <code>PAGES</code> object. <i>If not set:</i> • any page break before the first page of the first document is suppressed. • any spacing <i>above</i> the <i>first</i> line of the <i>first</i> page of the <i>first</i> document is suppressed (by setting <code>pages.pr.down</code> to zero). At the top of a page, any spacing above a line is redundant.
<code>PAGES_FLAGS_PRINTING</code>	If set, <code>PAGES</code> has been created to perform a printing or previewing operation; if <i>not</i> set, <code>PAGES</code> has been created to perform a paginating operation.						
<code>PAGES_FLAGS_ZERODOWN</code>	This flag is only set when a page break occurs. <i>If set</i> , any spacing <i>above</i> the first line of the <i>next</i> page is suppressed (by setting <code>pages.pr.down</code> to zero). At the top of a page, any spacing above a line is redundant.						
<code>PAGES_FLAGS_NOTFIRST</code>	This flag is set after the <i>first</i> line on the <i>first</i> page of the <i>first</i> document has been processed. Once set, it remains set for the life of the <code>PAGES</code> object. <i>If not set:</i> • any page break before the first page of the first document is suppressed. • any spacing <i>above</i> the <i>first</i> line of the <i>first</i> page of the <i>first</i> document is suppressed (by setting <code>pages.pr.down</code> to zero). At the top of a page, any spacing above a line is redundant.						
<code>pages.pheight</code>	This is used during pagination. It is the cumulative height, in printer units, of the lines on the current page which have already been processed. In effect, it measures the current height of the current page (from the top of the page).						
<code>pages.brk_height</code>	This is used during pagination. It is a <i>candidate</i> page break position, measured in printer units. In effect, it measures the height of the candidate page (from the top of the page to the break point). The value in this property represents a potential (but legal) page break position. It is important when lines of text <i>which must be kept together</i> are being processed. Until all of the lines in such a group have been processed, it cannot be known in advance whether a page break can occur after the group or whether it must be forced before this group. In the latter case, this property will become the actual height of the page.						

<code>pages.brk_above</code>	This is used during pagination. It has a function which is closely related to that of the property <code>pages.brk_height</code> discussed above. It is the space above the first line element following the candidate page break position. The space above a line element is often referred to as the <i>down</i> value; this is used to define a gap between consecutive lines such as occurs between the last and first lines of consecutive paragraphs. Following on from the discussion of <code>pages.brk_height</code> above; where a group of lines must be kept together and are forced onto a new page, the <i>down</i> value of the first line will be set to zero, to avoid unnecessary blank space appearing at the top of the page, when printed. The cumulative page height for the new page is adjusted to take this change into account.														
<code>pages.brk_pos</code>	The position within the document corresponding to the previous page break.														
<code>pages.y</code>	The current <i>vertical</i> print position on the current page, measured in printer units. The position is measured relative to the top of the page.														
<code>pages.nrec</code>	The current record number in the todo list; i.e. the current entry in the array of <code>WDR_PRINT</code> records, anchored in the property <code>pages.todo</code> .														
<code>pages.page</code>	The current page number.														
<code>pages.reset_page</code>	This property can take the value <code>TRUE</code> or <code>FALSE</code> . It is set to <code>TRUE</code> if the page number on the next footer is to be reset (to zero).														
<code>pages.newdoc</code>	This property can take the value <code>TRUE</code> or <code>FALSE</code> . It is set to <code>TRUE</code> if, on completion of processing a document, another document is to be processed. Set by the <code>ao_run</code> method but the decision is taken by the <code>PAGES</code> done callback method.														
<code>pages.started</code>	This property can take the value <code>TRUE</code> or <code>FALSE</code> . It is set to <code>TRUE</code> if printing has started.														
<code>pages.newpage</code>	This property can take the value <code>TRUE</code> or <code>FALSE</code> . It is set to <code>TRUE</code> if the pagination process has decided to begin a new page.														
<code>pages.region</code>	This property contains a flag to record the current <i>context</i>; it is set to one of the following <i>mutually exclusive</i> values: <table><tr><td><code>PAGES_REGION_BODY</code></td><td>if set, the main body of a document is currently being handled.</td></tr><tr><td><code>PAGES_REGION_TOP</code></td><td>if set, the top header text of a document is currently being handled.</td></tr><tr><td><code>PAGES_REGION_BOTTOM</code></td><td>if set, the bottom header text of a document is currently being handled.</td></tr><tr><td><code>PAGES_REGION_LAST_BOTTOM</code></td><td>if set, the last bottom header text of the <i>final</i> document is currently being handled.</td></tr><tr><td><code>PAGES_REGION_DOC_END</code></td><td>if set, the <i>current</i> document is exhausted</td></tr><tr><td><code>PAGES_REGION_END</code></td><td>if set, the <i>final</i> document is exhausted</td></tr><tr><td><code>PAGES_REGION_VERY_END</code></td><td>if set, all documents are exhausted and the printing process has been terminated. It indicates that the <code>PAGES</code> object is ready to destroy itself.</td></tr></table>	<code>PAGES_REGION_BODY</code>	if set, the main body of a document is currently being handled.	<code>PAGES_REGION_TOP</code>	if set, the top header text of a document is currently being handled.	<code>PAGES_REGION_BOTTOM</code>	if set, the bottom header text of a document is currently being handled.	<code>PAGES_REGION_LAST_BOTTOM</code>	if set, the last bottom header text of the <i>final</i> document is currently being handled.	<code>PAGES_REGION_DOC_END</code>	if set, the <i>current</i> document is exhausted	<code>PAGES_REGION_END</code>	if set, the <i>final</i> document is exhausted	<code>PAGES_REGION_VERY_END</code>	if set, all documents are exhausted and the printing process has been terminated. It indicates that the <code>PAGES</code> object is ready to destroy itself.
<code>PAGES_REGION_BODY</code>	if set, the main body of a document is currently being handled.														
<code>PAGES_REGION_TOP</code>	if set, the top header text of a document is currently being handled.														
<code>PAGES_REGION_BOTTOM</code>	if set, the bottom header text of a document is currently being handled.														
<code>PAGES_REGION_LAST_BOTTOM</code>	if set, the last bottom header text of the <i>final</i> document is currently being handled.														
<code>PAGES_REGION_DOC_END</code>	if set, the <i>current</i> document is exhausted														
<code>PAGES_REGION_END</code>	if set, the <i>final</i> document is exhausted														
<code>PAGES_REGION_VERY_END</code>	if set, all documents are exhausted and the printing process has been terminated. It indicates that the <code>PAGES</code> object is ready to destroy itself.														

<code>pages.holding</code>	This property can take the value <code>TRUE</code> or <code>FALSE</code>. When set to <code>TRUE</code>, the print element to be processed (in <code>ao_run</code>) is placed on the queue implemented by the <code>VASEG</code> array object whose handle is in <code>pages.todo</code>. The property is set to <code>TRUE</code> if a print element must go onto a <i>new</i> line and, at the same time, must be kept on the <i>same</i> page as the following print element.
<code>pages.pos</code>	The current position within the document being handled.
<code>pages.pr</code>	The current print element. The information is contained in a data structure of type <code>WDR_PRINT</code> .
<code>pages.tabs</code>	Tab settings for the top and bottom headers, passed to the header <code>PRNLAY</code> layout object when created.
<code>pages.spacing</code>	Spacing information for the top and bottom headers, passed to the header <code>PRNLAY</code> layout object when created.
<code>pages.margins</code>	Margins for the top and bottom headers, passed to the header <code>PRNLAY</code> layout object when created.
<code>pages.time</code>	A character string containing the current time (set by <code>ao_init</code>).
<code>pages.date</code>	A character string containing the current date (set by <code>ao_init</code>).
<code>pages.ioclen</code>	Length of the command buffer sent to the printer (set by <code>ao_queue</code>).

Page dimensions

The following diagram illustrates the meaning of the various members of a `PAGES_PAGE` structure in relation to the components of a typical document. The outer rectangle represents the page while the inner rectangles show the positions of the header text, the main body of the document and the footer text respectively.



PAGES methods

AO_INIT

Initialise and queue

```
INT ao_init(PAGES_INIT *in, INT printing, PAGES_PARAMS *par);
```

Initialise the PAGES active object and begin the print/paginate/preview operation by sending an AO_QUEUE message.

The method is called by an instance of the PRINTER class as part of the implementation of that class's PR_PRINT, PR_PAGINATE and PR_PREVIEW methods. Three parameters, containing the information needed to build this instance of PAGES, are required.

The parameter `printing` indicates whether the PAGES active object represents a printing, pagination or preview operation. It can take one of three possible values:

PRINTER_PRINTING	perform a printing operation
PRINTER_PAGINATE	perform a pagination operation
PRINTER_PREVIEW	perform a preview operation

The parameters `in` and `par` contain initialisation information and point to data structures of type `PAGES_INIT` and `PAGES_PARAMS` respectively. The two data structures reflect the different origins of the information contained within them.

In general terms, *some* of the information contained within the `PAGES_PARAMS` data structure is set up internally by the PRINTER object itself as default values, whereas the information contained within the `PAGES_INIT` data structure is ultimately derived from the application.

In some respects, this division is artificial. However, the design does minimise the effort required of an application if the defaults are adequate. A more sophisticated application would need to sub-class PRINTER and either replace the `pr_init` method or add new methods to modify these default values.

The default values supplied by the PRINTER object in the parameter `par` are as follows:

<code>pg.width</code>	<code>PAGE_WIDTH_A4</code>
<code>pg.height</code>	<code>PAGE_LENGTH_A4</code>
<code>pg.body.tl.x</code>	1800
<code>pg.body.tl.y</code>	1800
<code>pg.body.width</code>	<code>PAGE_WIDTH_A4</code> - 3600
<code>pg.body.height</code>	<code>PAGE_LENGTH_A4</code> - 3600
<code>pg.hdtop</code>	720
<code>pg.hdbot</code>	720
<code>top.f.height</code>	240
<code>bot.f.height</code>	240
<code>bot.align</code>	<code>SCRALY_ALIGN_CENTRE</code>
<code>pgbeg</code>	1
<code>pgend</code>	0xFFFF

where `PAGE_WIDTH_A4` and `PAGE_LENGTH_A4` are both defined in *pagesize.h* and `SCRALY_ALIGN_CENTRE` can be found in the `SCRALY` class definition (see *The Document Layout Classes* chapter). All other members of this `PAGES_PARAMS` structure and its sub-structures are uninitialised.

The complete content of `*par` is copied into the property `pages.par`.

The information contained in the parameter `in` is as follows:

<code>wdr</code>	The handle of the printer resource (WDR) object.
<code>c</code>	The callback method numbers and the handles of their "owning" objects.
<code>fname</code>	The address of a buffer containing the file name of the source. This member is only set if <code>PAGES</code> is to perform a printing or previewing operation. It is <i>uninitialised</i> for a pagination operation.
<code>toptxt</code>	The address of a buffer containing the top header text. This member is only set if <code>PAGES</code> is to perform a printing or previewing operation. It is <i>uninitialised</i> for a pagination operation.
<code>bottxt</code>	The address of a buffer containing the bottom header text. This member is only set if <code>PAGES</code> is to perform a printing or previewing operation. It is <i>uninitialised</i> for a pagination operation.

The complete content of `*in` is copied into the property `pages.in`.

The callback information referred to above, is itself contained within a data structure of type `PAGES_CALLS` and is set up by the application and passed to the `PRINTER` object before being handed on, in turn, to this instance of `PAGES`. This data structure, while defined in the `PAGES` class definition, is shown below:

```
typedef struct
{
    VOID *hread;
    WORD mread;
    VOID *hdone;
    WORD mdone;
} PAGES_CALLS;
```

As mentioned in the introductory text, `PAGES` must have a mechanism for requesting the next print element from the application and for keeping the application informed of the current status of the printing operation. This is achieved by means of *callback* methods.

The creator of the `PAGES` object must supply the method numbers of two call-back methods together with the handles of their corresponding objects. The call-back methods are referred to as the *read* and the *done* call-back methods, expressions which will be used in this chapter.

The *read* call-back method provides the means by which the `PAGES` object can get the next print element from the application. The method number is supplied in `pages.in.c.mread` and the handle of the corresponding object is supplied in `pages.in.c.hread`.

The *done* call-back method provides the mechanism by which the application can be told about the status of the printing operation; for example, the completion of a page or the completion of a document. This allows the application to take appropriate action. The method number is supplied in `pages.in.c.mdone` and the handle of the corresponding object is supplied in `pages.in.c.hdone`.

Although the `FORM` class `PRNLAY` supplies a *read* call-back method, in general, the design of the classes which supply these two methods is very much application dependent. However, the methods themselves *must* conform to the general specification described in the `PAGELAY` notional mixin class in this chapter. In practice, `PRNLAY` is subclassed.

The horizontal and vertical page dimensions held in the `pages.par` property are converted from twips to horizontal and vertical printer units respectively by calling the `wdr_twips_to_xy` method of the printer resource (WDR) object.

If the `PAGES` active object is intended to perform pagination, the method creates and prepares a `VAFAT` object in which to build an array of `UWORD` elements to contain the number of characters in each page. The handle of the `VAFAT` object is set into the `pages.pagarr` property.

If the `PAGES` active object is intended to perform either a printing or a preview operation, the method fetches the current time and date, using an instance of the `OLIB` class `TIME`, and stores their character representations in the properties `pages.time` and `pages.date` respectively. An instance of `VASEG` is created and initialised so that each record in the array will be large enough to contain a print element (i.e. a `WDR_PRINT` structure); the handle of this object is set into the `pages.todo` property. Also, the `PAGES_FLAGS_PRINTING` flag is set in the property `pages.flags`.

For a preview operation, a preview printer driver object (an instance of `PRVPDR`) is created and initialised directly by sending it a `PDR_INIT` message. It is important to note that the `PRVPDR` class is a subclass of `PDR` but does *not* reside in a separate `dyl`. The information in the `PDR_INIT` *data structure* required by the `pdr_init` method, is extracted from both `PAGES` itself and the associated printer resource (`WDR`) object. Note that the member `dyl` is set to zero. For more information on previewing, see *The Print Preview Class* chapter in this manual.

For a printing operation, a `WDR_OPEN_PRINT` message is sent to the associated printer resource (`WDR`) object which, as part of its behaviour, creates a printer driver (`PDR`) object. In either case, the handle of the created printer driver object is set into the `pages.pdr` property.

For a printing or a preview operation, a document object (an instance of `EPFDOC`) and a corresponding printer layout object (an instance of `PRNLAY`) are created for the top header and their handles stored in the properties `pages.ephead` and `pages.prhead` respectively.

The priority of the `PAGES` active object is set to `PRIORITY_ACTIVE_COMPUTE` and the application manager is sent an `AM_ADD_TASK` message to add the active object to the application manager's active object task queue.

The print/paginate/preview operation is started by sending an `AO_QUEUE` message and the method terminates by returning zero.

On the Series 3, there is a slight difference to the behaviour of, and the interface to, this method. See the Series 3a/Series 3 notes section at the end of this chapter for detail.

AO_RUN

Fetch next print element

```
INT ao_run(VOID);
```

The method is called by the application manager when an I/O operation to the printer has completed *or* the `PAGES` active object has been re-scheduled by a call to its superclass `ao_queue` method.

In *general terms*, this method fetches the next print element to be processed as represented by a data structure of type `WDR_PRINT` and further:

- if printing or previewing, the `ao_queue` method is called to translate the information in this print element into a set of printer specific commands and to start the sequence of I/O operation(s) to the printer using the Plib function `p_ioc`.
- if paginating, the information in the print element is used in the construction of the page array object anchored in `pages.pagarr`. This is followed by a call to the superclass `ao_queue` method to schedule the next call to this (i.e. the `ao_run`) method.

When printing or previewing, there are circumstances where a print element cannot be processed immediately but must be placed in a queue, known as a *todo* list - an instance of the `VASEG` class. This is done to handle what are commonly known as *widows* and *orphans*. In a situation where a group of lines must be kept together, the position of page breaks cannot be determined until all such lines have been fetched in.

Print elements are commonly fetched for processing by calling the *read* call-back method. However, in handling widows and orphans as mentioned above, a print element may, instead, be fetched from an existing *todo* list; alternatively, a print element may not be processed immediately but will be placed on the *todo* list - depending on the general context.

When a page break occurs, the method will call the *done* call-back method with a completion status of `PAGES_DONE_PAGE`; similarly, when the processing of a document is complete, the *done* call-back method will be called with a completion status of `PAGES_DONE_DOC`.

If the `PAGES_REGION_VERY_END` flag is set in the property `pages.region`, all documents will have been processed and the printing process itself will have been terminated. The *done* call-back method is called with a completion status of `PAGES_DONE_END`. The `PAGES` object destroys itself by sending itself a `DESTROY` message.

AO_ABRUN**Handle error**

```
VOID ao_abrun(VOID);
```

This method supports APPMAN's architecture for handling `p_leave` within the `ao_run` method. `p_leave` may be called if a write operation has not succeeded.

The *done* call-back method is called with a completion status of `PAGES_DONE_ERROR` to notify the application that an error has occurred. This allows the application to take appropriate action.

The method then destroys this instance of `PAGES` by sending a `DESTROY` message.

AO_QUEUE**Translate the print element and print data**

```
VOID ao_queue(VOID);
```

If `PAGES` is performing a paginating operation, the method simply calls the superclass `ao_queue` method to re-schedule a call to `ao_run`.

If `PAGES` is performing a printing or previewing operation, the method sends a `PDR_PRINT` message to the associated `PDR` object.

For a printing operation, the `pdr_print` method translates the current print element into a sequence of printer commands. The printer commands are then sent to the printer device by doing a `P_FWRITE` I/O operation to the printer port. If the `PDR` object produces *no printer* commands, nothing is sent to the printer; instead, the method simply calls the superclass `ao_queue` method to re-schedule a call to `ao_run`.

For a previewing operation, the `pdr_print` method translates the current print element into a sequence of drawing actions to a bitmap. As this does not require any I/O activity, this method (i.e. `ao_queue`) simply follows on by calling the superclass `ao_queue` method to re-schedule a call to `ao_run`.

Before the `PDR_PRINT` message is sent, either for printing or previewing, some preliminary processing is done:

- if the *final* document to be printed or previewed is exhausted, the `WDR_PRINT_END` flag is Ored into the `flags` field of the print element and the `WDR_PRINT_PAGE` flag is cleared. This will cause the printing/previewing process to terminate.
- if the current page is not within the range of pages to be printed or previewed (as defined by `pages.par.pgbeg` and `pages.par.pgend`), the print element is ignored, the `PDR_PRINT` message is not sent and the method simply calls its superclass's `ao_queue` method to re-schedule a call to `ao_run`.
- if printing or previewing has not yet started, the `WDR_PRINT_START` flag is Ored into the `flags` field of the print element. This will cause the printer to be initialised and set up correctly or the preview bitmap to be cleared.
- if a page break is to be forced, the current vertical print position is adjusted appropriately.
- if a line break is to be forced, the line element indent is adjusted appropriately.

WDR

WDR
file head model wid wdrname
destroy wdr_init wdr_count_models wdr_sense_model_name wdr_set_model wdr_sense_model wdr_typeface wdr_search_typeface wdr_font_height wdr_search_height wdr_get_width_table wdr_sense_width wdr_twips_to_xy wdr_open_print wdr_load_record

The WDR printer resource class, encapsulates the handling of a *WDR* resource file.

A *WDR* resource file contains printer specific information organised as a series of resources. For example, for each printer model supported, a resource file exists containing the various command sequences to control that printer.

A WDR object also provides methods to supply specific information from a *WDR* resource file.

The *WDR Printing* chapter in the *Additional System Information* manual gives a full and comprehensive description of the structure of *WDR* resource files. Further useful information on printing can be found in the *Printing* chapter of the *Object Oriented Programming Guide*.

A typical WDR file has resources containing the following information:

- a list of printer models supported
- the character command sequences to control printer operation
- a map used to translate the printer's character set onto that used by the SIBO computer (based on the IBM code page 850)
- a list of typefaces supported by each model.
- a list of fonts available in each typeface.
- a width table for each font, where the widths *in the WDR file* are stored in difference form (see *Widths of characters in fonts* in the *WDR Printing* chapter of the *Additional System Information* manual).

An instance of WDR is normally created by a PRINTER object in its `pr_open_wdr` method. The WDR class itself supports:

- The accessing of the contents of a *WDR* resource file.

The WDR class may be used to read the contents of the *WDR* file. It does this by using an RSCFILE class component. The WDR class stores the current model and the list of available typefaces in memory. Other information is read from the file as and when required to minimise memory requirements.

- The conversion of measurements from units of twips to horizontal and vertical printer units.

The *WDR* resource file includes the minimum horizontal and vertical travel for each printer model. These distances are specified in units of twips. The `wdr_twips_to_xy` method is provided to convert horizontal and vertical distances from twips to printer units.

- The creation and initialisation of a suitable printer driver object - usually an instance of either the *PDR* class or a suitable subclass of *PDR*.

A suitable printer driver object may be created and initialised using the `wdr_open_print` method; the method returns the handle of the object which is usually an instance of the *PDR* class or a suitable subclass. See the description of the `wdr_open_print` method for further details.

An instance of *WDR* can be created which is limited to reading the list of models supported. In this case the class does not store the details of the current model thus minimising memory requirements. In this mode only the `wdr_count_models`, `wdr_sense_model_name`, `wdr_sense_width`, `wdr_set_model` and `wdr_destroy` methods are available.

A loaded font width table contains a sequence of unsigned bytes containing information about the width of each character in a font. Two kinds of width table are available:

- a monospace font width table - contains two bytes, the first of which contains zero, and the second of which specifies the width of a character. (By definition each character in a monospace font has the same width.) The zero in the first byte signifies a monospace font width table.
- a proportional font width table - contains 256 bytes, the first of which contains one; the remaining 255 bytes specify the width of the characters whose code ranges from 0x00 to 0xFF. Thus the fiftieth byte specifies the width of the character whose code is 0x31.

Note that, on being loaded from the *WDR* file, width tables are converted from differences to absolute character widths.

Class definition

The *WDR* class subclasses *ROOT* and is defined in the sub-category file *prdrv.cl* (with generated header file *prdrv.g*).

```
CLASS wdr root
{
  REPLACE destroy
  ADD wdr_init           Init and optionally load model data
  ADD wdr_count_models   Return the number of models
  ADD wdr_sense_model_name Return model name by model number
  ADD wdr_set_model       Set the current model number
  ADD wdr_sense_model     Sense struct of current model
  ADD wdr_typeface        Get typeface struct by typeface index
  ADD wdr_search_typeface Get typeface struct and index given typeface
  ADD wdr_font_height     Font height by typeface index, height index
  ADD wdr_search_height   Get height index given height
  ADD wdr_get_width_table Width table given typeface, height and style
  ADD wdr_sense_width     Get printed width of buf, len
  ADD wdr_twips_to_xy     Convert from twips to printer units
  ADD wdr_open_print      Create a PDR for printer output
  ADD wdr_load_record     Load specified resource record
```

CONSTANTS

```

{
WDR_PRINT_PAGE      0x01
WDR_PRINT_LINE      0x02
WDR_PRINT_RIGHT     0x04
WDR_PRINT_FONT      0x08
WDR_PRINT_TEXT      0x10
WDR_PRINT_START     0x20
WDR_PRINT_END       0x40
WDR_PRINT_IDLE      0x4000    ! Used by pages
WDR_PRINT_KEEP      0x8000    ! Used by pages

WDR_PDR_LANDSCAPE   0x01

WDR_RSC_HEADER      1        Header resource ID
WDR_RSC_COMMANDS    2        Commands resource ID

WDR_DYL_LOAD        0x01     .WDR requires a DYL if set
WDR_HP_PCL           0x02     printer driver is HP PCL compatible

WDR_STYLE_NORMAL    0x0000
WDR_STYLE_UNDERLINE 0x0001
WDR_STYLE_BOLD      0x0002
WDR_STYLE_ITALIC    0x0004
WDR_STYLE_SUPER     0x0008
WDR_STYLE_SUB       0x0010
WDR_STYLE_MONOSPACE 0x8000    Reserved for external use
WDR_STYLE_SANS_SERIF 0x4000    Reserved for external use

WDR_TYPF_PROPORTIONAL 0x01
WDR_TYPF_SCALED       0x02
WDR_TYPF_SERIF       0x04

WDR_MODEL_LANDSCAPE_AVAILABLE 1
WDR_MODEL_MINX_IS_DOTS_PER_INCH 4

WDR_SCALE_DEFAULT_HEIGHT 1000    reference height = 1000 twips = 50 point

WDR_FONT_NAME_LEN    20    significant characters from font name
PRINTER_NAME_LEN     24    maximum length of printer name
PRINT_TYPE_LEN       9     printer type length (filename+'\0')
PDR_FILE_LEN         9     printer driver filename length
}

```

TYPES

```

{
typedef struct
{
    UWORD height;           Height (or min height for scalable fonts)
    UWORD height_max;       Max height (only relevant for scalable fonts)
    UWORD height_delta;     Delta height (only relevant for scalable fonts)
    UWORD width_scale;      Multiplier for font width table
    UWORD width_normal;     Width for mono, rid for proportional
    UWORD width_italic;
    UWORD width_bold;
    UWORD width_bold_italic;
    UWORD command;          Set font command number
} WDR_FONT;

typedef struct
{
    TEXT name[WDR_FONT_NAME_LEN];    Typeface name
    UWORD typeface;                   RTF/Word compatible typeface
    UWORD type;                       WDR_TYPF_PROPORTIONAL
                                      WDR_TYPF_SCALED
    WORD trans_rid;                   rid of translates record
    UWORD num_heights;                Number of different typeface heights
    WDR_FONT font[1];                 List of different heights
} WDR_TYPEFACE;
}

```

```
typedef struct
{
    UWORD minx;           minimum delta x
                          (in twips, unless MINX_IS_DPI flag set)
    UWORD miny;           minimum delta y in twips
    UWORD skipx;          amount printer auto indents
    UWORD skipy;          amount printer auto feeds
    UWORD flags;          WDR_MODEL_LANDSCAPE_AVAILABLE
                          WDR_MODEL_MINX_IS_DOTS_PER_INCH
    UWORD num_typefaces;  number of typefaces supported by model
    WDR_TYPEFACE *typeface[1]; list of typeface rids/pointers to typeface data
} WDR_MODEL;

typedef struct
{
    UWORD rid;            rid of model block
    TEXT name[PRINTER_NAME_LEN]; model name
} WDR_MODEL_INDEX;

typedef struct
{
    TEXT id[6];           file identifier
    UWORD flags;          flags (WDR_DYL_LOAD)
    UWORD num_model;      number of models described in file
    WDR_MODEL_INDEX model[1]; list of model names/rid's
} WDR_HEADER;

typedef struct width_table
{
    struct width_table *next;
    UWORD rid;            Resource ID width table (used as a key)
    UWORD height;         Needed if font is scaled
    UBYTE *table;         Address of width table
} WDR_WIDTH_TABLE;

typedef struct
{
    WORD flags;           WDR_PRINT_XXX
    WORD typf;            Typeface number for WDR_PRINT_FONT
    WORD fheight;         Font height for WDR_PRINT_FONT
    WORD style;           Font style for WDR_PRINT_FONT
    WORD down;            Line down for WDR_PRINT_LINE
    WORD indent;          Line indent for WDR_PRINT_LINE
    WORD height;          Line height for WDR_PRINT_LINE
    WORD right;           Right movement for WDR_PRINT_RIGHT
    TEXT *buf;            Text to print for WDR_PRINT_TEXT
    UWORD blen;           Length of data at buf for WDR_PRINT_TEXT
} WDR_PRINT;
}

PROPERTY 1
{
    PR_RSCFILE *file;     Resource file containing driver data
    WDR_HEADER *head;     Header and model index
    WDR_MODEL *model;     The current model
    WDR_WIDTH_TABLE *wid; List of font widths
    TEXT wdrname[P_FNAME_SIZE]; .WDR File name
}
}
```

Property

<code>wdr.file</code>	The handle of an instance of the <code>RSCFILE</code> class which is used to read resources from the <i>WDR</i> resource file.
<code>wdr.head</code>	The address of the <i>WDR</i> header resource, a data structure of type <code>WDR_HEADER</code> . The header resource is fetched from the resource file by the <code>wdr_init</code> method; it achieves this by sending an <code>RS_READ</code> message to the <code>RSCFILE</code> component object.

<code>wdr.model</code>	Information about the current printer model. Amongst other things, it includes the number of typefaces available for the current printer model and the resource ID for each typeface.
<code>wdr.wid</code>	This is a linked list of <code>WDR_WIDTH_TABLE</code> data structures each of which contains: - the resource ID of the font width table. - the height of the font (in the case of a scalable font). - a pointer to the loaded font width table itself.
<code>wdr.wdrname</code>	The full file specification of the <i>WDR</i> resource file.

It is useful to note that the `WDR_PRINT` structure defines the fields of a *print element* as used by the `PAGES` and `PDR` classes and its use is discussed in these classes. The structure is not used by the `WDR` class.

WDR methods

DESTROY

Destroy

```
VOID destroy(VOID);
```

Destroy the `WDR` instance.

The method frees the linked list of `WDR_WIDTH_TABLE` data structures anchored in the property `wdr.wid` and frees the memory occupied by the font width tables.

The memory used to contain the header resource whose address is held in the property `wdr.head`, is freed; `wdr.head` is reset to `NULL`.

All memory cells containing printer model information as anchored in the property `wdr.model`, are freed; `wdr.model` itself is reset to `NULL`.

The method concludes by supersending a `DESTROY` message.

WDR_INIT

Initialise WDR

```
VOID wdr_init(TEXT *filename, INT model);
```

Initialise the `WDR` instance.

This method takes two parameters:

- `filename` points to a buffer containing the filename of the *WDR* resource file.
- `model` contains the printer model. See the *WDR Printing* chapter in the *Additional System Information* manual for more information on the concept of model numbers.

The method builds a full file specification using the name pointed to by `filename`; the default path is used to supply any missing components. The resulting name is written to the property `wdr.wdrname`.

The method creates an instance of the `RSCFILE` class and writes the handle to the property `wdr.file`. The resulting `RSCFILE` object is initialised by sending it an `RS_INIT` message and passing the full file specification of the *WDR* resource file as an argument.

The header resource is loaded by sending the `RSCFILE` object an `RS_READ` message and passing it the resource ID of the header. The address of the loaded resource is written to the property `wdr.head`. The header resource contains a list of models supported. A check is made to ensure that the header is valid. It contains a five character identification field which should always be "WDR05". If this is not so, the method calls `p_leave` with an argument of `E_FILE_INVALID`.

The resource for the specific printer model specified by the parameter `model` is loaded by sending a `WDR_SET_MODEL` message specifying an argument of `model`. The address of the loaded model resource is written to the property `wdr.model`.

If the `WDR` instance is only to be used to sense the models supported then `model` should have the value `-1`; this avoids needless memory allocation by the `wdr_set_model` method.

WDR_COUNT_MODELS**Return the number of models**

```
INT wdr_count_models(VOID);
```

Return the number of printer models supported.

The method simply returns the value of `wdr.head->num_model`.

WDR_SENSE_MODEL_NAME**Get model name from model no.**

```
TEXT *wdr_sense_model_name(INT model);
```

Returns the address of the area containing the name of the printer model corresponding to a specified model number.

This method takes a single parameter; `model` contains the number of the printer model whose name is required.

The method uses the value of `model` as an index to address the appropriate `WDR_MODEL_INDEX` data structure within the header resource; it then simply returns the address of the string containing the model name (i.e. `&self->wdr.head->model[model].name[0]`).

Note that the name will contain no more than `PRINTER_NAME_LEN` characters (defined in *prdrv.g*).

WDR_SET_MODEL**Set the current model**

```
VOID wdr_set_model(INT model);
```

Free allocated memory, load the resource for the printer model specified by the parameter `model` and load the resource for each typeface supported by the specified printer model.

The method frees the linked list of `WDR_WIDTH_TABLE` data structures anchored in the property `wdr.wid` and frees the memory occupied by the font width tables. All memory cells containing printer model information as anchored in the property `wdr.model`, are freed; `wdr.model` itself is reset to `NULL`.

If the parameter `model` contains the value -1, no model resource information is to be read in and the method simply returns.

If the value of the parameter `model` is greater than the number of models supported, the method resets `model` to zero.

The method reads the resource for the specified printer model from the *WDR* resource file by sending a `RS_READ` message to the component `RSCFILE` object, and writes the address of the loaded resource to `wdr.model`.

For each typeface, it loads the corresponding resource and writes the address of the loaded resource to the corresponding element of the `wdr.model->typeface` array.

WDR_SENSE_MODEL**Sense current model data**

```
WDR_MODEL *wdr_sense_model(VOID);
```

Sense the data for the current printer model.

The method simply returns the content of `wdr.model`, i.e. the address of the `WDR_MODEL` data structure containing the current printer model information.

WDR_TYPEFACE**Get typeface by index**

```
WDR_TYPEFACE *wdr_typeface(INT typfix);
```

Return the address of a `WDR_TYPEFACE` data structure.

The method takes a single parameter; `typfix` contains the *index* of an entry within the array of pointers to `WDR_TYPEFACE` data structures for the current printer model. Each `WDR_TYPEFACE` data structure contains typeface information for the current printer model.

The method simply uses the index to return the address of the corresponding typeface entry. Formally, it returns:

```
self->wdr.model->typeface[typfix]
```

WDR_SEARCH_TYPEFACE Get typeface by typeface number

```
INT wdr_search_typeface(INT typf, WORD *ptypfix, WDR_TYPEFACE **ppdata);
```

Search for the typeface, in the printer model information, whose typeface number matches the supplied value.

This method takes three parameters:

- `typf` contains the typeface number corresponding to the typeface being sought. A typeface number uniquely identifies a typeface (Courier, for example) and is compatible with DOS/WORD font numbers. Each typeface resource contains a typeface number as part of its identity; see the `WDR_TYPEFACE` structure.
- `ptypfix` is the address of an area into which this method will write the index of the entry within the array of pointers to `WDR_TYPEFACE` data structures containing the typeface with number `typf`.

This parameter can be `NULL` in which case *no* attempt is made to write the index.

- `ppdata` is the address of an area into which this method will write the *address* of the `WDR_TYPEFACE` data structure containing the typeface with number `typf`.

This parameter can be `NULL` in which case *no* attempt is made to write the address.

If the specified typeface is not present, the method will attempt to search for a substitute typeface. The following table shows how the substitution is done. The left-hand column shows the specified typeface (implied by the typeface number) while the right-hand column shows the corresponding base font that is substituted.

Proportional Serif	->	Times
Proportional Sans Serif	->	Helvetica
Mono	->	Courier (default mono)
Times, Helvetica	->	Courier (default mono)

Note that if the specified typeface is Times or Helvetica, implying that either (or both) of these base fonts is not present, no matching substitute is available and the default monospaced font is used.. For further details, see the *Printer driver font mapping* section of the *Word Processor File Format* chapter of the *Additional System Information* manual.

The method returns

- `TRUE` if the specified typeface was located or a suitable substitution was made.
- `FALSE` if neither the specified typeface nor the substitution was found - in this case, the typeface index is set to zero which corresponds to the *default* typeface.

WDR_FONT_HEIGHT Get font height by typeface & font indexes

```
INT wdr_font_height(INT typfix, INT fhix);
```

Return the height in twips of the font with a given font (i.e. height) index in a typeface with a given typeface index.

This method takes two parameters:

- `typfix` contains the index of an entry within the array of pointers to `WDR_TYPEFACE` structures.
- `fhix` contains the index of an entry within the array of `WDR_FONT` structures in the typeface determined by `typfix` above.

For a non-scalable font, the method simply returns the height of the font. Formally this is the value:

```
self->wdr.model.typeface[typfix]->font[fhix].height
```

For a scalable font, the height is explicitly calculated. `fhix` is used as a scaling factor. The value returned is the result of:

```
self->wdr.model.typeface[typfix]->font[0].height  
  
    plus  
  
self->wdr.model.typeface[typfix]->font[0].height_delta*fhix
```

WDR_SEARCH_HEIGHT Get font index given height

```
INT wdr_search_height(INT typfix, UWORD *pheight);
```

Return the font (i.e. height) index of the font with a specified height (in twips) in a given typeface.

This method takes two parameters:

- `typfix` contains the index of an entry within the array of pointers to `WDR_TYPEFACE` structures.
- `pheight` is the address of an area which contains the height of the font (in twips).

The value returned is the index of an entry within the array of `WDR_FONT` structures in the typeface determined by `typfix` above.

If a font with the exact desired height is not located, the method returns the index of the tallest of all those fonts whose heights are lower than the value in `*pheight`. In this case, `*pheight` is overwritten with the new height.

WDR_GET_WIDTH_TABLE Get a requested font width table

```
UBYTE *wdr_get_width_table(INT typf, INT height, INT style);
```

Return the address of the font width table for the font with a given height in a given typeface in a given style.

The method takes three parameters:

- `typf` contains the typeface number of the typeface being sought.
- `height` specifies the height of the font in twips
- `style` specifies the style. The only combination of styles which are used by this method are:

```
WDR_STYLE_NORMAL  
WDR_STYLE_ITALIC  
WDR_STYLE_BOLD  
WDR_STYLE_ITALIC | WDR_STYLE_BOLD
```

All other styles are ignored. If `style` contains *neither* `WDR_STYLE_ITALIC` *nor* `WDR_STYLE_BOLD` in any combination, then `WDR_STYLE_NORMAL` is assumed by default.

The method starts by sending a `WDR_SEARCH_TYPEFACE` message to find the index and the address of the `WDR_TYPEFACE` data structure of the typeface with number `typf`.

For monospace fonts, the method simply returns a pointer to the font width table.

For proportional fonts, the font width table to be selected depends on the style combinations in the parameter `style`. If the selected table has previously been loaded, the method simply returns the required address; otherwise, a new (uninitialised) `WDR_WIDTH_TABLE` structure is allocated and inserted into the existing queue so that `wdr.wid` points to the new entry and the new entry points to the existing entries. The required font width table resource is loaded into the new entry by sending an `RS_READ` message to the `RSCFILE` component object.

The remaining fields of a new `WDR_WIDTH_TABLE` entry are filled in as follows:-

- `rid` is set to the associated resource ID
- `height` is set to the font height
- the first byte of the table itself is set to 1 to distinguish it from a monospace table

Notes:-

If the requested typeface is unavailable, a substitute typeface will be used.

If a font of the desired height is unavailable, the tallest possible font which is less than the desired font will be substituted.

The method supports scalable proportional fonts but does *not* support scalable monospace fonts.

WDR_SENSE_WIDTH

Get printed width of text

```
INT wdr_sense_width(UBYTE *pwid, TEXT *buf, INT len);
```

Return the printed width of text, in printer units.

The method takes three parameters:

- `pwid` contains the address of the font width table to be used.
- `buf` contains the address of a buffer holding the text whose width is to be found.
- `len` contains the length of the text.

The method simply adds up the width of each character in the buffer pointed to by `buf`, using the width values defined in the font width table.

WDR_TWIPS_TO_XY

Convert twips to printer units

```
VOID wdr_twips_to_xy(WORD **ppx, WORD **ppy);
```

Convert twips to printer units.

The method takes two parameters;

- `ppx` points to a list of addresses, each of which points to a word containing a twips value to be converted into *horizontal* printer units. The list of addresses is terminated by a `NULL`.
- `ppy` points to a list of addresses, each of which points to a word containing a twips value to be converted into *vertical* printer units. The list of addresses is terminated by a `NULL`.

Resulting values are rounded up to the nearest integer which avoids small measurements coming out as zero. To get zero, the caller must explicitly enter zero.

WDR_OPEN_PRINT

Create a PDR for printer output

```
VOID *wdr_open_print(INT flags, INT page_length, VOID **ppcb);
```

Create and initialise an instance of the PDR (printer driver) class or a suitable subclass of PDR, returning the handle of the instance.

The method takes three parameters;

- `flags` specifies the orientation of a page. If `WDR_PDR_LANDSCAPE` is set, then landscape orientation is required, otherwise portrait orientation is implied.
- `page_length` specifies the height of a page in printer units.
- `ppcb` is the address of an area into which the handle of a channel to the opened printer port will be inserted.

If the flag `WDR_DYL_LOAD` is set in `wdr.head->flags`, then it is assumed that an instance of a subclass of PDR is to be created and that this subclass is to be found in a separate DYL. A DYL with the same filename as the WDR resource file but with an extension of `.dyl` is assumed to exist and an attempt is made to load and link to it. If this DYL does not exist or cannot be found, the method will terminate with a `p_leave`. An instance of the *first* class in the DYL is created.

If the flag `WDR_DYL_LOAD` is *not* set in `wdr.head->flags`, then an instance of the basic PDR class as defined in FORM is created.

The created object is initialised and printing is started by sending it a `PDR_INIT` message. See the PDR class for a description of the `pdr_init` method and the information passed to it.

WDR_LOAD_RECORD**Load resource record**

```
VOID wdr_load_record(INT rid,VOID **pcell);
```

Load a resource from the *WDR* resource file.

The method takes two parameters;

- *rid* specifies the resource ID of the resource to be loaded.
- *pcell* is the address of an area into which the *address* of a memory cell containing the loaded resource is placed; i.e. the address of the loaded resource is written to **pcell*.

The resource is loaded by sending an *RS_READ* message to the *RSCFILE* component of *WDR*.

PDR

PDR
par mode typfix fhix style lheight a trans_rid outlen skipy
destroy pdr_init pdr_print pdr_add_command pdr_destroy pdr_start pdr_end pdr_page pdr_text pdr_line pdr_right pdr_font pdr_style

PDR is the printer driver class and is that part of document printing that encapsulates the conversion of print elements (as defined by the content of a *WDR_PRINT* data structure) into a sequence of printer commands.

The printer commands generated are specific to a particular printer; information about the printer model is passed to the *PDR* object at initialisation time.

An instance of *PDR* is normally created by an instance of the printer resource (*WDR*) class but is made a component of a *PAGES* object; in general, there is a degree of dependence on the *WDR* object which is asked to provide further information from time to time. *PDR* can, under some circumstances, be created directly by other suitable classes (e.g. *PAGES*).

Once created, active objects such as *PAGES* use a *PDR* object to build a sequence of printer specific commands on its behalf ; the *PAGES* active object then schedules the transmission of the commands to the printer.

Note that if the *PDR* class is subclassed, the normal usage is to load the subclass from a *DYL*: see the description of the *pdr_init* method for more detail.

A description of the contents of WDR resource files can be found in the *WDR Printing and Resource Files* chapters of the *Additional System Information* manual. Further useful information on printing can be found in the *Printing* chapter of the *Object Oriented Programming Guide*.

Class definition

The PDR class subclasses ROOT and is defined in the sub-category file *prdrv.cl* (with generated header file *prdrv.g*).

```

CLASS pdr root
{
  REPLACE destroy
  ADD pdr_init           Called from WDR
  ADD pdr_print          Called externally by the print active object
  ADD pdr_add_command    Add command to printer buffer
  ADD pdr_destroy=p_dummy For a DYL subclass destroy
  ADD pdr_start          Start printing
  ADD pdr_end            Finish printing
  ADD pdr_page           Start a new page
  ADD pdr_text           Print text at current pos
  ADD pdr_line           Start a new line
  ADD pdr_right          Position to the right
  ADD pdr_font           Set the font
  ADD pdr_style          Set the font style
}

CONSTANTS
{
  PDR_CMD_RESET 0
  PDR_CMD_FORM_LENGTH 1
  PDR_CMD_PREAMBLE 2
  PDR_CMD_POSTAMBLE 3
  PDR_CMD_UNDERLINE_ON 4
  PDR_CMD_UNDERLINE_OFF 5
  PDR_CMD_BOLD_ON 6
  PDR_CMD_BOLD_OFF 7
  PDR_CMD_ITALIC_ON 8
  PDR_CMD_ITALIC_OFF 9
  PDR_CMD_SUPERSCRIPT_ON 10
  PDR_CMD_SUPERSCRIPT_OFF 11
  PDR_CMD_SUBSCRIPT_ON 12
  PDR_CMD_SUBSCRIPT_OFF 13
  PDR_CMD_NEW_PAGE 14
  PDR_CMD_CARRIAGE_RETURN 15
  PDR_CMD_MOVE_DOWN 16
  PDR_CMD_MOVE_RIGHT_PREFIX 17
  PDR_CMD_MOVE_RIGHT 18
  PDR_CMD_MOVE_RIGHT_SUFFIX 19
  PDR_CMD_LANDSCAPE 20
}

TYPES
{
  typedef struct
  {
    PR_WDR *wdr;           Ref back to creating wdr
    WORD flags;            WDR_PDR_LANDSCAPE
    WORD page_length;      Page length for setting form size
    HANDLE dyl;            Handle of loaded DYL
    WDR_HEADER *head;      Header
    WDR_MODEL *model;       Model
  } PDR_INIT;

  typedef struct
  {
    UBYTE *commands;       Command strings
    UBYTE *outbuf;          Print output buffer
    UBYTE *trans_res;       Character set translates resource
    TEXT **tix;             Translates lookup table
  } PDR_ALLOC;
}

```

```
PROPERTY
{
    PDR_INIT par;
    UWORD mode;           Mode (landscape)
    WORD typfix;          Current typeface index
    WORD fhix;            Current font height index
    WORD style;           Current font style
    WORD lheight;         Height of current line
    PDR_ALLOC a;          Various allocated cells
    UWORD trans_rid;      Resource ID of loaded translates
    WORD outlen;          Length of data in output buffer
    UWORD skipy;
}
```

Property

<code>pdr.par</code>	Initialisation data set by the <code>pdr_init</code> method and defined as a data structure of type <code>PDR_INIT</code> . This includes items such as pointers to the associated WDR object and the current printer model information.		
<code>pdr.mode</code>	This property is used to indicate whether printing is to be done in landscape or portrait mode. If the <code>WDR_PDR_LANDSCAPE</code> flag is set, printing is to be done in landscape mode; if the property contains <code>NULL</code> then printing is to be done in portrait mode.		
<code>pdr.typfix</code>	The index of an entry within the array of pointers to <code>WDR_TYPEFACE</code> structures. The <code>WDR_TYPEFACE</code> structure identified contains information on the <i>current typeface</i> . The array is part of the <code>WDR_MODEL</code> data structure representing the current printer model.		
<code>pdr.fhix</code>	The index into the array of <code>WDR_FONT</code> structures which corresponds to the <i>current font height</i> . The array is part of the <code>WDR_TYPEFACE</code> data structure representing the current typeface.		
<code>pdr.style</code>	The current font style. This can be a combination of a number of individual styles represented by an ored combination of the following flags: <pre>WDR_STYLE_NORMAL WDR_STYLE_UNDERLINE WDR_STYLE_BOLD WDR_STYLE_ITALIC WDR_STYLE_SUPER</pre> See the description of the <code>pdr_style</code> method for more detail.		
<code>pdr.lheight</code>	The height of the current line in printer units. This property is set whenever a print element is handled which has <code>WDR_PRINT_LINE</code> set in its <code>flags</code> member and is copied from the print element's <code>height</code> member. In other words, it is set when a new line is forced.		
<code>pdr.a</code>	This is a data structure of type <code>PDR_ALLOC</code> which contains a number of pointers to allocated memory cells. They are grouped together in this structure for convenience. The individual members of this structure are important and are discussed below: <table><tr><td><code>commands</code></td><td> This is a pointer to a table of commands supported by the current printer model(s). The table starts with a byte count giving the number of commands followed by the commands themselves. Each command starts with a byte count giving the total length of the command. The remaining bytes contain a format string consisting of the printer command itself and formatting characters as used by the <code>Plib</code> function <code>p_atob</code>. See the <i>Plib Reference</i> manual and the <i>WDR Printing</i> chapter of the <i>Additional System Information</i> manual for more detail. The table itself is loaded from the WDR resource file during execution of the <code>pdr_init</code> method. </td></tr></table>	<code>commands</code>	This is a pointer to a table of commands supported by the current printer model(s). The table starts with a byte count giving the number of commands followed by the commands themselves. Each command starts with a byte count giving the total length of the command. The remaining bytes contain a format string consisting of the printer command itself and formatting characters as used by the <code>Plib</code> function <code>p_atob</code>. See the <i>Plib Reference</i> manual and the <i>WDR Printing</i> chapter of the <i>Additional System Information</i> manual for more detail. The table itself is loaded from the WDR resource file during execution of the <code>pdr_init</code> method.
<code>commands</code>	This is a pointer to a table of commands supported by the current printer model(s). The table starts with a byte count giving the number of commands followed by the commands themselves. Each command starts with a byte count giving the total length of the command. The remaining bytes contain a format string consisting of the printer command itself and formatting characters as used by the <code>Plib</code> function <code>p_atob</code>. See the <i>Plib Reference</i> manual and the <i>WDR Printing</i> chapter of the <i>Additional System Information</i> manual for more detail. The table itself is loaded from the WDR resource file during execution of the <code>pdr_init</code> method.		

outbuf	The address of the output buffer in which the sequences of printer commands are built.
trans_res	A pointer to a translate table as loaded in from the resource file. Translate tables are described in the <i>WDR Printing</i> chapter of the <i>Additional System Information</i> manual for more detail.
tix	A pointer to a lookup table for the translate table referenced by trans_res. The lookup table is a table of <i>addresses</i>. The ASCII value of any character gives the offset into the lookup table for that character's entry which, in turn, gives the address of the translate table entry <i>for that character</i>. In other words : <code>*(pdr.a.tix+(ASCII value of a char'))</code> points to the translate table entry for that character.
pdr.trans_rid	The resource ID of the current translate table.
pdr.outlen	The length of data currently held in the output (i.e. the commands) buffer which is in allocated memory pointed by pdr.a.outbuf.
pdr.skipy	This property is set to the printer vertical auto-feed value whenever a page break occurs (pdr_page) and the printer is started (pdr_start). It is reset to zero after every new line (pdr_line). The printer vertical auto-feed value itself is copied from the printer model information supplied when this instance of PDR is created (see the PDR_INIT and the WDR_MODEL data structures.) This property is used to calculate the amount by which the print head must <i>actually</i> move down when a new line is requested and is of particular importance when a page break occurs.

PDR methods

DESTROY

Destroy

VOID destroy(VOID);

Destroy the PDR instance.

The destroy method begins by sending PDR_DESTROY message. The pdr_destroy method, as supplied in PDR, is a dummy method which can be replaced by a subclass. The intention is that any *subclass specific* destroy tasks are done, and indeed *must only* be done, within the pdr_destroy method.

(Do not be confused between the destroy method and the pdr_destroy method.)

All memory whose pointers are held in the PDR_ALLOC data structure in property pdr.a are freed.

If an external DYL was loaded (indicated by a positive value in pdr.par.dyl), as is often the case when PDR is subclassed, the DYL whose handle is contained in pdr.par.dyl, is unloaded.

The method finally supersends a DESTROY message.

This method *must not* be replaced by subclassers - an attempt to return into the DYL after it has been freed will fail.

PDR_INIT

Initialise PDR

VOID pdr_init(PDR_INIT *par, VOID **ppcb);

Initialise the instance of PDR.

The parameter par points to a data structure of type PDR_INIT which contains the information required to initialise the instance. The entire content of *par is copied into the property pdr.par.

The PDR_INIT structure, shown below, is defined in *prdrv.cl*:

```
typedef struct
{
    PR_WDR *wdr;
    WORD flags;
    WORD page_length;
    HANDLE dyl;
    WDR_HEADER *head;
    WDR_MODEL *model;
} PDR_INIT
```

The significance of the members of the PDR_INIT struct is as follows:

wdr	The handle of an instance of an associated WDR class. Although the PDR object is created by this instance of WDR, it does require the services of this WDR object.
flags	An ORED combination of flags as follows: WDR_PDR_LANDSCAPE - if set, it indicates that landscape orientation is required. This is the only flag to be set in this property; subclassers may wish to add additional flags.
page_length	The page height in <i>printer units</i> .
dyl	If the PDR class is used directly, this member is NULL. If the PDR class is subclassed and the subclass has been loaded from a DYL, then this member will contain the category handle of that DYL.
head	The address of the WDR file header resource.
model	The address of the WDR_MODEL data structure containing the information on the current printer model.

The method opens a channel to the printer port and writes the handle of the opened channel to *ppcb by sending a PR_OPEN_PORT message to the object whose handle is contained in w_am->appman.spare1. This is a property of the application manager and is assumed to contain the handle of an instance of the PRINTER class or its equivalent. A FORM PRINTER object always inserts a copy of its own handle into w_am->appman.spare1 during initialisation.

A WDR_LOAD_RECORD message is sent to the associated WDR object requesting it to load the commands resource for the current printer from the WDR resource file; the address of the loaded resource is written to pdr.a.commands

The method allocates a cell of length 256 bytes and writes its address to pdr.a.outbuf. This area will be used to build the sequence of printer commands. The cell will be re-allocated if it eventually proves to be too short.

On the Series 3, there is a slight difference to the behaviour of this method. See the Series 3a/Series 3 notes section at the end of this chapter for detail.

PDR_PRINT **Translate print command**

```
INT pdr_print(WDR_PRINT *pr, UBYTE **pbuf);
```

Translate a print element into a sequence of printer specific commands and return the length of the generated commands.

The print element is a data structure of type WDR_PRINT pointed to by the parameter pr. The method takes the print element and translates the contents into a sequence of printer specific commands; the commands themselves are written to a buffer whose address is contained in the property pdr.a.outbuf and the start address of the sequence is written to *pbuf.

Before starting the translation process, the property pdr.out.len (the current length of the data in the output buffer) is reset to zero.

The command sequences generated depend on the setting of the `flags` member of the print element. Separate methods exist to handle each possible setting of `flags`. The detailed work is delegated to other PDR methods, each of which builds and adds commands to existing command sequences in the output buffer and updates the current length of the buffer as held in the property `pdr.outlen`.

The method finally writes the address of the output buffer to `*pbuf` and returns the length of the generated command sequences, i.e. the current value of `pdr.outlen`. This return value is used by the calling instance of `PAGES` to determine how much data to transmit to the printer. A subclass of `PDR` that wishes to direct output to a device other than the printer may replace this method to perform the output and return zero. In such a case, the output must have completed before the `pdr_print` method returns.

As mentioned earlier, a print element is a data structure of type `WDR_PRINT` defined in *prdrv.cl*; this is shown below together with an explanation of the individual fields:

```
typedef struct
{
    WORD flags;
    WORD typf;
    WORD fheight;
    WORD style;
    WORD down;
    WORD indent;
    WORD height;
    WORD right;
    TEXT *buf;
    UWORD blen;
} WDR_PRINT;
```

`flags` This member gives meaning to the print element. It is an ORed combination of flags.

The command sequences generated by `pdr.print` depend on the combinations set. However, they are added to the output buffer in the same order as the flags are described below.

`WDR_PRINT_START` This flag indicates that the printer is to be started. The command sequence to do this is generated by sending a `PDR_START` message. The method requires no arguments.

`WDR_PRINT_PAGE` This flag indicates that the print element is to go onto a new page; in other words, a page break is required. The command sequence to do this is generated by sending a `PDR_PAGE` message. The method requires no arguments.

`WDR_PRINT_LINE` This flag indicates that the print element is to go onto a new line. A copy of the line height as found in `pr->height` is copied into the property `pdr.lheight`, as this could prove useful to subclasses.

The command sequence to force a new line is generated by sending a `PDR_LINE` message. The method requires two arguments which govern the initial position of the printhead on the new line - the vertical distance through which the print head is to move *down* and the horizontal distance the print head is to move to the *right* from the left-hand margin.

The first argument is the value of `pr->down` *plus* `pr->height`.

The second argument is the value of `pr->indent`.

`WDR_PRINT_FONT` This flag indicates that a new font is to be set. The command sequence to do this is generated by sending a `PDR_FONT` message.

The method requires three arguments - the typeface index, the height of the font and the required style.

The arguments are the values `pr->typf`, `pr->fheight` and `pr->style` respectively.

WDR_PRINT_RIGHT	This flag indicates that the print head must be moved <i>to the right</i>. The command sequence to do this is generated by sending a PDR_RIGHT message. The method requires a single argument which specifies the amount by which the print head is to be moved. The argument is the value of <code>pr->right</code>. Note:- if WDR_PRINT_TEXT is also set, then the print head is moved to the right <i>before</i> attempting to print any text.												
WDR_PRINT_TEXT	This flag indicates that there is text to be printed. The command sequence to do this is generated by sending a PDR_TEXT message. The method requires two arguments which define the text to be printed - the address of a buffer containing the text and the length of the text to be printed. The first argument is <code>pr->buf</code>. The second argument is the value of <code>pr->blen</code>. Note:- if WDR_PRINT_RIGHT is also set, then the print head is moved to the right <i>before</i> starting to print any text.												
WDR_PRINT_END	This flag indicates that this print element terminates the print process. The command sequence to do this is generated by sending a PDR_END message. The method requires no arguments.												
typf	The typeface number. It is a number that uniquely identifies the typeface, (Courier, for example) and is compatible with DOS/WORD font numbers. Each typeface resource contains a typeface number as part of its identity; see the WDR_TYPEFACE structure in the WDR class definition. This member is important when the WDR_PRINT_FONT flag is set and is used as an argument to the <code>pdr_font</code> method.												
fheight	The height of the font in twips. This member is important when the WDR_PRINT_FONT flag is set and is used as an argument to the <code>pdr_font</code> method.												
style	The style to be applied to the font. The style is represented by an ored combination of flags each of which represents an individual style as shown below. This member is important when the WDR_PRINT_FONT flag is set and is used as an argument to the <code>pdr_font</code> method. <table><tr><td>WDR_STYLE_NORMAL</td><td>Plain text.</td></tr><tr><td>WDR_STYLE_UNDERLINE</td><td>Text is underlined.</td></tr><tr><td>WDR_STYLE_BOLD</td><td>Text is boldened.</td></tr><tr><td>WDR_STYLE_ITALIC</td><td>Text is italicised.</td></tr><tr><td>WDR_STYLE_SUPER</td><td>Text is superscripted.</td></tr><tr><td>WDR_STYLE_SUB</td><td>Text is subscripted.</td></tr></table>	WDR_STYLE_NORMAL	Plain text.	WDR_STYLE_UNDERLINE	Text is underlined.	WDR_STYLE_BOLD	Text is boldened.	WDR_STYLE_ITALIC	Text is italicised.	WDR_STYLE_SUPER	Text is superscripted.	WDR_STYLE_SUB	Text is subscripted.
WDR_STYLE_NORMAL	Plain text.												
WDR_STYLE_UNDERLINE	Text is underlined.												
WDR_STYLE_BOLD	Text is boldened.												
WDR_STYLE_ITALIC	Text is italicised.												
WDR_STYLE_SUPER	Text is superscripted.												
WDR_STYLE_SUB	Text is subscripted.												

down	The downwards displacement of the print head in printer units. When a print element is to go onto a new line, the print head is moved down by this value <i>plus</i> the value given in height. This member gives a mechanism for defining the <i>extra</i> spacing which is often needed before a line of text is printed (as occurs, for example, before the first line of a new paragraph.) This member is important when the <code>WDR_PRINT_LINE</code> flag is set and is used in the construction of an argument to the <code>pdr_line</code> method.
indent	The right indentation of the print head in printer units. This member is important when the <code>WDR_PRINT_LINE</code> flag is set and is used as an argument to the <code>pdr_line</code> method.
height	The height of the line in printer units. When a print element is to go onto a new line, the print head is moved down by this value <i>plus</i> the value given in down. This member is important when the <code>WDR_PRINT_LINE</code> flag is set and is used in the construction of an argument to the <code>pdr_line</code> method.
right	The rightwards displacement of the print head, in printer units, from its current position. It defines how far to the right the print head is to move. If the print element also contains text to be printed, the print head is moved right <i>before</i> text is printed. This member is important when the <code>WDR_PRINT_RIGHT</code> flag is set and is used as an argument to the <code>pdr_right</code> method.
buf	The address of a buffer containing text to print. This member is important when the <code>WDR_PRINT_TEXT</code> flag is set and is used as an argument to the <code>pdr_text</code> method.
blen	The length of the text to print. This member is important when the <code>WDR_PRINT_TEXT</code> flag is set and is used as an argument to the <code>pdr_text</code> method.

PDR_ADD_COMMAND

Add command to buffer

```
VOID pdr_add_command(INT num,WORD *args);
```

Add a printer command to the output buffer.

This method takes two parameters:

- `num` represents the number of the command format string within the commands table. It can take one of the `PDR_CMD_...` values as defined in the sub-category file *prdrv.cl*. The address of the commands table is in `pdr.a.commands`.

Recall that, in general, a command format string consists of the command itself (one or more characters) followed by formatting control characters which are discussed in the description of the `Plib` function `p_atob` in the *Plib Reference* manual.

- `args` points to a contiguous list of arguments; this parameter will be `NULL` if the printer command requires no arguments.

If no arguments are supplied, the command sequence added to the output buffer is simply the printer command as found in the commands table.

If arguments are supplied, the way the method proceeds depends on the content of the command format string as follows:

- If the *first* character of the command format string is an asterisk (*), the first argument pointed to by `args` is assumed to be an integer value containing a *repeat count*. Any other arguments follow the repeat count.
- If the first character is not an asterisk, the method assumes a repeat count of one.

- A single command sequence is constructed, consisting of the printer command and the values in the argument list converted according to the formatting control characters.
- A number of copies of the constructed single command sequence are added to the output buffer as defined by the repeat count calculated above.

A special case occurs where the first character of the command string is an asterisk and the next character is NULL (i.e. the character '\0'); again, `args` will point to an integer value containing a repeat count. In this situation, a number of '\0' characters are added to the output buffer as defined by the repeat count.

The method automatically re-allocates the output buffer if it is not big enough.

PDR_DESTROY Destroy method, subclassable by DYL

```
VOID pdr_destroy(VOID);
```

This is a dummy method and does nothing.

The method is intended for use by subclasses; its use is more fully discussed in the description of the `destroy` method.

PDR_START Start printing

```
VOID pdr_start(VOID);
```

Generate the command sequences to initialise and set up the printer and add them to the output buffer.

The following commands are added to the output buffer by sending a `PDR_ADD_COMMAND` message:

- `PDR_CMD_RESET` instructing the printer to reset itself.
- `PDR_CMD_FORM_LENGTH` to set the form length. This requires a single argument specifying the length of the form. The length is specified in units of *lines* and the assumption is made that there are 6 lines per inch; thus the value passed is the result of the calculation:


```
(pdr.par.page_length * pdr.par.model->miny) / 240
```
- `PDR_CMD_PREAMBLE`.
- `PDR_CMD_LANDSCAPE` instructing the printer to operate in landscape mode *if and only if*, on initialisation of this instance of `PDR`, landscape orientation was requested *and* the current printer model supports landscape orientation (i.e. `WDR_PDR_LANDSCAPE` is set in `pdr.par.flags` *and* `WDR_MODEL_LANDSCAPE_AVAILABLE` is set in `pdr.par.model->flags`).

If this command is generated, then the `WDR_PDR_LANDSCAPE` flag is set into the property `pdr.mode`.

The property `pdr.typfix` containing the index into the array of pointers to `WDR_TYPEFACE` structures, corresponding to the *current typeface*, is initialised to -1. As any sensible index is always non-negative, this guarantees that any subsequent call to the `pdr_font` method will cause the relevant translate table resource to be loaded.

To complete the start up command sequence, the `pdr_font` method is called to ensure that a default font is set. The default font has a typeface number of zero, a font height of 240 twips and normal style. It is worth noting that 240 twips is the height of a single line based on the assumption that there are 6 lines per inch.

Finally, the method writes the printer auto-feed value (as found in the printer model information `pdr.par.model->skipy`) into the property `pdr.skipy`. This ensures that the downward displacement of the print head for the first new line is calculated correctly.

PDR_END Finish printing

```
VOID pdr_end(VOID);
```

Generate the command sequences to terminate printing and add them to the output buffer.

This method calls the `pdr_page` method to add a `PDR_CMD_NEW_PAGE` command to the output buffer and then adds a `PDR_CMD_POSTAMBLE` command directly by calling the `pdr_add_command` method.

PDR_PAGE**Start a new page**

```
VOID pdr_page(VOID);
```

Generate the command sequences to start a new page and add them to the output buffer.

This method adds a `PDR_CMD_NEW_PAGE` command to the output buffer and then writes the printer auto-feed value (as found in the printer model information `pdr.par.model->skipy`) into the property `pdr.skipy`. This ensures that the downward displacement of the print head for the first new line on the new page, is calculated correctly.

PDR_TEXT**Print text at current position**

```
VOID pdr_text(TEXT *buf, INT len);
```

Generate the command sequences to print text at the current position and add them to the output buffer.

This method takes two parameters; `buf` points to a buffer containing the text to be printed and `len` contains the length of the buffer.

The method substitutes the following characters into the buffer:

- all `SCRLAY_SYM_SOFT_HYPHEN` and `SCRLAY_SYM_HARD_HYPHEN` characters are replaced with a '-' character.
- all `SCRLAY_SYM_HARD_SPACE` characters are replaced with a ' ' character.

If a translate table exists, the method translates any characters that need to be translated using both the translate table and its associated lookup table (see the description of the property `pdr.a.tix` and `pdr.a.trans_res`).

Finally, the method adds the text, including all of the substitutions and translations, to the output buffer.

PDR_LINE**Start a new line**

```
VOID pdr_line(INT down, INT indent);
```

Generate the command sequences to start a new line and add them to the output buffer.

The method takes two parameters; `down` specifies the vertical distance through which the print head is to move; `indent` specifies the initial position of the print head relative to the *left-hand* edge of the page. Both `indent` and `down` are given in printer units.

The following commands are added to the output buffer by sending a `PDR_ADD_COMMAND` message:

- `PDR_CMD_CARRIAGE_RETURN`.
- `PDR_CMD_MOVE_DOWN` to move the print head down. This requires a single argument specifying the amount of vertical travel. If the parameter `down` is non-zero *and* this is the first new line on the page, then the argument passed is the value of `down` *less* the vertical printer auto-feed value. (A negative result is reset to zero)

In the context of this method, the first line on a new page is implied by the value of the property `pdr.skipy`. This is set to the printer auto-feed value at the start of printing and on page breaks; it is reset to zero *by this method* after the `PDR_CMD_MOVE_DOWN` command has been added to the output buffer.

If the parameter `indent` is non-zero, the print head is to be moved horizontally. However, *before* this is done, the printer style is reset to normal, if it is other than normal. The pre-existing style is re-applied *after* the print head has moved.

Thus, if the parameter `indent` is non-zero, the following occurs:

- If the existing printer style is other than normal, a `PDR_STYLE` message is sent with an argument of `WDR_STYLE_NORMAL` to add a command sequence to the output buffer to reset the style to normal. The existing style, as defined by the content of the property `pdr.style` is temporarily saved.

- A `PDR_RIGHT` message is sent to add a command sequence to move the print head right. The `pdr_right` method itself requires an argument specifying the amount of horizontal travel. The value of the argument passed is the value of `indent` less the horizontal printer auto-feed value (A negative result is reset to zero).
- If necessary, a `PDR_STYLE` message is sent to add a command sequence to the output buffer in order to re-set the style to its pre-existing value.

PDR_RIGHT

Position to the right

```
VOID pdr_right(INT right);
```

Generate the command sequences to move the print head *right* from its current position and add them to the output buffer.

The method takes a single parameter; `right` specifies the horizontal distance through which the print head is to move from its current position and is given in printer units.

The following commands are added to the output buffer by sending a `PDR_ADD_COMMAND` message:

- `PDR_CMD_MOVE_RIGHT_PREFIX.`
- `PDR_CMD_MOVE_RIGHT.`
- `PDR_CMD_MOVE_RIGHT_SUFFIX.`

ALL three commands require a single argument specifying the amount of horizontal travel; the argument passed to all commands is the value of the parameter `right`.

The method generates three command sequences to allow for the fact that some printers require a mode switch before and after the actual move right command. However, for many printers, this is not necessary and both the prefix and suffix commands are effectively null.

PDR_FONT

Set the font

```
VOID pdr_font(INT typf,INT height,INT style);
```

Generate the command sequences to set the font and add them to the output buffer.

The method takes three parameters:

- `typf` specifies the typeface number which identifies the typeface, (Courier, for example) and is compatible with DOS/WORD font numbers
- `height` specifies the height of the font in twips
- `style` specifies the style to be applied and is represented by an ored combination of flags (see the description of the `pdr_style` method for the flags and their meanings)

The method sends a `WDR_SEARCH_TYPEFACE` message to the associated `WDR` object to retrieve the index of the entry within the array of pointers to `WDR_TYPEFACE` structures which represents the typeface with number `typf`. This index is referred to as the typeface index.

The method then sends a `WDR_SEARCH_HEIGHT` message to retrieve the index of the entry within the array of `WDR_FONT` structures which most closely represents the font with height `height`. This index is referred to as the font index.

See the section on `WDR` in this chapter for a full description of the data structures and for more detail on the `wdr_search_typeface` and `wdr_search_height` methods.

If *both* the typeface index and the font index are the same as the current values held in the properties `pdr.typfix` and `pdr.fhix` respectively, then the method only needs to send a `PDR_STYLE` message with an argument of `style`, to add the command sequence to set the style.

If, however, *either* the typeface index *or* the font index are different from the current values held in `pdr.typfix` and `pdr.fhix`:

- Either one or both parameters `height` and `style` are set into the properties `pdr.typfix` and `pdr.fhix`, respectively to reflect the changes. These new values are now regarded as the current values.
- If the resource ID of the translate table for the *specified* typeface is different from the current translate table resource ID as defined by the property `pdr.trans rid`, then the new translate table is loaded by sending a `WDR_LOAD_RECORD` message to the associated `WDR` object and the `pdr.trans rid` is updated with the new resource ID. The translate lookup table is also re-built.
- A `PDR_STYLE` message is sent, with an argument of zero, to add a command sequence to the output buffer to set the style to normal.
- For a non-scalable font, a `PDR_ADD_COMMAND` message is sent to add the 'set font' command as found in the command member of the *current* `WDR_FONT` data structure; formally:-
`self->pdr.par.model.typeface[pdr.typfix]->font[pdr.fhix].command`
- For a scalable font, a `PDR_ADD_COMMAND` message is sent to add the 'set font' command as found in the command member of the *first* `WDR_FONT` data structure; formally:-
`self->pdr.par.model.typeface[pdr.typfix]->font[0].command`

This command requires an argument specifying the font height in *points*.

- Finally, a `PDR_STYLE` message with an argument of `style`, is sent to add the command sequence to set the style.

PDR_STYLE

Set the font style

```
VOID pdr_style(INT style);
```

Generate the command sequences to set the style and add them to the output buffer.

The method takes a single parameter; `style` specifies the style to be applied.

In practice, `style` is a combination of individual styles each of which is represented by a flag. The flags, which can be ORED together, are as follows :

<code>WDR_STYLE_NORMAL</code>	specifies that the style is set to normal - i.e. no underline, no italic etc.
<code>WDR_STYLE_UNDERLINE</code>	specifies that underlining is to be set.
<code>WDR_STYLE_BOLD</code>	specifies that bold is to be set.
<code>WDR_STYLE_ITALIC</code>	specifies that italic is to be set.
<code>WDR_STYLE_SUPER</code>	specifies that superscript is to be set.
<code>WDR_STYLE_SUB</code>	specifies that subscript is to be set.

The property `pdr.style` contains the current setting of the style flags. By comparing the current style with the required new style, as defined by the content of the parameter `style`, the method generates a sequence of commands to turn *individual* styles on or off as appropriate. The method uses the services of the `pdr_add_command` method to add the commands to the output buffer.

For example, to turn underlining off and bold emphasis on, the commands `PDR_CMD_UNDERLINE_ON` and `PDR_CMD_BOLD_OFF` are generated and added to the output buffer.

The method concludes by setting the property `pdr.style` to the value in the parameter `style`.

The PAGELAY mixin class

PAGELAY
Mread mdone

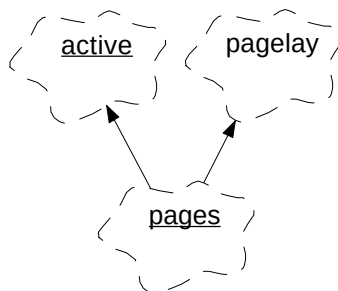
The PAGELAY mixin class provides the *formal specification* for the call-back methods that must be supported by any class that provides access to the text content of a formatted document. These methods may be called by the PAGES class. The call-back methods `mread` and `mdone` are also referred to as the *read* and the *done* methods in other parts of this manual.

For a general discussion on call-back methods and mixin classes, see the *Introduction* chapter in this manual.

The PAGELAY class does not appear in the FORM library and an instance of PAGELAY will never be created. The FORM library supplies the PRNLAY class to build and manipulate the layout of printer text. This class, described later in this chapter, provides the `sl_print_read` method as the required `mread` or the *read* callback method. However, the FORM library does *not* supply a class which can provide the necessary `mdone` or *done* call-back method; this is normally supplied by the application.

Class diagram

The following class diagram formally illustrates the relationship between the PAGELAY mixin class and the PAGES class. This diagram shows both the ACTIVE class and the PAGELAY class in order to emphasise the multiple inheritance aspect of mixin classes.



Class definition

```

CLASS pagelay root
{
  DEFER mread          fetch print element
  DEFER mdone          report status
}
  
```

Property

None.

PAGELAY call-back methods

PAGELAY_MREAD

Get next WDR_PRINT element

```
UINT pagelay_mread(INT flag, WDR_PRINT *pr)
```

This method is also referred to as the *read* method in this chapter.

The printing/previewing/pagination process is normally broken down into a sequence of operations such as moving the printing position down, changing the typeface, printing a number of characters and so on. Each of these operations can be described by what is known as a print element.

Once created, the `PAGES` object drives this process by requesting print elements from the application. It is the application's responsibility to "know" what it wants to do next.

For print and preview operations, the `PAGES` object is responsible for taking a print element and translating it into an appropriate sequence of commands suitable for the chosen printer and scheduling any resulting I/O operations.

For pagination operations, the `PAGES` object uses the print elements to build pagination information.

This method provides the mechanism by which a `PAGES` object obtains a print element from an application.

A print element is represented by the information contained in a data structure of type `WDR_PRINT`. While this structure is used by the application, the `PAGES` object and by the printing (`PDR`) object, it is in fact defined in the printer resource (`WDR`) class definition.

`PAGES` supplies two parameters when calling this method:

- `pr` is a pointer to a data structure of type `WDR_PRINT`.
- `flag` indicates the type of operation for which the instance of `PAGES` has been constructed; if set to `TRUE`, the `PAGES` object is performing a print or preview operation; if set to `FALSE`, the `PAGES` object is performing a pagination operation.

The method must insert the information which describes the next element to be printed, into this data structure. While the *Printing* chapter in the *Object Oriented Programming Guide* discusses `WDR_PRINT` in greater detail, an overview is given below.

The `pr->flags` field describes the *type* of the print element and affects how the element is to be used. A number of flags may be set into this field; they are *not* mutually exclusive and are summarised below.

<code>WDR_PRINT_START</code>	If set, this print element will start the printing process. In effect it causes the printer to be reset and appropriately initialised.
<code>WDR_PRINT_END</code>	If set, this print element will terminate the printing process. It tells the <code>PAGES</code> object that there is no more data available.
<code>WDR_PRINT_LINE</code>	If set, the current print position is to be moved back to the beginning of the line, then down by <code>pr->down</code> printer units, then down again by <code>pr->height</code> printer units and finally moved right by <code>pr->indent</code> printer units.
<code>WDR_PRINT_FONT</code>	If set, the font (i.e. the typeface) is to be changed to that specified in <code>pr->typf</code> , the height changed to that specified in <code>pr->fheight</code> and the style changed to that specified in <code>pr->style</code> .
<code>WDR_PRINT_TEXT</code>	If set, <code>pr->blen</code> bytes of text from the buffer pointed to by <code>pr->buf</code> are to be printed.
<code>WDR_PRINT_KEEP</code>	If set, the line containing this print element is to be kept, if possible, on the same page as the following print element.
<code>WDR_PRINT_PAGE</code>	If set, this print element is to go onto a new page. In other words, a page break will be forced.
<code>WDR_PRINT_IDLE</code>	If set, the <code>PAGES</code> object is to ignore this print element and then suspend itself. To resume, the application must send the <code>PAGES</code> object an explicit <code>AO_QUEUE</code> message.

PAGELAY_MDONE

Handle status messages

```
INT pagelay_mdone(PAGES_DONE *pdone, PAGES_PARAMS *par);
```

This method is also referred to as the *done* method in this chapter.

This call-back method, supplied by the application, provides the mechanism by which a `PAGES` object can keep an application informed of the current status of the printing, previewing or paginating operation.

The content of the method is application dependent; however, it must take note of the information `PAGES` passes to it. In return, `PAGES` may require "feedback" from the method depending on the precise circumstances in which it is called.

PAGES passes two parameters to the method:

- `par` is a pointer to a data structure of type `PAGES_PARAMS`, containing such information as page dimensions. See the `PAGES` class for more detail on the contents of this structure.
- `pdone` is a pointer to a data structure of type `PAGES_DONE`.

`par` will contain the handle of the `pages.par` property of `PAGES`.

`pdone` will contain information *relating* to the current status; in particular `pdone.event` *indicates* the status of the printing, previewing or paginating operation and can take one of the following values:

PAGES_DONE_END	Set when printing, previewing or paginating is complete and no further documents are to be printed. Whether printing or paginating, the <code>PAGES</code> object destroys itself after this call-back method returns. If <code>PAGES</code> is paginating, the page array will have been built and its handle placed in <code>pdone->pages</code>. Responsibility for the page array passes to the application which must ensure that it is destroyed before the application itself terminates. Any value returned by this method is ignored.
PAGES_DONE_PAGE	Set when a page break occurs. <code>PAGES</code> puts the new page number into <code>pdone->page</code>. Any value returned by this method is ignored.
PAGES_DONE_ERROR	Set when an error occurs. If an error occurs, the <code>PAGES ao_abrun</code> method is called which: • does a notify by supersending an <code>AO_ABRUN</code> message • calls <i>this</i> call-back method to inform the application • sends itself a <code>DESTROY</code> message Any value returned by this method is ignored.
PAGES_DONE_DOC	Set when the printing, previewing or paginating of a document is complete. If more copies of the same document or new documents are to be printed then the method should return <code>TRUE</code>; if <i>no more</i> documents are to be printed then the method should return <code>FALSE</code>.

PRNLAY

SCRLAY	PRNLAY
paras first nomemory adjust scan rd fmt doc slines spadjust st	tbxlen senselen sensebuf pos line nlines below excess ngaps ngap used ptab plabel
destroy sl_init sl_set sl_sense sl_line_ends sl_pos_to_xl sl_xl_to_pos sl_begin_read sl_read sl_format_line sl_scroll sl_view sl_discard_layout sl_set_lines sl_para_changed sl_rescale	sl_print_read sl_print_pos

The PRNLAY class, in conjunction with its superclass SCRLAY, provides services to lay out lines and line segments for a printer, from a document that is composed of a sequence of paragraphs.

In effect, PRNLAY provides property and structures which *model* the layout of a document on the printer. The methods supplied by this class allow the layout *model* to be manipulated.

It is important to note that the behaviour of PRNLAY is, in essence, the same as the document layout class SCRLAY. All of SCRLAY's property and behaviour is re-usable by PRNLAY.

Only two extra methods are provided by PRNLAY in order to provide the full behaviour.

An instance of PRNLAY is normally referenced by two other objects: its creator (normally a window class) and a PAGES class. PAGES needs a class to provide it with a *read* call-back method, a method which can supply PAGES with a sequence of print elements; many applications, such as the word-processor, use the PRNLAY `sl_print_read` method as the call-back method; the specification for the PAGES read call-back method can be found in the description of the PAGELAY mixin class in this chapter.

PAGES, itself, creates an instance of PRNLAY to handle layout for header text.

This section makes references to data structures (e.g. Tboxes) which are described in the section on the SCRLAY class in *The Document Layout Classes* chapter of this manual. It is strongly recommended that PRNLAY be read in conjunction with SCRLAY.

Class definition

The PRNLAY class subclasses the FORM class SCRLAY and is defined in the sub-category file *scrlay.cl* (with generated header *scrlay.g*). The SCRLAY class is documented in *The Document Layout Classes* chapter in this manual.

```
CLASS prnlay scrlay
{
  ADD sl_print_read      Read text for printing
  ADD sl_print_pos       Set the start position for printing

  TYPES
  {
    typedef struct
    {
      SCRLAY_PLABEL s;    as for the screen
      UBYTE *wid;         the font width table
      UWORD margin;       margin for paragraph labels in printer units
      UWORD gutter;       gutter between label and para margin
    } PRNLAY_PLABEL;
  }

  PROPERTY
  {
    WORD tbxlen;          Number of bytes to still to read from TBOX
    WORD senselen;        Number of bytes to still to read from sensebuf
    TEXT *sensebuf;       Address of text being read (justified only)
    UWORD pos;            Document position to read
    WORD line;            Current line in paragraph
    WORD nlines;          Number of lines in paragraph
    WORD below;           Carried over from previous paragraph
    WORD excess;          Excess width for justified alignment
    WORD ngaps;           Number of gaps for justified alignment
    WORD ngap;            Number of gaps so far
    WORD used;            Excess width used so far
    SCRLAY_TBOX *ptab;    Last tab in line or NULL if no tabs
    WORD plabel;          Line has a para label if TRUE
  }
}
```

Property

<code>prnlay.tbxlen</code>	This contains the number of characters remaining to be read from the current Tbox. See <code>SCRLAY</code> for a definition of <code>SCRLAY_TBOX</code> . This property is re-set to zero by the <code>sl_print_pos</code> method.
<code>prnlay.senselen</code>	This property is only relevant for paragraphs with <i>justified alignment</i>. It is used in conjunction with the <code>prnlay.sensebuf</code> property and records the number of characters remaining to be processed within a block of contiguous characters sensed from the document using the <code>sensechars</code> call-back method. This property is re-set to zero by the <code>sl_print_pos</code> method
<code>prnlay.sensebuf</code>	This property is only relevant for paragraphs with <i>justified alignment</i>. The handling of a block of contiguous characters, sensed from the document using the <code>sensechars</code> call-back method, is slightly different when justified alignment is used. Each contiguous section of non-blank characters in the block must be printed separately. This allows the gaps to be adjusted (by moving the print head) to ensure correct alignment. This property records the current position within a block of contiguous characters.
<code>prnlay.pos</code>	The position within the document where text is to be read from next.
<code>prnlay.line</code>	The current line in the current paragraph. Note that the first line is line <i>zero</i> .
<code>prnlay.nlines</code>	The total number of lines in the current paragraph

<code>prnlay.below</code>	This is the space required <i>below</i> the <i>previous</i> paragraph, measured in printer units. The value is added to the value of the space above the current paragraph to calculate the total downward movement of the print head before printing the first line of the current paragraph. This property is re-set to zero by the <code>sl_print_pos</code> method.
<code>prnlay.excess</code>	This property is only relevant for paragraphs with <i>justified alignment</i>. It is a measure of the number of pixels by which the characters on a line (excluding any trailing whitespace) fall short of the right hand margin. This value is used in the calculation of the adjustment to the gaps between contiguous non-blank characters, necessary to achieve justified alignment.
<code>prnlay.ngaps</code>	This property is only relevant for paragraphs with <i>justified alignment</i>. This is the number of gaps in a line; generally speaking, a gap corresponds with a blank character. If there are any left hand used tabs in the line, it is the number of gaps after the <i>last used</i> left hand tab. This property is re-set in the <code>SL_PRINT_READ</code> method whenever the first Tbox in a line is being handled.
<code>prnlay.ngap</code>	This property is only relevant for paragraphs with <i>justified alignment</i>. It is used in the <code>SL_PRINT_READ</code> method to keep a record of how many of the gaps in a line have been adjusted, when printing that line. This information is used to ensure that the line is aligned correctly. This property is re-set in the <code>SL_PRINT_READ</code> method whenever the first Tbox in a line is being handled.
<code>prnlay.used</code>	This property is only relevant for paragraphs with <i>justified alignment</i>. It is used in the <code>SL_PRINT_READ</code> method to keep a record of how much of the excess width in a line has been "used up" by the adjustment of gaps between words.
<code>prnlay.ptab</code>	The address of the final <i>used</i> tab on a line or <code>NULL</code> if the line has no used tabs. The tab is represented by a Tbox (a <code>SCRLAY_TBOX</code> structure).
<code>prnlay.plabel</code>	This property is only relevant if a <code>senseplabel</code> call-back method is supplied by the document content object. In this event, paragraph labels are to be printed. It records the number of printer units the printhead must move <i>after</i> the label has been printed, to reach the start of the paragraph margin. This property is only relevant to the first line of a paragraph.

PRNLAY methods

SL_PRINT_READ

Read text for printing

```
UINT sl_print_read(INT wantData, WDR_PRINT *pr);
```

Read a portion of text for printing and build a print element containing information which describes the text, the typeface, the font height etc.

The method takes two parameters:

- `wantData` indicates the type of operation for which this portion of text is being read. If set to `TRUE`, a print or preview operation is in progress; if set to `FALSE`, a pagination operation is in progress.
- `pr` points to a data structure of type `WDR_PRINT`. This structure represents what is known as a print element. The method writes all the necessary information about the portion of text into this data structure. Although this structure is defined in the `WDR` class definition, it is used by `PAGES` methods and `PDR` methods. See the description of the `PDR` class and the *Printing* chapter of the *Object Oriented Programming Guide*.

It returns the position within the document corresponding to the start of the portion of text represented by the print element.

The method constructs layout for one paragraph at a time and uses this information to construct a series of `WDR_PRINT` print element for successive sections of text. The method only builds one print element at a time but is expected to be called repeatedly until all of the document has been processed. Much of the property of `PRNLAY` (and its superclass `SCRLAY`) is used to record the "current position" within the document and the layout data structures.

The method will correctly calculate items such as right indentations and the address/length of text to be printed; it will also flag line breaks and font changes as required by setting the appropriate `WDR_PRINT_...` flags. The method will also flag a new page when a paragraph is required to start on a new page.

If the `senseLabel` call-back method has been supplied, `WDR_PRINT` elements will be created to cause a label to be printed in the left hand margin of the first line.

When the end of the document is reached, a `WDR_PRINT_END` flag will be set; this will, ultimately, cause printing to terminate.

SL_PRINT_POS

Set start position for printing

```
VOID sl_print_pos(UINT pos, UINT doclen);
```

Set the start position and the document length for the next call to `SL_PRINT_READ`.

The method takes two parameters:

- `pos` indicates the start position within the document to be printed. This position should correspond to the beginning of a paragraph.
- `doclen` contains the length of the document

The method discards any existing layout by sending an `SL_DISCARD_LAYOUT` message. If the value passed in the parameter `doclen` is non-zero, this value is recorded as the new document length. If the value is zero, the recorded document length remains unchanged.

A number of items of property are re-set. The following list shows which items are re-set and the corresponding new values:

<code>scr lay.rd.pos</code>	the value in <code>pos</code>
<code>scr lay.fmt.pos</code>	the value in <code>pos</code>
<code>scr lay.rd.pp</code>	NULL
<code>prn lay.tbxlen</code>	0
<code>prn lay.senselen</code>	0
<code>prn lay.below</code>	0

Series 3a/Series 3 notes

The version of the FORM classes described in this chapter are those which exist on the Series 3a and *Workabout*. On the Series 3, the classes and the relationships between them are essentially the same. However, there are some differences which need to be discussed.

1. On the Series 3, the `pr_print` method opens the printer port device itself (by sending a `PR_OPEN_PORT` message) rather than allowing it to be done by the `pdr_init` method as occurs on the Series 3a.

The handle to the opened port is passed to the `PAGES` object as the second parameter in the call to the `PAGES ao_init` method. Further, the parameter is also used as a flag to indicate whether the `PAGES` object is to perform a printing or paginating operation (previewing does not exist on the Series 3). A `NULL` value is used to indicate that the `PAGES` object is to perform a pagination rather than a printing operation. The `PAGES ao_init` method is prototyped as:

```
VOID ao_init (PAGES_INIT *in, VOID *port, PAGES_PARAMS *par);
```

2. On the Series 3a, the handle of the `PRINTER` object is set into the `spare1` property of the application manager by the `pr_init` method. On the Series 3, this is not done. Instead, the handle can be found in the `printer` property of `HWIM`'s `WSERV` active object.
3. On the Series 3a, the `PRINTER` class method `pr_port_data` takes three parameters, the last of which can take the value `TRUE` or `FALSE`. On the Series 3, however, this final parameter does not exist and has the effect that the method can only return printer port information as set by the user of the `PRINTER` object (by an earlier call to the `pr_set_port_type`).
4. On the Series 3a, the `pr_sense_model` method searches for a `.wdr` file of the same name as that held in property or the environment variable `P$M`. If the file cannot be found, all the `LOC::` drives are searched. If the file still cannot be found, the `ROM` is searched and, only if it cannot be found here, is the default file `ROM::BJ.WDR` and model number zero used.

The search behaviour on the Series 3 differs slightly. Here, if the file cannot be found, drives `A:`, `B:`, and `M:` are searched before using the default file `ROM::BJ.WDR` and model number zero.

CHAPTER 5

THE PRINT PREVIEW CLASS

The Print Preview class, or the PRVPDR class, to give its correct name, is a subclass of PDR that provides the necessary behaviour to build a preview of a document.

Previewing allows us to see up to four pages (up to two in landscape mode) of a document at time, in "miniature", to get an overall view of the layout of the text and to see how it would look when printed. The pages are displayed on the screen.

A great deal of the property and behaviour of PRVPDR is provided by the base class PDR. However, a number of methods are replaced by PRVPDR, in particular, those dealing with the initialisation and "printing" of a document.

The class does not perform I/O to a real physical printer; instead, requests such as printing text, starting a new line and moving the print head are converted into drawing actions on a bitmap and manipulating the position within the bitmap where drawing is to be done.

In effect, each page is drawn to a bitmap and each bitmap is compressed and placed into a data segment.

This class does *not* contain behaviour to display the previewed document. The application user interface is normally responsible for decompressing the bitmaps and displaying the previewed pages.

It should be noted that although PRVPDR is a subclass of PDR, it is *not* loaded from a separate DYLIB.

This class is *not* available on the on the Series 3.

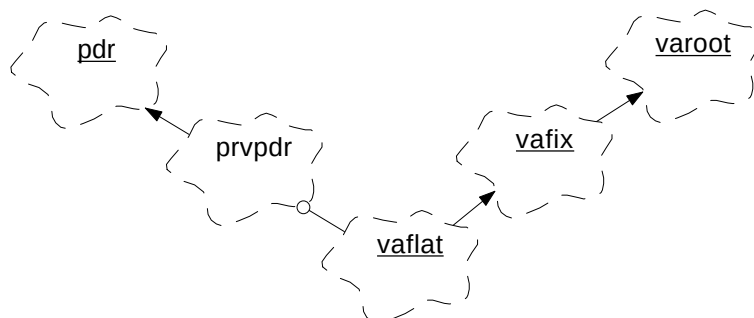
Precursors

An understanding of the print preview class will be helped by a knowledge of:

- the p_enter and p_leave error handling services
- the *Graphics Output* chapter of the *Window Server Reference*
- *The Document Printing Classes* chapter of this manual
- the OLIB variable array class, VAFLAT

Class diagram

The following diagram covers the relationship between the PRVPDR class and other classes and is discussed in detail in this chapter. The underlined classes are either discussed in another chapter of this manual or they refer to OLIB classes in which case they are *all* discussed in the *OLIB Reference* manual.



PRVPDR

PDR	PRVPDR
par mode typfix fhix style lheight a trans_rid outlen skipy	pChWidths SpaceWidth FontY TwipsRound PageHeight Scale Round Pos hPrvDone mPrvDone Bmp SegHandle SegPos SegHeight SegSize pArray pBitRow pLastRow pRowRec
destroy ~~pdr_init~~ ~~pdr_print~~ pdr_add_command ~~pdr_destroy~~ ~~pdr_start~~ pdr_end ~~pdr_page~~ pdr_text pdr_line pdr_right ~~pdr_font~~ pdr_style	pdr_init pdr_print pdr_destroy pdr_start pdr_page pdr_font

Class definition

The PRVPDR class subclasses the PDR class and is defined in the sub-category file *prvpdr.cl* (with generated header file *prvpdr.g*).

```

CLASS prvpdr pdr
{
  REPLACE pdr_init           Allocate larger buffer
  REPLACE pdr_print
  REPLACE pdr_destroy        To get around bug in pdr class
  REPLACE pdr_start          Start printing
  REPLACE pdr_page
  REPLACE pdr_font

  TYPES
  {
    typedef struct
    {
      INT      Id;           ID of bitmap
      HANDLE   SegHandle;    handle of bitmap segment
      UPOINT   Size;
      UWORD    ByteWidth;
      UWORD    BitWidth;    width of used bitmap, use this for scaling
    } PRV_BITMAP;
  }

```

```

typedef struct
{
    UWORD typeface;
    UWORD height;
    UWORD style;
} FONT_DESC;

typedef struct
{
    UBYTE Type;
    UBYTE Length;
} BMP_RASTER_TL;

typedef struct
{
    BMP_RASTER_TL tl;
    UBYTE Data[2];
} BMP_RASTER_ROW_REC;
}

PROPERTY
{
    UBYTE    *pChWidths;           character widths for current font
    UWORD    SpaceWidth;           width of space in current font
    UWORD    FontY;                pixel height of current font
    UWORD    TwipsRound;           used for rounding in twips conversion
    UWORD    PageHeight;
    UPOINT    Scale;               x and y scales (printer units to bitmap units)
    UPOINT    Round;               used to round unit conversions
    UPOINT    Pos;                 x and y position in current page bitmap
    VOID      *hPrvDone;           Callback handle for %done & completion
    WORD      mPrvDone;            Callback method for %done & completion
    PRV_BITMAP Bmp;
    HANDLE    SegHandle;           segment to save drawing to
    LONG      SegPos;              current position in segment
    INT       SegHeight;           height of bitmap segment (lines)
    INT       SegSize;             size of segment (in paragraphs)
    PR_ROOT   *pArray;             varray of page positions in segment
    UBYTE     *pBitRow;            current row from bitmap
    UBYTE     *pLastRow;           previous row from bitmap
    BMP_RASTER_ROW_REC *pRowRec; compressed data from preview segment
}
}

```

Property

<code>prvpdr.pChWidths</code>	The address of the font width table for the current typeface, font height and style.
<code>prvpdr.SpaceWidth</code>	The width of the space character in the current font, in <i>printer units</i>
<code>prvpdr.FontY</code>	The height of the current font, in <i>pixels</i> .
<code>prvpdr.TwipsRound</code>	This is a horizontal and vertical correction factor used in the conversion of twips to pixels in the <i>vertical</i> direction. <code>prvpdr.TwipsRound</code> is the ratio of the number of vertical bits in the bitmap (the number of 'lines' in the bitmap) to the height of the page to be displayed in twips. i.e. <code>prvpdr.Bmp.Size.y / prvpdr.PageHeight</code>.
<code>prvpdr.PageHeight</code>	The height of a page to be displayed in <i>twips</i>. The actual value contained in this property depends on the display mode. In landscape mode, this value is set to the width of the page; in portrait mode, this value is set to the height of the page.

prvpdr.Scale	This is a horizontal and vertical scaling factor used in the conversion of printer units to pixels. prvpdr.Scale.x gives the number of horizontal <i>print head movements</i> needed to print the full width of the page; prvpdr.Scale.y gives the number of vertical <i>print head movements</i> needed to print the full height of the page.										
prvpdr.Round	This is a horizontal and vertical correction factor used in the conversion of printer units to pixels. prvpdr.Round.x is the ratio of the horizontal scaling factor to the number of horizontal bits needed to draw a line (prvpdr.Bmp.Bitwidth); prvpdr.Round.y is the ratio of the vertical scaling factor to the number of vertical bits available in the bitmap (prvpdr.Bmp.Size.y).										
prvpdr.Pos	The x and y position in the current page bitmap measured in <i>printer units</i>										
prvpdr.hPrvDone	The handle of the object providing the <i>done</i> callback method.										
prvpdr.mPrvDone	The method number of the <i>done</i> callback method. For the general specification of this method, see the prntprv_mdome method in the description of the PRNTPRV mixin class in this chapter. N.B. The <i>done</i> call-back method here is quite distinct from the PAGES <i>done</i> call-back method referred to in <i>The Document Printing Classes</i> chapter.										
prvpdr.Bmp	This is a data structure of type PRV_BITMAP and contains information relating to the bitmap used for drawing a representation of the page. The individual members of this structure are shown below. N.B. the Size, Bytewidth and Bitwidth are shown in a different order to that in the structure. <table><tr><td>Id</td><td>The ID of the bitmap as returned by a call to the Window Server function gCreateBit</td></tr><tr><td>SegHandle</td><td>The handle of the bitmap segment as returned by the Plib function p_sgopen</td></tr><tr><td>Bitwidth</td><td> The number of horizontal bits needed to draw a single line so that it fits into the application's window and the ratio of this value to the height of the page measured in pixels is the same as the ratio of the width to the height of the page measured in twips. This value is used to calculate the value of prvpdr.round.x, described above. </td></tr><tr><td>Bytewidth</td><td>The number of bytes needed to accommodate a single line of the bitmap. It is the value of (Size.x/8) and assumes that Size.x is an exact multiple of eight.</td></tr><tr><td>Size</td><td> The dimensions of the bitmap in <i>pixels</i>. The x component is the value of Bitwidth rounded <i>up</i> to an exact multiple of 8. The y component is normally determined by the height of the application's window. </td></tr></table>	Id	The ID of the bitmap as returned by a call to the Window Server function gCreateBit	SegHandle	The handle of the bitmap segment as returned by the Plib function p_sgopen	Bitwidth	The number of horizontal bits needed to draw a single line so that it fits into the application's window and the ratio of this value to the height of the page measured in pixels is the same as the ratio of the width to the height of the page measured in twips. This value is used to calculate the value of prvpdr.round.x, described above.	Bytewidth	The number of bytes needed to accommodate a single line of the bitmap. It is the value of (Size.x/8) and assumes that Size.x is an exact multiple of eight.	Size	The dimensions of the bitmap in <i>pixels</i>. The x component is the value of Bitwidth rounded <i>up</i> to an exact multiple of 8. The y component is normally determined by the height of the application's window.
Id	The ID of the bitmap as returned by a call to the Window Server function gCreateBit										
SegHandle	The handle of the bitmap segment as returned by the Plib function p_sgopen										
Bitwidth	The number of horizontal bits needed to draw a single line so that it fits into the application's window and the ratio of this value to the height of the page measured in pixels is the same as the ratio of the width to the height of the page measured in twips. This value is used to calculate the value of prvpdr.round.x, described above.										
Bytewidth	The number of bytes needed to accommodate a single line of the bitmap. It is the value of (Size.x/8) and assumes that Size.x is an exact multiple of eight.										
Size	The dimensions of the bitmap in <i>pixels</i>. The x component is the value of Bitwidth rounded <i>up</i> to an exact multiple of 8. The y component is normally determined by the height of the application's window.										
prvpdr.SegHandle	The handle of an external data segment into which are copied the compressed bitmaps containing the drawn representation of each document page. This is also referred to as the preview data segment. The segment is created by an instance of the PRINTER class and the handle is passed to this instance of PRVPDR in a call to the pdr_init initialisation method.										
prvpdr.SegPos	The current position within the preview data segment measured in bytes.										
prvpdr.SegHeight	The height of the bitmap segment. Effectively, this represents the number of <i>lines</i> of bits available for drawing.										

<code>prvpdr.SegSize</code>	The current size of the preview data segment, in paragraphs (i.e. units of sixteen bytes).
<code>prvpdr.pArray</code>	The handle of a VAFLAT object. The array is used to hold a series of values which give the position of consecutive compressed bitmaps within the preview data segment. It is designed to hold entries (records) which are the length of a LONG 'C' data type and has a granularity of 16 entries (records). The instance of VAFLAT is initialised by the application before the creation of this instance of PRVDR and contains a single entry (record) holding a zero value.
<code>prvpdr.pBitRow</code>	The address of a buffer to contain a copy of the <i>current</i> row from the bitmap. The buffer itself is <code>prvpdr.Bmp.Bytewidth</code> bytes long. This buffer is used by the <code>pdr_page</code> method during bitmap compression.
<code>prvpdr.pLastRow</code>	The address of a buffer to contain a copy of the <i>previous</i> row from the bitmap. The buffer itself is <code>prvpdr.Bmp.Bytewidth</code> bytes long. This buffer is used by the <code>pdr_page</code> method during bitmap compression.
<code>prvpdr.pRowRec</code>	The address of a buffer to contain compressed data from the bitmap. The buffer itself is <code>(prvpdr.Bmp.Bytewidth plus the length of structure BMP_RASTER_ROW_REC)</code> bytes long. This buffer is used by the <code>pdr_page</code> method during bitmap compression.

PRVPDR methods

PDR_INIT

Initialise

VOID `pdr_init(PDR_INIT *pPar)`

Initialise the instance of PRVPDR. This method replaces the subclass `pdr_init` method.

The method takes a single parameter: `pPar` points to a data structure of type `PDR_INIT` which contains information required to initialise the instance.

The method starts by copying the entire content of `*pPar` into the property `pdr.par`.

A `PR_PREVIEW_DATA` message is then sent to the `PRINTER` object to fetch the address of the `PRINTER` property `printer.prv`. This is a data structure of type `PREVIEW_DATA` and contains information required for the preview operation. For more detail on the content of this structure, see the `PRINTER` class `pr_preview_start` method in *The Document Printing Classes*.

Note that the assumption is made that a copy of the handle of the `PRINTER` object is contained in the application manager's `spare1` property. A `FORM PRINTER` object, as part of its initialisation process, always inserts a copy of its own handle into the application manager's `spare1` property for the convenience of a large number of methods within a variety of classes.

A number of items of property are set by copying corresponding members from the `PREVIEW_DATA` structure: `prvpdr.SegHandle`, `prvpdr.pArray`, `prvpdr.hPrvDone`, `prvpdr.mPrvDone` plus the `Size` and `Bitwidth` members of `prvpdr.Bmp`. The `Bytewidth` member of `prvpdr.Bmp` is set to the value of `prvpdr.Bmp.Size.x` divided by eight.

A memory cell, large enough to contain three buffers, is allocated and added to the cleanup list. The memory cell is partitioned as follows:

- the address of the first `prvpdr.Bmp.Bytewidth` bytes is set into the property `prvpdr.pBitRow`; this buffer is `prvpdr.Bmp.Bytewidth` bytes long and will contain the *current* bitmap row.
- the address of the second `prvpdr.Bmp.Bytewidth` bytes is set into the property `prvpdr.pLastRow`; this buffer is `prvpdr.Bmp.Bytewidth` bytes long and will contain the *previous* bitmap row.
- the address of the third `prvpdr.Bmp.Bytewidth` bytes is set into the property `prvpdr.pRowRec`; this buffer is `(prvpdr.Bmp.Bytewidth + length of a BMP_RASTER_ROW_REC structure)` bytes long and will contain the bitmap raster row record.

A bitmap is created in its own memory segment, using the `gCreateBit Window Server` function, and the returned bitmap ID is set into `prvpdr.Bmp.Id`; the size of the bitmap is determined by the value of `prvpdr.Bmp.Size`. The bitmap memory segment is then opened, using the `p_sgopen Plib` function, and the returned handle to the opened memory segment is set into `prvpdr.Bmp.SegHandle`. Note that if `p_sgopen` returns an error condition, the bitmap is explicitly freed (using `wFree`) and `p_leave` called, passing it the error code returned by `p_sgopen`.

A `PR_GET_PARAMS` message is sent to the `PRINTER` object to get the address of the printer parameters. The printer parameters are held in a `PRINTER_PARAMS` data structure in `PRINTER`'s property `printer.p`.

The horizontal and vertical scaling factors and rounding values are calculated and set into the properties: `prvpdr.Scale`, `prvpdr.Round` and `prvpdr.TwipsRound`. These values will be used to calculate the number of bits required to represent items of text in the bitmap representation of the document. Specifically, they are used to convert twips and printer units into numbers of pixels. Note that, in calculating these values, account is taken of whether the document is in landscape or portrait mode.

Finally, the memory cell containing the three buffers is removed from the cleanup list.

PDR_PRINT

Interpret print command

```
INT pdr_print(WDR_PRINT *pr, UBYTE **ppbuf)
```

Interpret a print element and manipulate, or draw, to the bitmap.

The method takes two parameters:

- `pr` contains the address of a print element; this is a data structure of type `WDR_PRINT`. See the description of the `PDR` superclass `pdr_print` method in *The Document Printing Classes* chapter in this manual for more information on the `WDR_PRINT` structure. The *Printing* chapter of the *Object Oriented Programming Guide* also contains some useful background information.
- `ppbuf` is not used by this method but is included in the method prototype for compatibility with the superclass `pdr_print` method. When *calling* this method, the parameter can be `NULL`.

In general, the method uses the information in the print element to print (i.e. to draw) to the bitmap and to manipulate the position within the bitmap where drawing is to be done. A print element will also indicate where page breaks occur and the start and end of the printing process.

The detailed working of this method is driven by the settings of the `flags` member of the print element. `flags` can contain an `ored` combination of values. Depending on the individual flags set, the method proceeds as follows:

WDR_PRINT_START The method sends a `PDR_START` message to *this* instance of `PRVPDR` to prepare the printing process.

WDR_PRINT_PAGE This flag indicates a page break request and causes the method to send a `PDR_PAGE` message to this instance of `PRVPDR` to copy (and compress) the bitmap of the current page to the preview data segment.

The code is constructed such that if the `pdr_page` method returns with an error, the `done` callback method is called to inform the application of the error and is followed by a call to `p_leave` specifying the returned error code.

WDR_PRINT_LINE This flag indicates a line break.

The sum of the `down` and `height` members of the print element indicates the amount by which the print position must be moved *downwards*.

The value of the `indent` member of the print element indicates the initial print position relative to the left hand edge of the page.

With this flag set, the *vertical* print position within the bitmap, as defined by the value of `prvpdr.Pos.y`, is *adjusted* by the sum of the `down` and `height` members of the print element; the *horizontal* print position within the bitmap, as defined by the value of `prvpdr.Pos.x`, is *set* to the value of the `indent` member of the print element provided that this is greater than zero. A zero or negative value of `indent` causes `prvpdr.Pos.x` to be set to zero.

- WDR_PRINT_FONT** This flag indicates a request to set a font and causes the method to send a `PDR_FONT` message to *this* instance of `PRVPDR` to set the typeface, font height and style as defined by the `typf`, `fheight` and `style` members of the print element respectively.
- The code is constructed such that if the `pdr_font` method returns with an error, the `done` callback method is called to inform the application of the error and is followed by a call to `p_leave` specifying the returned error code.
- WDR_PRINT_RIGHT** This flag indicates a request to move the print position to the right.
- The *horizontal* print position within the bitmap, as defined by the value of `prvpdr.Pos.x`, is *incremented* by the value of the `right` member of the print element provided that its value is greater than zero. If the value of `right` is zero or negative, no adjustment is made.
- WDR_PRINT_TEXT** This flag indicates that there is text to be printed.
- The address of a buffer containing the text to be printed (i.e. `drawn`) to the bitmap is contained in the `buf` member of the print element. The length of the text to be printed is contained in the `len` member.
- No attempt is made to draw actual scaled characters because the resolution of the screen is not sufficiently fine. Instead, filled rectangles are drawn for every word of text, scaled according to the size of the word and the font height. A temporary graphics context is used for the drawing.
- WDR_PRINT_END** The method terminates the printing (i.e. drawing) process by sending a `PDR_PAGE` message to this instance of `PRVPDR` to ensure that the bitmap of the *final* page is copied (and compressed) to the preview data segment.
- If the called `pdr_page` method completes successfully, it (`pdr_print`) calls the *done* call-back method, passing a value of `PAGES_DONE_DOC`, to inform the application of the event.
- If the called `pdr_page` method fails with an un-recoverable error, it (`pdr_page`) calls the *done* call-back method, passing a value of `PAGES_DONE_ERROR` before calling `p_leave` to propagate the error.

The method always returns a zero value.

PDR_DESTROY

Destroy

`VOID pdr_destroy(VOID)`

Destroy the instance of `PRVPDR`.

The memory cell from which the three buffers, allocated in `pdr_init`, and anchored in the properties `prvpdr.pBitRow`, `prvpdr.pLastRow` and `prvpdr.pRowRec`, is freed.

The bitmap is freed using the window server function `wFree` and the bitmap memory segment is freed using the `Plib` function `p_sgclose`.

PDR_START

Start printing (drawing)

`VOID pdr_start(VOID)`

Prepare to start the printing (i.e. drawing) process.

The method clears the bitmap ready for drawing by doing the following:

- Create a temporary graphics context, specifying the bitmap as the drawable entity.
- Clear the pixels in the whole bitmap by calling the window server function `gClrRect`.
- Free the temporary graphics context.

The property `pdr.typfix` is set to -1; this guarantees that a font width table will be loaded in. See the `pdr_font` method.

An internal call is made to code which implements the `pdr_font` method to set the default typeface, a font height of 120 twips and the normal style. This code runs under the protection of `p_enter`. If an un-recoverable error occurs within this code, the *done* call-back method will be called, passing a value of `PAGES_DONE_ERROR` before calling `p_leave` to propagate the error.

PDR_PAGE

Start a new page

```
INT pdr_page(VOID)
```

Copy (and compress) the bitmap of the *current* page to the preview data segment.

The preview data segment is the external data segment allocated by the `PRINTER` object and whose address is passed to `PRVPDR` during initialisation (see the `pdr_init`) method.

The method calls the window server function `wFlush` to ensure that all drawing to the bitmap is complete.

The method then takes the bitmap representing the current page and compresses the data using raster graphics compression techniques and copies the compressed data to the preview data segment.

The bitmap itself is then cleared in exactly the same way as described in the `pdr_start` method, ready for another page to be drawn.

If a full page is successfully drawn to the bitmap, the method informs the application by calling the *done* call-back method, passing a value of `PAGES_DONE_PAGE`.

Note that the code implementing this method runs under the protection of `p_enter`. If an un-recoverable error occurs within this code, the *done* call-back method will be called, passing a value of `PAGES_DONE_ERROR` before calling `p_leave` to propagate the error.

PDR_FONT

Set the font

```
INT pdr_font(INT typeface, INT height, INT style)
```

Set the typeface, font height and style.

The method takes three parameters:

- `typeface` specifies the *number* of the required typeface.
- `height` specifies the height, in twips, of the required font.
- `style` specifies the required style and can be an ORED combination of: `WDR_STYLE_NORMAL`, `WDR_STYLE_UNDERLINE`, `WDR_STYLE_BOLD`, `WDR_STYLE_ITALIC`, `WDR_STYLE_SUPER` and `WDR_STYLE_SUB`. See the description of the `pdr_style` method of the `PDR` class in *The Document Printing Classes* chapter of this manual.

The method sends a `WDR_SEARCH_TYPEFACE` message to the associated `WDR` object to retrieve the typeface index. The typeface index is the index of the entry within the array of pointers to `WDR_TYPEFACE` structures which represents the typeface with number `typf`. In effect, it identifies the location of the information on the required typeface.

The method then sends a `WDR_SEARCH_HEIGHT` message to retrieve the font height index. This is the index of the entry within the array of `WDR_FONT` structures which most closely represents the font with height `height`. In effect, it identifies the location of the information on the required font.

See the description of the `WDR` class in *The Document Printing Classes* chapter of this manual for more information on typefaces and fonts and the data structures representing them.

The font height is converted from twips to pixels and the resulting value set into the property `prvpdr.FontY`.

If the required typeface *or* the required font height *or* the required style differs from the existing ones (as recorded in the subclass property: `pdr.typfix`, `pdr.fhix` and `pdr.style` respectively), then a `WDR_GET_WIDTH_TABLE` message is sent to get the address of the new font width table. This address is set into the property `prvpdr.pChwidths`; the width of the blank character is set into the property `prvpdr.SpaceWidth`.

Note that the code implementing this method runs under the protection of `p_enter`. If an un-recoverable error occurs within this code, the *done* call-back method will be called, passing a value of `PAGES_DONE_ERROR` before calling `p_leave` to propagate the error.

The PRNTPRV mixin class

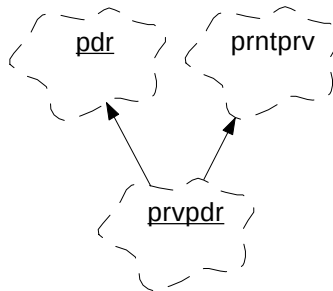
PRNTPRV
mdone

The PRNTPRV mixin class provides the *general specification* for the call-back method(s) that may be called by the PRVPDR class. The call-back method `mdone` is also referred to as the *done* method. For a general discussion on call-back methods and mixin classes, see the *Introduction* chapter in this manual.

The PRNTPRV class does not appear in the FORM library and an instance of PRNTPRV will never be created. The FORM library does *not* supply a class which can provide the required *done* call-back method; this is normally supplied by the application.

Class diagram

The following class diagram formally illustrates the relationship between the PRNTPRV mixin class and the PRVPDR class. This diagram shows both the PDR class and the PRNTPRV class in order to emphasise the multiple inheritance aspect of mixin classes.



Class definition

```

CLASS prntprv root
{
  DEFER mdone          report status
}
  
```

Property

None.

PRNTPRV call-back methods

PRNTPRV_MDONE

Handle status messages

```
INT prntprv_mdone(INT event);
```

This method is also referred to as the *done* method in this chapter.

This call-back method, supplied by the application, provides the mechanism by which a PRVPDR object can keep an application informed of the current status of the previewing operation. It is very similar, in some respects, to the PAGELAY *done* call-back method. However, unlike the PAGELAY *done* call-back method, this method is only passed the *status* of the PRVPDR object.

The content of the method is application dependent; however, it should take note of the information PRVPDR passes to it.

PRVPDR passes a single parameter:

- `event` indicates the status of the previewing operation and can take one of the following values:

PAGES_DONE_PAGE	This is set when a full page has been successfully drawn to the bitmap, the bitmap has been compressed and copied to the data segment and the bitmap itself has been cleared and other property reset ready to build the next page. Any value returned by this method is ignored by PRVPDR.
PAGES_DONE_ERROR	This is set when an un-recoverable error occurs. Any value returned by this method is ignored by PRVPDR.
PAGES_DONE_DOC	This is set when the previewing operation is complete. The last page will have been successfully drawn to the bitmap and the bitmap itself copied to the data segment. Any value returned by this method is ignored by PRVPDR.

Although this method is prototyped to return an INT value, a PRVPDR object makes no use of it. As a useful convention, it is suggested that this method return a TRUE value.

CHAPTER 6

THE CALENDAR IMAGE CLASS

Precursors

An understanding of the CALIMG class will be helped by a knowledge of:

- the graphical calendar display in the Series 3a Agenda application.

Note that on the Series 3a it is more convenient to use the CALWIN class rather than the CALIMG class. See the *Calendar Classes* chapter of the *HWIM Reference* manual.

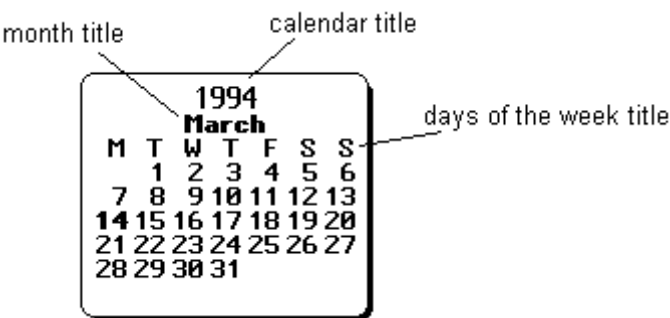
The CALIMG class simply subclasses ROOT and uses no other classes as components.

CALIMG

CALIMG	
rect	bwidth
title_rect	filler
crs	dayNameAbbrev
day	deftitle
startOfWeek	title
thisyear	emphasised
thismth	img
thisday	bmid
ystart	
destroy	ci_adjust_date
ci_init	ci_redraw
ci_set_title	ci_sense
ci_emphasise	ci_view
ci_move_cursor	ci_pos_mxy
ci_goto_date	ci_today_changed

The CALIMG class supports a graphical calendar display as used by the Series 3a Agenda application. The calendar display provides a convenient method for the user to either select a date or determine the day of the week for a given date. Note that the user is responsible for passing the ID of a suitable window for the calendar image.

The various titles in the calendar are indicated in the following picture:



Note that the borders are not drawn by the CALIMG object and thus must be explicitly drawn by the application.

The calendar supports multiple rows and columns of months as illustrated in the following picture:

1994													
February							March						
M	T	W	T	F	S	S	M	T	W	T	F	S	S
	1	2	3	4	5	6		1	2	3	4	5	6
7	8	9	10	11	12	13	7	8	9	10	11	12	13
14	15	16	17	18	19	20	14	15	16	17	18	19	20
21	22	23	24	25	26	27	21	22	23	24	25	26	27
28							28	29	30	31			
April													
M	T	W	T	F	S	S							
					1	2							
4	5	6	7	8	9	10							
11	12	13	14	15	16	17							
18	19	20	21	22	23	24							
25	26	27	28	29	30								

On the Series 3a the practical limit to the maximum number of rows is two whilst the limit on the number of columns is six.

Note that the edges of the calendar are notionally mapped onto the preceding and following months. Thus in the above example moving the cursor upwards eventually scrolls the display to the preceding month.

Class definition

The CALIMG class subclasses ROOT and is defined in the sub-category file *calimg.cl* (with generated header file *calimg.g*).

```

CLASS calimg root
{
  REPLACE destroy
  ADD ci_init
  ADD ci_set_title
  ADD ci_emphasise
  ADD ci_move_cursor
  ADD ci_goto_date
  ADD ci_adjust_date
  ADD ci_redraw
  ADD ci_sense
  ADD ci_view
  ADD ci_pos_mxy
  ADD ci_today_changed
}

CONSTANTS
{
  CI_MAX_MONTH          12
  CI_GRIDY              TRUE    /* get month row */
  CI_GRIDX              FALSE   /* get month col */

  CALIMG_CURSOR_NO_FLASH 0x01
  CALIMG_DOW_EVERY_ROW   0x02
  CALIMG_FONT_DATA_KNOWN 0x04

  CALIMG_LEFT            0x00 /* physically go to day on the left */
  CALIMG_HOME            0x01 /* goto leftmost day & month of row */
  CALIMG_PREV_DAY        0x02 /* decrement by day */
  CALIMG_PREV_MONTH      0x03 /* decrement by month */

  CALIMG_RIGHT           0x04 /* physically go to day on the right */
  CALIMG_END             0x05 /* goto most right day & month of row */
  CALIMG_NEXT_DAY        0x06 /* increment by day */
  CALIMG_NEXT_MONTH      0x07 /* increment by month */

  CALIMG_UP              0x08 /* physically go to day above */
  CALIMG_PAGEUP          0x09 /* goto previous page maintain x,y in month */
  CALIMG_PREV_WEEK       0x0A /* decrement by week */
  CALIMG_PREV_YEAR       0x0B /* decrement by year */

  CALIMG_DOWN            0x0C /* physically go to day below */
  CALIMG_PAGEDN          0x0D /* goto next page,maintain x,y in month */
  CALIMG_NEXT_WEEK       0x0E /* increment by week */
  CALIMG_NEXT_YEAR       0x0F /* increment by year */
}

```

```

CALIMG_GOTO_TODAY      0x10
CALIMG_WEST            0x01
CALIMG_EAST            0x02
CALIMG_NORTH           0x03
CALIMG_SOUTH           0x04

CALIMG_ADJ_DAY         0x01
CALIMG_ADJ_MONTH       0x02
CALIMG_ADJ_YEAR        0x04
}

```

TYPES

```

{
typedef struct
{
    UBYTE title_ascent; ascent for title
    UBYTE title_lht;    line height for title
    UBYTE dow_ascent;   ascent for day of week
    UBYTE dow_lht;      line height for day of week
    UBYTE mth_ascent;   ascent for month
    UBYTE mth_lht;      line height for month
    UBYTE day_ascent;   ascent for day/date
    UBYTE day_lht;      line height for day/date
    UBYTE daygapx;      gap between 2 dates
    UBYTE daywidth;     2 numeric width characters ie width for 1 day
} CI_EXT_FONT;         Extended font information

typedef struct
{
    WORD fid;           font ID
    UWORD style;        font style whether bold,italics,etc
    UWORD leading;       leading below text
} CI_FONT_DATA;

typedef struct
{
    UWORD wid;          window ID
    UWORD width;        calimg window width
    P_POINT tl;         tl.x=left & right gutters; tl.y=top & bottom gutters
    UBYTE mrow;         number of rows of months to display
    UBYTE mcol;         number of cols of months to display
    UWORD flags;
    CI_FONT_DATA title; font info for main top line title
    CI_FONT_DATA month; font info for month title
    CI_FONT_DATA dow;   font info for day of week
    CI_FONT_DATA day;   font info for the days itself
    UWORD daygap;       CHAR NUMBER to use as horizontal gap between 2 days
    UBYTE mthgapx;      no of pixels of horizontal gap between 2 months
    UBYTE hscrlm;       granularity for scrolling horizontally by month
    UWORD startm;        month number to display as first month, ie tl month
    ULONG days;          days field of daysec struct ie days since 1/1/1900
    CI_EXT_FONT font;    extended font info, defaults filled in by ci_init()
} IN_CALIMG;

typedef struct
{
    UBYTE year;         Year number since 1900
    UBYTE month;        month number 0 to 11
    P_POINT pos;        x,y position within a month
} CI_CURSOR;

typedef struct
{
    WORD year;
    WORD month;
    WORD day;
} CI_DATE;
}

```

```
PROPERTY
{
  P_RECT rect[12];      region to draw for each month, tl==tl of month title line
  P_RECT title_rect;    use to print title
  CI_CURSOR crs;        current cursor pos x,y & month & year
  BYTE day;             day number in month, 0 to 30, negative days are possible
  UBYTE startOfWeek;
  UBYTE thisyear;        as opposed to last year & next year
  UBYTE thismth;         as opposed to last month & next month
  UBYTE thisday;         TODAY
  UBYTE ystart;          year of the tl month on display
  UBYTE bwidth;          2*char width of bold font for today
  UBYTE filler;
  TEXT dayNameAbbrev[7]; 1st letters of days of week, starting at startOfWeek
  TEXT deftitle[3];      default title
  TEXT *title;
  WORD emphasised;        keeping track of whether it is already emphasised
  IN_CALIMG img;
  INT bmid;               bitmap ID for all days in a month
}
```

Property

calimg.rect	defines the drawing region for each month in the calendar view - used internally.
calimg.title_rect	defines the drawing region for the calendar title - used internally.
calimg.crs	the cursor coordinates in the current month - (0,2) for example defines a cursor at the intersection of column one and row three.
calimg.day	the day number of the current day in the range 0 to 30 where 0 is the first of the month.
calimg.startOfWeek	the day number (in the range 0 to 6 where 0 is Monday) of the first day of the week. This is usually 0 on UK machines.
calimg.thisyear	the year number for today - where today is determined by a call to p_date - in the range 0 to 254 inclusive where 0 is 1900.
calimg.thismth	the month number for today in the range 0 to 11 inclusive where 0 is January.
calimg.thisday	the day number for today in the range 0 to 30 inclusive where 0 is the first of the month.
calimg.ystart	the year number of the first month in the calendar view - the first month is in the top left corner.
calimg.bwidth	two times the width of a bold numeric character in the day font - this is the width of the day symbol when the day is 'today'.
calimg.filler	used internally.
calimg.dayNameAbbrev	contains the first letter of each day with the first element corresponding to the day in calimg.startOfWeek. On UK machines the first element usually contains 'M'.
calimg.deftitle	the default format string for the calendar title - i.e. "%Y".
calimg.title	the format string for the calendar title - NULL indicates that the format string is to be read from calimg.deftitle. All occurrences of %Y and %y are replaced with the appropriate year(s). All occurrences of %% are replaced with %. All other occurrences of % are ignored. Thus "A& silly %% example %Y title" generates "A silly % example 1994 title" in 1994 and "A silly % example 1994-1995 title" in 1994-1995.
calimg.emphasised	TRUE if the calendar is emphasised, and FALSE otherwise.
calimg.img	data passed to the ci_init method: see the description of the ci_init method for details of the fields.
calimg.bmid	ID of a bitmap - the bitmap is for internal use only.

CALIMG methods

DESTROY

Destroy the calendar

VOID destroy(VOID)

Destroy the CALIMG instance.

If `calimg.emphasised` is TRUE, the method erases the text cursor.

If `calimg.title` is non-zero, the method frees the cell with address `calimg.title`.

The method then frees the bitmap with ID `calimg.bmid` and supersends a DESTROY message.

CI_INIT

Initialise the calendar

VOID ci_init(IN_CALIMG *init)

Initialise the instance of CALIMG according to the content of the IN_CALIMG structure pointed to by `init`.

The IN_CALIMG structure is defined as follows:

```
typedef struct
{
    UWORD wid;
    UWORD width;
    P_POINT tl;
    UBYTE mrow;
    UBYTE mcol;
    UWORD flags;
    CI_FONT_DATA title;
    CI_FONT_DATA month;
    CI_FONT_DATA dow;
    CI_FONT_DATA day;
    UWORD daygap;
    UBYTE mthgapx;
    UBYTE hscrlm;
    UWORD startm;
    ULONG days;
    CI_EXT_FONT font;
} IN_CALIMG;
```

The significance of the members of the IN_CALIMG structure is as follows:

<code>wid</code>	specifies the ID of the window which provides the drawing region.
<code>width</code>	specifies the width in pixels of the drawing region.
<code>tl</code>	specifies the gutter dimensions. the <code>x</code> member specifies the width in pixels of the left and right gutters and must not be less than 3. the <code>y</code> member specifies the height in pixels of the top and bottom gutters and must not be less than 3.
<code>mrow</code>	specifies the number of rows in the calendar view.
<code>mcol</code>	specifies the number of columns in the calendar view.
<code>flags</code>	an Ored combination of flags: see below for the available flags.
<code>title</code>	specifies the font characteristics of the main title. A description of the CI_FONT_INFO structure may be found below.
<code>month</code>	specifies the font characteristics of the month title. A description of the CI_FONT_INFO structure may be found below.
<code>dow</code>	specifies the font characteristics of the days of the week title. A description of the CI_FONT_INFO structure may be found below.

day	specifies the font characteristics of the day of the month numbers. A description of the <code>CI_FONT_INFO</code> structure may be found below.
daygap	specifies a character, the width of which in the day number font and style, defines the spacing between day numbers.
mthgapx	specifies the horizontal pixel separation between adjacent months.
hscr_lm	specifies the default granularity for scrolling horizontally by month: should be divisible by the total number of months in the calendar view.
startm	specifies the month number of the first month that appears in the top left corner of the calendar.
days	specifies the current date expressed as the number of days elapsed since 1/1/1900.
font	specifies additional font information for the main title, month title, days of the week title and the day of the month numbers. This information includes the line heights and line ascents as described below.

The `flags` member of the `CI_INIT` structure may contain an Ored combination of the following flags:

<code>CALIMG_CURSOR_NO_FLASH</code>	draw a non-flashing cursor indicating the current day. The default is a flashing cursor.
<code>CALIMG_DOW EVERY_ROW</code>	specifies that the days of the week title is to be included only in the first row of months. This flag must be set on the Series 3a when two rows of months are required otherwise the calendar view will be too large for the screen.
<code>FONT_DATA_KNOWN</code>	indicates that the line heights and ascents are specified in <code>init->font</code> . Otherwise the method overwrites <code>font</code> with default values.

The `CI_EXT_FONT` structure specifies the line heights and line ascents in the calendar view. It is defined as follows:

```
typedef struct
{
    UBYTE title_ascent;
    UBYTE title_lht;
    UBYTE dow_ascent;
    UBYTE dow_lht;
    UBYTE mth_ascent;
    UBYTE mth_lht;
    UBYTE day_ascent;
    UBYTE day_lht;
    UBYTE daygapx;
    UBYTE daywidth;
} CI_EXT_FONT;
```

The significance of the members of the `CI_EXT_FONT` structure is as follows:

<code>title_ascent</code>	specifies the ascent in pixels of the calendar title.
<code>title_lht</code>	specifies the height in pixels of the calendar title.
<code>dow_ascent</code>	specifies the ascent in pixels of the days of the week title.
<code>dow_lht</code>	specifies the height in pixels of the days of the week title.
<code>mth_ascent</code>	specifies the ascent in pixels of the month title.
<code>mth_lht</code>	specifies the height in pixels of the month title.
<code>day_ascent</code>	specifies the ascent in pixels of a day of the month number.
<code>day_lht</code>	specifies the height in pixels of a day of the month number.
<code>daygapx</code>	specifies the gap between adjacent days of the month.
<code>daywidth</code>	specifies the width of a day of the month number i.e. two times the width of a numeric character.

The `CI_FONT_INFO` structure which is used to specify the appearance of all text displayed in the calendar is defined as follows:

```
typedef struct
{
    WORD fid;
    UWORD style;
    UWORD leading;
} CI_FONT_DATA;
```

The significance of the members of the `CI_FONT_INFO` structure is as follows:

`fid` specifies the font ID of the text.

`style` specifies the style of the text.

`leading` specifies the leading - i.e. vertical spacing - below the text.

If the total number of months in the calendar as specified by `init->mcol*init->mrow` exceeds `CI_MAX_MONTH` the method calls `p_leave` with an argument of `E_GEN_TOOMANY`.

The method initialises the property according to the content of the `CI_INIT` structure with address `init` as described above.

If `init->hscrlm` is greater than the total number of months in the calendar as specified by `init->mcol*init->mrow` then the method resets `init->hscrlm` to the total number of months in the calendar.

If `init->startm` is outside the allowed range of 0 to 11 inclusive, the method resets `init->startm` to 0.

If the current month as specified by `init->days` is not included in the calendar view, the method adjusts `init->startm` appropriately.

Write `*init` to `calimg.img`.

CI_SET_TITLE

Set the title

```
VOID ci_set_title(TEXT *zts)
```

Replace the calendar title with the zero terminated string pointed to by `zts`.

The method frees the cell pointed to by `calimg.title`.

If `zts` is `NULL` the method writes zero to `calimg.title`.

Otherwise the method allocates an appropriately sized cell and copies into the cell the title string pointed to by `zts`. The address of the cell is written to `calimg.title`.

CI_EMPHASISE

Emphasise the view

```
VOID ci_emphasise(UINT flag)
```

Emphasise the calendar if `flag` is `TRUE`, otherwise de-emphasise the calendar.

If `flag` is equal to `calimg.emphasised` the method simply returns as no change is required.

Otherwise the method records the emphasis state by writing `flag` to `calimg.emphasised`.

If `calimg.img.flags` does not contain `CALIMG_CURSOR_NO_FLASH`, and `flag` is `TRUE` the method draws the cursor by calling `wTextCursor`.

If `calimg.img.flags` does not contain `CALIMG_CURSOR_NO_FLASH`, and `flag` is `FALSE` the method erases the cursor by calling `wEraseTextCursor`.

CI_MOVE_CURSOR**Move the cursor**

```
VOID ci_move_cursor(INT flag)
```

Move the cursor to the position specified by `flag` and redraw the calendar view ensuring that the current date is visible.

The allowed values of `flag` are as follows:

CALIMG_GOTO_TODAY	move the cursor to the date today - as stored by the machine - by sending <code>self</code> a <code>CI_GOTO_DATE</code> message with appropriate arguments.
CALIMG_LEFT	move the cursor to the left one day by sending <code>self</code> a <code>CI_POS_MXY</code> message specifying <code>calimg.img.hscrlm</code> as the scroll increment. If the resulting cursor position is not valid, move the cursor to the last day of the month.
CALIMG_RIGHT	move the cursor to the right one day by sending <code>self</code> a <code>CI_POS_MXY</code> message specifying <code>calimg.img.hscrlm</code> as the scroll increment. If the cursor is either on the last day of the month or in the last column of the month the method moves the cursor rightwards into the first column of the next month. If this is not a valid day, the method then moves the cursor upwards until a valid day is found.
CALIMG_UP	move the cursor up one day by directly calling the <code>ci_pos_mxy</code> member function specifying <code>mcol</code> as the scroll increment.
CALIMG_DOWN	move the cursor down one day by directly calling the <code>ci_pos_mxy</code> member function specifying <code>mcol</code> as the scroll increment.
CALIMG_HOME	move the cursor horizontally to the left most day in the calendar view by directly calling the <code>ci_pos_mxy</code> member function.
CALIMG_END	move the cursor horizontally to the right most day in the calendar view by directly calling the <code>ci_pos_mxy</code> member function.
CALIMG_PAGEUP	move the calendar view and the cursor backwards in time by <code>calimg.img.mrow*calimg.img.mcol</code> months by directly calling the <code>ci_pos_mxy</code> member function.
CALIMG_PAGEDN	move the calendar view and the cursor forwards in time by <code>calimg.img.mrow*calimg.img.mcol</code> months by directly calling the <code>ci_pos_mxy</code> member function.
CALIMG_PREV_DAY	move the cursor backwards in time one day by sending <code>self</code> a <code>CI_ADJUST_DATE</code> message specifying a scroll increment of <code>calimg.img.hscrlm</code> .
CALIMG_NEXT_DAY	move the cursor forwards in time one day by sending <code>self</code> a <code>CI_ADJUST_DATE</code> message specifying a scroll increment of <code>calimg.img.hscrlm</code> .
CALIMG_PREV_MONTH	move the cursor backwards in time by one month by sending <code>self</code> a <code>CI_ADJUST_DATE</code> message specifying a scroll increment of <code>calimg.img.hscrlm</code> .
CALIMG_PREV_YEAR	move the cursor backwards in time by one year by sending <code>self</code> a <code>CI_ADJUST_DATE</code> message specifying <code>calimg.img.mrow*calimg.img.mcol</code> as the scroll increment.
CALIMG_NEXT_MONTH	move the cursor forwards in time by one month by sending <code>self</code> a <code>CI_ADJUST_DATE</code> message specifying a scroll increment of <code>calimg.img.hscrlm</code> .
CALIMG_NEXT_YEAR	move the cursor forwards in time by one year by sending <code>self</code> a <code>CI_ADJUST_DATE</code> message specifying <code>calimg.img.mrow*calimg.img.mcol</code> as the scroll increment.

CALIMG_PREV_WEEK	move the cursor backwards in time one week by sending self a CI_ADJUST_DATE message specifying a scroll increment of calimg.img.hscrln.
CALIMG_NEXT_WEEK	move the cursor forwards in time one week by sending self a CI_ADJUST_DATE message specifying a scroll increment of calimg.img.hscrln.

Note that both the ci_adjust_date and the ci_pos_mxy methods redraw the calendar view once the cursor has been repositioned.

CI_GOTO_DATE Move the cursor by date

```
VOID ci_goto_date(CI_DATE *pdate)
```

Move the cursor to the date specified by the CI_DATE structure pointed to by pdate and redraw the calendar view ensuring that the current date is visible.

The CI_DATE structure is defined as follows:

```
typedef struct
{
    WORD year;
    WORD month;
    WORD day;
}CI_DATE;
```

The significance of the members of the CI_DATE structure is as follows:

year	The candidate year number in the range 0 to 254 inclusive where 0 is the year 1900 - set to zero if a negative value is specified.
month	The candidate month number in the range 0 to 11 inclusive where 0 is January - set to the nearest limit if the specified value is outside of the allowed range.
day	The candidate day number in the range 0 to 30 inclusive where 0 is the first of the month - set to the nearest valid day in the month if the specified value is not a valid day. Note of course that the last valid day number may be less than 30.

The method updates calimg.crs according to the values specified in pdate and then updates the view by sending self a CI_VIEW message specifying a scroll increment of calimg.img.hscrln.

CI_ADJUST_DATE Adjust the current date

```
VOID ci_adjust_date(CI_DATE *pinc, INT flag, INT hscrln)
```

Move the cursor forwards or backwards in time as specified by pinc and flag and redraw the calendar view ensuring that the current date is visible specifying a scroll increment of hscrln.

The action is controlled by writing one of the following values to flag:

CALIMG_ADJ_YEAR	adjust the year, the month and the day.
CALIMG_ADJ_MONTH	adjust the month and the day.
CALIMG_ADJ_DAY	adjust the day.

The year is adjusted by moving the cursor forwards or backwards in time by pinc->year years.

The month is adjusted by moving the cursor forwards or backwards in time by pinc->month months and if the cursor is not on a valid day moving the cursor to the last day of the month.

The day is adjusted by moving the cursor forwards or backwards in time by pinc->day days.

Note that if the cursor moves to a year that is out of range the method beeps and then returns without modifying the property. The method call thus has no effect.

The method draws the view by sending self a CI_VIEW message specifying a scroll increment of hscrln.

CI_REDRAW**Redraw part of the view**

```
VOID ci_redraw(P_RECT *prect)
```

Redraw calendar months that overlap the rectangle defined by `prect`.

If `calimg.img.flags` contains `CALIMG_CURSOR_NO_FLASH`, and the current cursor position is visible, draw an appropriately sized inverted obloid at the current cursor position. (Otherwise the system takes care of drawing the flashing cursor.)

CI_SENSE**Sense the current date**

```
VOID ci_sense(ULONG *psense)
```

Write the current date to the `ULONG` pointed to by `psense`. The current date is expressed as the number of days elapsed since 1/1/1900.

CI_VIEW**Draw the view**

```
VOID ci_view(INT hscrlm)
```

Draw the calendar view ensuring that the current date is visible specifying a scroll increment of `hscrlm`.

The method sets the first month in the calendar view as specified by `calimg.img.startm` and `calimg.ystart` and then scrolls the calendar view forwards or backwards in time an integral multiple of `hscrlm` months until the current date is included in the calendar view.

On occasion this will lead to months outside the allowed date range being included. In such cases the method sets the first month in the calendar view as specified by `calimg.img.startm` and `calimg.crs.year` and then repeats the above algorithm.

The method then draws the calendar view, using the `wScrollRect` routine whenever possible.

CI_POS_MXY**Move the cursor by position**

```
VOID ci_pos_mxy(CI_CURSOR *pcrs,INT gravity,INT scl)
```

Move the cursor to the year, month and coordinates specified by the `CI_CURSOR` structure with address `pcrs` specifying a scroll increment of `scl`.

Note that if the coordinates within the month do not correspond to a valid day the method moves the cursor according to the value of `gravity` until a valid day is located.

The `CI_CURSOR` structure is defined as follows:

```
typedef struct
{
    UBYTE year;
    UBYTE month;
    P_POINT pos;
} CI_CURSOR;
```

The significance of the members of the `CI_CURSOR` structure is as follows:

`year` the year number in the range 0 to 254 inclusive where 0 is the year 1900.

`month` the month number in the range 0 to 11 inclusive where 0 is January.

`pos` an x,y position within a month: the third day on the second row for example has position (2,1).

If `pcrs->pos.x` is less than zero, the method resets `pcrs->pos.x` to 6, and moves the cursor backwards in time one month.

If `pcrs->pos.x` is greater than 6, the method resets `pcrs->pos.x` to 0, and moves the cursor forwards in time one month.

If `pcrs->pos.y` is less than zero, the method resets `pcrs->pos.y` to 6, and moves the cursor backwards in time `calimg.img.mcol` months.

If `pcrs->pos.y` is greater than 5, the method resets `pcrs->pos.y` to 0, and moves the cursor forwards in time `calimg.img.mcol` months.

If as a result of one of the above tests the year exceeds 2154, the method beeps and then returns.

If the coordinates specified by `pcrs->pos` do not correspond to a valid day in the current month, the method moves the cursor in the manner indicated by `gravity` until a valid day is located. The allowed values for `gravity` are as follows:

- | | |
|---------------------------|--|
| <code>CALING_NORTH</code> | move the cursor upwards in the calendar until a valid day is located. |
| <code>CALING_WEST</code> | <p>if the cursor is in the bottom row of the current month, the bottom row contains no valid days and the cursor lies to the right of the last day of the month, move the cursor to the last day of the current month.</p> <p>otherwise if the cursor is in the bottom row of the current month and the bottom row contains no valid days move the cursor to the first day in the last valid row of the current month i.e. to coordinates (0,4), or (0,3) if (0,4) is not valid.</p> <p>otherwise move the cursor leftwards in the calendar view until a valid day is located.</p> |
| <code>CALING_SOUTH</code> | move the cursor downwards in the calendar view until a valid day is located. |
| <code>CALING_EAST</code> | <p>if the current position is in a bottom row of a month which contains no valid days, move the cursor to the last valid day of the month.</p> <p>otherwise move the cursor rightwards in the calendar view until a valid day is located.</p> |

The method draws the view by sending `self` a `CI_VIEW` message specifying a scroll increment of `scr1`.

CI_TODAY_CHANGED

Update today's date

`VOID ci_today_changed(VOID)`

Update today's date and redraw the calendar as required.

The method obtains the actual date by calling the `p_date` PLIB routine and then writes the year number to `calimg.thisyear`, writes the month number to `calimg.thismth` and writes the day number to `calimg.thisday`.

The method redraws the calendar view as required to ensure that today's date is correctly highlighted.

CHAPTER 7

THE POLYTEXT CLASSES

The Polytext classes are a set of classes for displaying text in a variety of window server fonts and styles. In addition, the classes also implement their own styles; for example, text can be displayed with an overstrike, a horizontal line through the text to implement "crossing out".

Text may be wrapped into a number of lines where the line boundaries are defined by the application.

FORM supplies three classes. The PTROOT class is an abstract class which *must* be subclassed to provide a usable Polytext class. PTROOT contains a number of deferred methods which must be supplied by a subclass.

PTSEG and PTFLAT subclass PTROOT and supply the required deferred methods. PTROOT itself can be seen as supplying the *basic* or *common* methods and properties needed to implement a fully functioning Polytext class.

A number of terms and concepts are used in the description of these classes and it will be useful to give them here.

A *phrase* describes a segment of text. It is a combination of the text itself and information which qualifies it, such as the length of text, the window server font to be applied; it is represented by a data structure of type PT_PHRASE. Note that a phrase can contain a maximum of PT_MAX_PHRASE_LEN characters. This and other symbols and structures can be found in the Polytext class definition in *polytext.cl*.

Phrases are collected into a *buffer* in the order in which they would be displayed. PTROOT makes no assumptions about the way a buffer is implemented; this decision is left to a subclass. PTSEG implements a buffer as an instance of a VAXVAR array while PTFLAT simply allocates a single cell and adds phrases in sequence into this cell.

A *line-table* is built when text is wrapped into a number of lines with each line having a definite length. The table is a list of byte values containing the number of text characters within each line. The number of bytes in the table is, therefore, the same as the number of lines.

The line-table, itself, is normally placed at the beginning of the buffer. The mechanism by which the line-table is inserted depends on the way the buffer is implemented. In PTSEG, the first entry in the VAXVAR array is reserved for the table while in PTFLAT, the table together with a preceding byte containing the number of bytes in the table, is inserted directly at the start of the buffer causing any existing records to be shifted and the buffer to be re-allocated, if necessary.

The Polytext classes are used as part of implementation of the Series 3a Agenda built-in application.

Note that the classes themselves are only defined and implemented in the version of FORM as exists on the Series 3a.

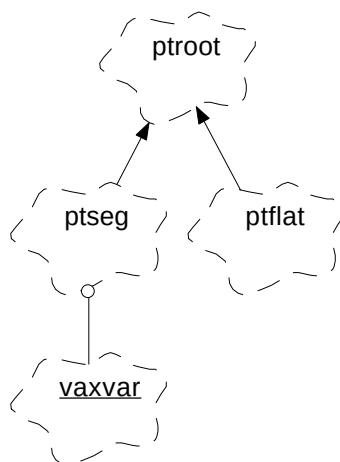
Precursors

An understanding of the Polytext classes will be helped by a knowledge of:

- the p_enter and p_leave error handling services
- the OLIB variable array class VAXVAR
- the Window Server functions: gSetGC, gPrintBoxText, gClrRect and gFontInfo

Class diagram

The following diagram covers the relationships between the Polytext classes which are discussed in detail in this chapter. The underlined classes are either discussed in another chapter of this manual or they refer to OLIB classes in which case they are *all* described in the *OLIB Reference* manual.



PTROOT

PTRoot	
nphrases	wwidth
nchars	lmargin
nlines	ascent
pt_add_phrase	pt_init
pt_wrap	pt_reset
pt_display_line	pt_append
pt_set_fonts	pt_put_lintab
pt_mod_by_num	pt_pbuf
pt_mod_by_attr	
pt_find	
pt_inquire	

Class definition

The PTRoot class subclasses ROOT and is defined in the sub-category file *polytext.cl* (with generated header file *polytext.g*).

```

CLASS polytext root
{
  DEFER pt_init
  DEFER pt_reset      Reset entire polytext as just initialized
  DEFER pt_append     for "internal" use - by ptroot only
  DEFER pt_put_lintab for "internal" use - by ptroot only
  DEFER pt_pbuf       for "internal" use - by ptroot only
  ADD pt_add_phrase   Add a phrase to end of polytext
  ADD pt_wrap         re-wrap text based on changed conditions
  ADD pt_display_line Draw single line of polytext
  ADD pt_set_fonts    set/reset all fonts per PT_FONT_SPEC array
  ADD pt_mod_by_num   i.e. modify phrase descriptor by phrasenum
  ADD pt_mod_by_attr  i.e. modify descriptor if attribute matches
  ADD pt_find         Return phrase number of next matching phrase
  ADD pt_inquire      Return screen position of phrase
}
  
```

CONSTANTS

{		
PT_FIND_DEFAULT	0x0000	Exec O_PT_FIND method in default mode
PT_FIND_BACKWARDS	0x0001	Execute O_PT_FIND method "descending"
PT_FIND_CAN_STAY	0x0002	Exec O_PT_FIND including start phrase
PT_STY_DEFAULT	0x00	Default polytext phrase style
PT_STY_BREAK_AT_START	0x01	Start of phrase is acceptable wrap pt
PT_STY_BREAK_AT_END	0x02	End of phrase is acceptable wrap point
PT_STY_OSTRIKE_XLEFT	0x20	Extend overstrike to left
PT_STY_OSTRIKE_XRIGHT	0x40	Extend overstrike to right
PT_STY_OVERSTRIKE	0x80	Overstrike displayed text
PT_MAX_PHRASE_LEN	236	Max text bytes in one phrase
PT_DESCR_MOD_FONT	0x0001	Modify font in phrase descriptor
PT_DESCR_MOD_WS_STYLE	0x0002	Modify style in phrase descriptor
PT_DESCR_MOD_PT_STYLE	0x0004	Modify style in phrase descriptor
PT_DESCR_MOD_ATTRIB	0x0008	Modify attrib (by phrase number only)
PT_DESCR_MOD_TLEN	0x0010	Modify text length (not implemented)
}		

TYPES

```

{
    typedef struct
    {
        UWORD font_id;    font ID for corresponding ptxt phrase
        UBYTE attrib;     polytext phrase attribute
    } PT_FONT_SPEC;      polytext font specifier.

    typedef struct
    {
        INT line;         Line number in which phrase starts
        INT offset;       Pixel offset to start of phrase
        INT width;        Pixel width of the phrase
    } PT_PHRASE_INFO;    Information returned by O_PT_INQUIRE method

    typedef struct
    {
        UWORD font_id;    Font for this segment
        UBYTE ws_style;    Window server style
        UBYTE pt_style;    Polytext display style
        UBYTE attrib;      To be specified by caller
        UBYTE tlen;        Length of text in segment
    } PT_PHRASE_DESCR;    Phrase descriptor

    typedef struct
    {
        PT_PHRASE_DESCR descr;
        TEXT txt[1];       Start of formatted text string.
    } PT_PHRASE;           Phrase (descriptor plus text)
}

```

PROPERTY

```

{
    UINT nphrases;        Number of phrases in polytext
    UINT nchars;          Total number of chars in entire polytext
    UINT nlines;          Number of lines (bytes) in line-length table
    UINT wwidth;          Width used for last wrap
    UINT lmargin;         Left margin for text display
    UINT ascent;          Ascent for text display
}

```

}

Property

<code>ptroot.nphrases</code>	The <i>total</i> number of phrases represented by this instance.
<code>ptroot.nchars</code>	The <i>total</i> number of characters represented by this instance.
<code>ptroot.nlines</code>	This property is of interest when the text represented by this instance has been wrapped; it is the <i>number</i> of lines into which the text has been wrapped.
<code>ptroot.wwidth</code>	The maximum width of a line, in pixels, available for displaying text. This property is important when text is being wrapped. This value <i>excludes</i> the width of the left-hand margin, if any.
<code>ptroot.lmargin</code>	The width of the left-hand margin, in pixels. Text is wrapped so that it fits between the left-hand margin and the right-hand margin. The right-hand margin is <code>ptroot.wwidth</code> pixels from the left-hand margin.
<code>ptroot.ascent</code>	Ascent for text display. This is the distance between the base line of the text and the top of the rectangle or "box" within which the segment of text is drawn and is specified by the application. For more information on this concept, see the description of the <code>gPrintBoxText</code> function in the <i>Graphics Output</i> chapter of the <i>Window Server Reference</i> .

PTROOT methods

PT_ADD_PHRASE**Add phrase to buffer**

```
INT pt_add_phrase(PT_PHRASE_DESCR *pd, TEXT *txt);
```

Add a phrase to the buffer.

The method takes two parameters:

- `pd` points to a data structure of type `PT_PHRASE_DESCR` which contains information describing this phrase, for example, the length of the text and the ID of the font to be applied.
- `txt` holds the address of a buffer containing the text of the phrase to be added.

The method takes the text and the phrase description supplied in the parameters and builds a `RC_VAXVAR` type data structure representing the data to be added to the buffer. The `RC_VAXVAR` structure is defined in the `OLIB` class `VAXVAR` but is shown below:

```
typedef struct
{
    UWORD len;
    UBYTE *buf;
} RC_VAXVAR
```

The method constructs a `PT_PHRASE` record, fully describing the phrase, and sets the address of this record into the member `buf`.

The member `len` contains the length of the data represented by this `PT_PHRASE` record.

The length of text in a single phrase is limited to `PT_MAX_PHRASE_LEN` characters. Thus, if more than `PT_MAX_PHRASE_LEN` characters are passed to this method, then a number of `PT_PHRASE` records will be created. In practice, no more than two records can ever be created.

New `PT_PHRASE` records are added to the buffer by sending one `PT_PUT_APPEND` message per record. The `pt_put_append` method is a deferred method and must be supplied by a subclass. The implementation of this method depends on the way the buffer itself is implemented, as discussed in the introduction to this chapter. The `PTFLAT` and `PTSEG` sub-classes supply a suitable method.

As new phrases are added to the buffer, the method updates the properties `ptroot.nchars` and `ptroot.nphrases`, the total number of characters and the total number of phrases respectively.

The method always returns zero.

PT_WRAP**Wrap the text**

```
INT pt_wrap(INT maxwidth,INT margin, UWORD wrapflag);
```

Wrap the text represented by this instance and return the number of lines generated.

The method takes three parameters:

- `maxwidth` is a value which gives the maximum length of each line in pixels. This value includes the length of the left hand margin (if any).
- `margin` is a value which gives the width of the left hand margin in pixels.
- `wrapflag` is a value which can take the value `TRUE` or `FALSE`.

The method wraps the text represented by this instance by reading through all phrases held in the buffer (by sending a series of `PT_PBUF` messages) and fitting the text into lines whose pixel width is given by the value of `maxwidth - margin`. The result of the wrapping process is a line-table as described in the introduction to this chapter. The method returns the number of lines generated.

Note that `pt_pbuf` is a deferred method and must be supplied by a subclass. The implementation of this method depends on the way the buffer itself is implemented as discussed in the introduction to this chapter. The `PTFLAT` and `PTSEG` sub-classes supply a suitable method.

If `wrapflag` is `TRUE`, the method will wrap the text into as many lines as necessary.

If, however, `wrapflag` is `FALSE`, an attempt is made to fit the text into a single line; if necessary the text is clipped to fit into the available width. In this case, the method *always* returns a value of one.

As new lines are added to the line-table, the method updates the property `ptroot.nlines`, the number of lines into which the text has been wrapped.

Once the line-table is complete, a `PT_PUT_LINTAB` message is sent to add the line-table to the buffer. `pt_put_lintab` is a deferred method and must be supplied by a subclass. The implementation of this method depends on the way the buffer itself is implemented, as discussed in the introduction to this chapter. The `PTFLAT` and `PTSEG` sub-classes supply a suitable method.

PT_DISPLAY_LINE**Draw a line of text**

```
VOID pt_display_line(P_RECT *prect,INT ascent,UINT displine);
```

Draw a single line of text to the screen

The method takes three parameters:

- `prect` points to data structure of type `P_RECT` and describes a rectangle within which the line of text is to be drawn.
- `ascent` is as used by the window server function `gPrintBoxText`. It measures the required distance between the base line of the text and the top of the rectangle within which the text is to be drawn.
- `displine` is the number of the line to be drawn and is used as an index into the line-table. This assumes that the text has previously been wrapped.

If no phrases exist or the number of the line to be displayed is invalid, the pixels within the specified rectangle are cleared and the method returns.

The phrases corresponding to the line to be displayed are fetched in turn from the buffer using the `pt_pbuf` deferred method. For each phrase fetched, the window server function `gSetGC` is called to switch the graphics context font and style to that specified by the phrase. The method assumes that a temporary graphics context has already been created by the application. The window server function `gPrintBoxText` is used to display the text within each phrase.

If a phrase has the Polytext style `PT_STY_OVERSTRIKE` set, a horizontal "line", two pixels deep, is drawn through the text. The "line" itself is drawn by clearing the top line of pixels and by setting the bottom line of pixels.

PT_SET_FONTS**Set font ID**

```
VOID pt_set_fonts(UINT count, PT_FONT_SPEC *pfspec0);
```

Set the font ID for phrases in the buffer.

The method takes two parameters:

- `pfspec0` is a pointer to an *array* of `PT_FONT_SPEC` data structures.
- `count` contains the number of entries in the array of `PT_FONT_SPEC` data structures whose address is passed in the parameter `pfspec0`.

As can be seen in the `PTROOT` class definition, each element in the array of `PT_FONT_SPEC` data structures consists of a member (`attrib`) containing a set of Polytext attributes and a member (`font_id`) containing a font ID.

The method scans through all the phrases in the buffer; for each phrase, all elements in the `PT_FONT_SPEC` array are examined. Where the Polytext attributes of the phrase match an element's attributes, the font ID in the phrase is *replaced* by that in the `PT_FONT_SPEC` element and scanning then continues with the next phrase in the buffer.

Note that the attributes of the phrase will match the `PT_FONT_SPEC` element's attributes, if a logical *AND* of the two sets results in a `TRUE` value.

As a result of this method, some or all of the phrases in the buffer will have new font IDs. It is also possible that none of the phrases will be changed.

PT_MOD_BY_NUM**Set font ID and style by phrase**

```
VOID pt_mod_by_num(UINT phrasenum, PT_PHRASE_DESCR *pdescr, UWORD flags);
```

Set the font ID and graphic styles for a specific phrase.

The method takes three parameters:

- `phrasenum` is an index which identifies the exact phrase within the buffer. A value of one refers to the first phrase while a value of two refers to the second and so on.
- `pdescr` points to a data structure of type `PT_PHRASE_DESCR` and contains the font-id, window server style, polytext style and attribute to be set into the phrase.
- `flags` contains a set of bit values which can be *ORED* together; it indicates which item(s) in the phrase descriptor is(are) to be set. The possible values are as follows:

`PT_DESCR_MOD_FONT`

`PT_DESCR_MOD_WS_STYLE`

`PT_DESCR_MOD_PT_STYLE`

`PT_DESCR_MOD_ATTRIB`

If `phrasenum` contains an invalid value (i.e. zero or a value greater than the total number of phrases in the buffer), the method does nothing and simply returns.

The address of the specific phrase within the buffer is found by sending a `PT_PBUF` message and passing `phrasenum` as an argument. Recall that `pt_pbuf` is a deferred method and must be supplied by a subclass. The implementation of this method depends on the way the buffer itself is implemented as discussed in the introduction to this chapter. The `PTFLAT` and `PTSEG` sub-classes supply a suitable method.

Depending on the setting of the `flags` parameter, corresponding members referenced by the `pdescr` parameter replace the equivalent members in the phrase descriptor as follows:

<code>PT_DESCR_MOD_FONT</code>	causes the phrase's <code>font_id</code> member to be replaced by <code>pdescr->font_id</code> .
<code>PT_DESCR_MOD_WS_STYLE</code>	causes the phrase's <code>ws_style</code> member to be replaced by <code>pdescr->ws_style</code> .
<code>PT_DESCR_MOD_PT_STYLE</code>	causes the phrase's <code>pt_style</code> member to be replaced by <code>pdescr->pt_style</code> .
<code>PT_DESCR_MOD_ATTRIB</code>	causes the the phrase's <code>attrib</code> member to be replaced by <code>pdescr->attrib</code> .

PT_MOD_BY_ATTRIB **Set font ID and style by attribute**

```
VOID pt_mod_by_attrib( PT_PHRASE_DESCR *pdescr, UWORD flags );
```

Set the font ID and graphic styles for phrases within the buffer.

The method takes two parameters:

- `pdescr` points to a data structure of type `PT_PHRASE_DESCR` and contains the font-id, window server style and polytext style to be set into the phrase(s). It also contains the attributes to be used to find matching phrases.
- `flags` contains a set of bit values which can be ored together; it indicates which item(s) in the phrase descriptor is(are) to be set. The possible values are as follows:

```
PT_DESCR_MOD_FONT
PT_DESCR_MOD_WS_STYLE
PT_DESCR_MOD_PT_STYLE
```

The method scans through all the phrases in the buffer by sending successive `PT_PBUF` messages. Where the Polytext attributes of the phrase match the attributes referenced by the `pdescr` parameter, corresponding *members* referenced by the `pdescr` parameter replace the equivalent members in the phrase descriptor, depending on the setting of the `flags` parameter as follows:

<code>PT_DESCR_MOD_FONT</code>	causes the phrase's <code>font_id</code> member to be replaced by <code>pdescr->font_id</code> .
<code>PT_DESCR_MOD_WS_STYLE</code>	causes the phrase's <code>ws_style</code> member to be replaced by <code>pdescr->ws_style</code> .
<code>PT_DESCR_MOD_PT_STYLE</code>	causes the phrase's <code>pt_style</code> member to be replaced by <code>pdescr->pt_style</code> .

Note that the attributes of the phrase will match the attributes referenced by `pdescr`, if a logical *AND* of the two sets results in a *TRUE* value.

PT_FIND **Find phrase by attribute**

```
INT pt_find( UINT phrase0, UWORD flags, UWORD attrib );
```

Find the next phrase whose attributes match a given set and return its index.

The method takes three parameters:

- `phrase0` is the index of the phrase within the buffer where the search is to begin.
- `flags` contains indicators which determine:
 1. whether or not the search process is to include the phrase represented by `phrase0`.
 2. the direction of search, i.e. forwards or backwards.
- `attrib` contains the attributes to be matched with the phrase attributes.

The method scans each phrase in the buffer, beginning with the phrase whose index (or relative position within the buffer) is given by `phrase0`, until the Polytext attributes of the phrase match the attributes given by the `attrib` parameter.

The index of the resulting phrase is returned. If no matching phrase can be found, a value of -1 is returned instead.

Note that if `PT_FIND_BACKWARDS` is set in the `flags` parameter, the search is done *backwards* from the phrase indicated by `phrase0`. Further, if `PT_FIND_CAN_STAY` is set, the search *includes* the phrase indicated by `phrase0`; otherwise, it is excluded.

PT_INQUIRE Find information about a phrase

```
INT pt_inquire(INT phrasenum, PT_PHRASE_INFO *pinfo);
```

Fetch information about a given phrase.

The method takes two parameters:

- `phrasenum` is the index of the phrase, i.e. the relative position of the phrase within the buffer.
- `pinfo` points to a `PT_PHRASE_INFO` data structure to be filled in by the method.

If no phrases exist or the parameter `phrasenum` contains an invalid value (i.e. zero or a value greater than the total number of phrases in the buffer), then a value of -1 is returned.

The method sends `PT_PBUF` messages to fetch the line-table and the given phrase and, using this information, fills in the `PT_PHRASE_INFO` data structure.

The `PT_PHRASE_INFO` data structure has three members described as follows:

- `line` The number of the line within which the given phrase will be displayed.
- `offset` The offset, in pixels, of the start of the given phrase from the beginning of the line.
- `width` The width, in pixels, of the given phrase.

On successful completion, the method returns zero.

PTROOT deferred methods

PT_INIT Initialise

```
INT pt_init(UINT granularity)
```

A deferred method for initialising the instance.

The method should handle a single parameter specifying the granularity of the buffer. The way this value is used will depend on the way the buffer is implemented. In very general terms, the size of a buffer should always be some multiple of the granularity.

The method is not used in this class

PT_RESET Reset

```
VOID pt_reset(VOID)
```

A deferred method for resetting the instance back to its initialised state.

The method is not used in this class.

PT_APPEND**Append a record**

VOID pt_append(RC_VAXVAR *pdescr)

A deferred method for adding phrases to the buffer

The method should handle a single parameter. This is a pointer to a data structure of type RC_VAXVAR containing the address of the phrase to be added to the buffer and the total length of this phrase (a PT_PHRASE data structure).

The way this parameter is used will depend on the way the buffer is implemented.

The method is used in this class by the pt_add_phrase method.

PT_PUT_LINTAB**Store line-length table**

VOID pt_put_lintab(UBYTE *plintable)

A deferred method for inserting the line-table into the buffer.

The method should handle a single parameter which is a pointer to the line-table itself.

The PTROOT methods assume that the line-table is always located at the *beginning* of the buffer and regards it as being phrase zero; in other words, the address of the line-table can be found by sending a PT_PBUF message with an argument of zero.

The method is used by the pt_wrap method.

PT_PBUF**Get address of phrase**

PT_PHRASE *pt_pbuf(UINT recnum)

A deferred method for obtaining the address of a phrase in the buffer.

The method should handle a single parameter. This is the index of the phrase; in other words, it is the relative position of the phrase within the buffer. A value of one refers to the first phrase while a value of two refers to the second phrase and so on. Note, however, that a value of zero refers to the line-table.

The method is used by the following methods: pt_wrap, pt_display_line, pt_set_fonts, pt_mod_by_num, pt_mod_by_attr, pt_find, and pt_inquire.

PTFLAT

PTROOT	PTFLAT
nphrases nchars nlines wwidth lmargin ascent	buffer bufsize granularity nextoff
pt_add_phrase pt_wrap pt_display_line pt_set_fonts pt_mod_by_num pt_mod_by_attr pt_find pt_inquire ~~pt_init~~ ~~pt_reset~~ ~~pt_append~~ ~~pt_put_lintab~~ ~~pt_pbuf~~	destroy pt_init pt_reset pt_append pt_put_lintab pt_pbuf

PTFLAT is a subclass of PTR00T. It implements the buffer as a single memory cell. Phrases are simply appended to the end of the cell as they are received. The line-table is inserted at the beginning of the cell.

If the cell proves too small to contain extra phrases, it is simply re-allocated. Extra property is provided by this subclass to control access to this cell and is detailed in the property section below.

All methods deferred by PTR00T are supplied here.

Class definition

The PTFLAT class subclasses PTR00T and is defined in the sub-category file *polytext.cl* (with generated header file *polytext.g*).

```
CLASS  ptflat  ptroot
{
  REPLACE  destroy
  REPLACE  pt_init
  REPLACE  pt_reset
  REPLACE  pt_append          for "internal" use - by ptroot only
  REPLACE  pt_put_lintab      for "internal" use - by ptroot only
  REPLACE  pt_pbuf            for "internal" use - by ptroot only

  PROPERTY
  {
    UBYTE  *buffer;
    UWORD  bufsize;
    UWORD  granularity;
    UWORD  nextoff;
  }
}
```

Property

ptflat.buffer	The address of the allocated cell containing the buffer.
ptflat.bufsize	The size, in bytes, of the allocated cell containing the buffer.
ptflat.granularity	The granularity of the buffer in bytes. The cell in which the buffer resides is always allocated in exact multiples of the granularity. If necessary, the size of the buffer is <i>always</i> rounded up to the next multiple of the granularity.
ptflat.nextoff	The offset, from the beginning of the cell, to the next available position within the buffer.

PTFLAT methods

DESTROY

Destroy the instance

VOID destroy(VOID)

Destroy this instance of PTFLAT.

The method frees the allocated cell which contains the buffer and then supersends a DESTROY message.

PT_INIT

Initialise

INT pt_init(UINT granularity)

Initialise this instance of PTFLAT.

The method takes a single parameter; *granularity* specifies the granularity of the buffer. The buffer is contained within a cell which is always allocated in multiples of this value.

The method takes the value of this parameter and sets it into the property *ptflat.granularity*.

A minimum buffer is constructed by allocating a cell of size `ptflat.granularity`; its address is set into the property `ptflat.buffer` and its current size set into the property `ptflat.bufsize`. The buffer will be expanded (i.e. re-allocated), as necessary by other PTFLAT methods.

In this subclass, the first byte of the buffer is used to contain the number of bytes allocated to the line-table; the line-table itself, will always *follow* this single byte and will *precede* the sequence of phrases. As part of the initialisation process, the first byte is set to zero, indicating that there is no line-table. The property `ptflat.nextoff`, containing the offset of the next free byte in the buffer, is initialised to one.

`p_leave` is called if there is insufficient memory to allocate the minimal buffer, otherwise the method returns zero.

N.B. the property `ptroot.nlines` will always contain the *current* number of lines into which the text is wrapped and will, therefore, give the *current* number of entries used in the line-table. This will not necessarily be the same as the value in the first byte of the buffer.

In general, the value of `ptroot.nlines` will *always* be less than or equal to the value in the first byte of the buffer.

PT_RESET

Reset

```
VOID pt_reset(VOID)
```

Reset the instance back to its initialised state.

The method effectively resets a number of `ptroot` properties, re-allocates the buffer to its minimal size and initialises it in the same way as described in the `pt_init` method. In effect, the method "empties" the PTFLAT object of all text and deletes any existing line-table.

The properties which are reset to zero are: `ptroot.nphrases`, `ptroot.nchars`, `ptroot.nlines` and `ptroot.width`.

PT_APPEND

Append a phrase

```
VOID pt_append(RC_VAXVAR *pdescr)
```

Append a phrase to the buffer.

The method takes a single parameter; `pdescr` points to a data structure of type `RC_VAXVAR`. The members of this structure, `len` and `buf`, contain the length of the phrase and a pointer to the phrase content (a `PT_PHRASE` data structure) respectively. The `RC_VAXVAR` structure can be found in the OLIB VAXVAR class definition.

The phrase data is appended to the end of the buffer. If necessary, the buffer is re-allocated to accommodate the new data and the `ptflat.nextoff` property is updated to point to the next free byte.

PT_PUT_LINTAB

Store line-length table

```
VOID pt_put_lintab(UBYTE *plintable)
```

Store the line-table in the buffer.

The method takes a single parameter; `plintable` contains the address of the line-table to be stored.

The method inserts the line-table into the buffer so that it starts at the second byte; any existing line-table will be overwritten. If necessary, the buffer is re-allocated and any existing phrases moved to make room for the new table. If the space reserved for the line-table is increased, the first byte in the buffer will be updated to reflect the new size.

PT_PBUF**Get address of phrase**

PT_PHRASE *pt_pbuf(UINT recnum)

Fetch the address of the specified phrase within the buffer.

The method takes a single parameter; `recnum` contains the index of the phrase whose address is required. In other words, it is the relative position within the buffer of the required phrase.

If `recnum` is zero, the address of the line-table is returned; otherwise, the method returns the address of the required phrase. A `recnum` value of one causes the address of the first phrase to be returned, a value of two causes the address of the second phrase to be returned, and so on.

Note, this method does *not* check that the value of `recnum` lies within sensible limits. If the value is greater than the number of phrases currently held in the buffer, an invalid address will be returned with unpredictable consequences.

PTSEG

PTR00T	PTSEG
nphrases nchars nlines width lmargin ascent	phrases
pt_add_phrase pt_wrap pt_display_line pt_set_fonts pt_mod_by_num pt_mod_by_attr pt_find pt_inquire	pt_init pt_reset pt_append pt_put_lintab pt_pbuf
~~pt_init~~ ~~pt_reset~~ ~~pt_append~~ ~~pt_put_lintab~~ ~~pt_pbuf~~	

PTSEG is a subclass of PTR00T. It implements the buffer as an indexed array of variable length records, where each record is stored in its own heap cell. This is achieved by using an instance of the OLIB class VAXVAR, where each entry in the array contains data relating to a *single* phrase.

The use of a VAXVAR object allows efficient random access to phrases and is suitable for a "medium" number of phrases or for a large number of phrases where the maximum number is known.

The first entry (i.e. entry number 0) in the VAXVAR array is always reserved for the line-table, while subsequent entries are used for the phrases themselves.

All methods deferred by PTR00T are supplied here.

Class definition

The PTSEG class subclasses PTR00T and is defined in the sub-category file *polytext.cl* (with generated header file *polytext.g*).

```

CLASS ptseg ptroot
{
  REPLACE pt_init
  REPLACE pt_reset
  REPLACE pt_append      for "internal" use - by ptroot only
  REPLACE pt_put_lintab   for "internal" use - by ptroot only
  REPLACE pt_pbuf         for "internal" use - by ptroot only

  PROPERTY 1
  {
    PR_VAXVAR *phrases;
  }
}

```

Property

ptseg.phrases The handle of an instance of a VAXVAR class. The array is used to store phrase data. The first entry in the array is always reserved for the line-table.

PTSEG methods

PT_INIT

Initialise

INT pt_init(UINT granularity)

Initialise this instance of PTSEG.

The method takes a single parameter; *granularity* specifies the granularity of the VAXVAR object which implements the buffer.

The method creates an instance of VAXVAR and sets the handle into the property `ptseg.phrases`. The VAXVAR buffer object is initialised (with a granularity as specified in the parameter) by sending it a `VA_INIT` message.

A `VA_APPEND` message is then sent to the VAXVAR buffer object to add a minimum sized entry (a single zero filled byte) to the array; this will be the *first* entry in the array and is reserved for the line-table.

The method always returns zero.

PT_RESET

Reset

VOID pt_reset(VOID)

Reset the instance back to its initialised state.

The method effectively resets a number of `ptroot` properties and sends a `VA_RESET` message to the VAXVAR buffer object to delete *all* entries from the array.

A `VA_APPEND` message is then sent to add a minimum sized entry (a single zero filled byte) to the array; this will be the *first* entry in the array and is reserved for the line-table.

In effect, the method "empties" the PTSEG object of all text and deletes any existing line-table.

The properties which are reset to zero are: `ptroot.nphrases`, `ptroot.nchars`, `ptroot.nlines` and `ptroot.wwidth`.

PT_APPEND**Append a phrase**

```
VOID pt_append(RC_VAXVAR *pdscr)
```

Append a phrase to the buffer.

The method takes a single parameter; `pdscr` points to a data structure of type `RC_VAXVAR`. The members of this structure, `len` and `buf`, contain the length of the phrase and a pointer to the phrase content (a `PT_PHRASE` data structure) respectively. The `RC_VAXVAR` structure can be found in the `OLIB VAXVAR` class definition.

The method simply sends a `VA_APPEND` message to the `VAXVAR` buffer object, passing `pdscr` as a parameter, to add a new entry containing the phrase data.

PT_PUT_LINTAB**Store line-length table**

```
VOID pt_put_lintab(UBYTE *plintable)
```

Store the line-table in the buffer.

The method takes a single parameter; `plintable` contains the address of the line-table to be stored.

The line-table is always inserted as the first entry in the `VAXVAR` array which will have been reserved at initialisation time (by the `pt_init` method).

The method builds a `RC_VAXVAR` record descriptor; the `buf` member is set to point to the line-table while the `len` member is set to the length of the line-table (the value of `ptroot.nlines`).

The first entry in the array is replaced by the new line-table by sending a `VA_REPLACE` message to the `VAXVAR` buffer object, specifying record number zero and passing it the address of the record descriptor.

PT_PBUF**Get address of phrase**

```
PT_PHRASE *pt_pbuf(UINT recno)
```

Fetch the address of the specified phrase within the buffer.

The method takes a single parameter; `recno` contains the index of the phrase whose address is required. In other words, it is the relative position within the buffer of the required phrase.

The method sends a `VA_PBUF` message to the `VAXVAR` buffer object, specifying the record number `recno`; if `recno` is zero, the address of the line-table is returned - otherwise, the method returns the address of the required phrase. A `recno` value of one causes the address of the first phrase to be returned, a value of two causes the address of the second phrase to be returned, and so on.

Note, this method will panic if the value of `recno` is greater than the number of phrases currently held in the buffer

INDEX

- AO_ABRUN
 - PAGES class method, 4-24
- AO_INIT
 - PAGES class method, 4-21
 - WRAP class method, 3-14
- AO_QUEUE
 - PAGES class method, 4-24
 - WRAP class method, 3-15
- AO_RUN
 - PAGES class method, 4-23
 - WRAP class method, 3-15
- buffer
 - polytext classes, 7-1
- calendar image
 - classes, 6-1
- CALIMG class
 - CI_ADJUST_DATE method, 6-9
 - CI_EMPHASISE method, 6-7
 - CI_GOTO_DATE method, 6-9
 - CI_MOVE_CURSOR method, 6-8
 - CI_POS_MXY method, 6-10
 - CI_REDRAW method, 6-10
 - CI_SENSE method, 6-10
 - CI_SET_TITLE method, 6-7
 - CI_TODAY_CHANGED method, 6-11
 - CI_VIEW method, 6-10
 - CL_INIT method, 6-5
 - DESTROY method, 6-5
 - methods, 6-5
 - oop, 6-1
- call back
 - methods FORM library, 1-5
- CI_ADJUST_DATE
 - CALIMG class method, 6-9
- CI_EMPHASISE
 - CALIMG class method, 6-7
- CI_GOTO_DATE
 - CALIMG class method, 6-9
- CI_MOVE_CURSOR
 - CALIMG class method, 6-8
- CI_POS_MXY
 - CALIMG class method, 6-10
- CI_REDRAW
 - CALIMG class method, 6-10
- CI_SENSE
 - CALIMG class method, 6-10

- CI_SET_TITLE
 - CALIMG class method, 6-7
- CI_TODAY_CHANGED
 - CALIMG class method, 6-11
- CI_VIEW
 - CALIMG class method, 6-10
- CL_INIT
 - CALIMG class method, 6-5
- class
 - CALIMG, 6-1
 - EPDOC document filter, 2-6
 - EPDOC, 2-5
 - EPDOC pagination property, 2-6
 - EPFDOC, 2-14
 - FORMDOC, 2-2
 - mixin, 1-5
 - PAGELAY, 4-46
 - PAGES, 4-14
 - PDR, 4-34
 - PRINTER environment variables, 4-5
 - printer layout SCRLAY, 3-2
 - PRINTER, 4-2
 - PRNLAY, 4-49
 - PRNTPRV, 5-9
 - PRVPDR, 5-2
 - PTFLAT, 7-9
 - PTROOT, 7-2
 - PTSEG, 7-12
 - screen layout SCRLAY, 3-2
 - SCRIMG, 3-15
 - SCRLAY data structure, 3-7
 - SCRLAY font width tables, 3-8
 - SCRLAY, 3-2
 - SCRLAY screen diagram, 3-19
 - SCRLAY special characters, 3-7
 - SCRLAY text line diagram, 3-21
 - WDR, 4-25
 - WRAP, 3-14
- class diagrams
 - FORM hierarchy, 1-3
 - FORM library, 1-3
- classes
 - FORM library overview, 1-1
 - FORM using, 1-1
 - mixin, 1-5
- DESTROY
 - CALIMG class method, 6-5
 - PDR class method, 4-37
 - PRINTER class method, 4-6
 - PTFLAT class method, 7-10
 - SCRIMG class method, 3-19
 - SCRLAY class method, 3-10
 - WDR class method, 4-29
- document filter
 - EPDOC class, 2-6
- document forming
 - FORM library, 1-1
- document layout
 - classes, 3-1

- document printing
 - classes, 4-1
- DYL
 - FORM library introduction, 1-1
- environment variables
 - print manager, 4-5
- EP_BACK_CHARS
 - EPDOC class method, 2-9
- EP_CLEAR
 - EPDOC class method, 2-11
- EP_COMPRESS
 - EPDOC class method, 2-11
- EP_DELETE
 - EPDOC class method, 2-11
- EP_EXTRACT
 - EPDOC class method, 2-10
- EP_INIT
 - EPDOC class method, 2-8
- EP_INSERT
 - EPDOC class method, 2-10
- EP_SENSE_CHARS
 - EPDOC class method, 2-8
- EP_SENSE_LEN
 - EPDOC class method, 2-8
- EPDOC class
 - document filter, 2-6
 - EP_BACK_CHARS method, 2-9
 - EP_CLEAR method, 2-11
 - EP_COMPRESS method, 2-11
 - EP_DELETE method, 2-11
 - EP_EXTRACT method, 2-10
 - EP_INIT method, 2-8
 - EP_INSERT method, 2-10
 - EP_SENSE_CHARS method, 2-8
 - EP_SENSE_LEN method, 2-8
 - EPDOC_ENQ_PAGE method, 2-12
 - EPDOC_GOTO_PAGE method, 2-12
 - EPDOC_PARA_START call back method, 2-13
 - EPDOC_POS_FILTER method, 2-12
 - EPDOC_SENSE_CHARS call back method, 2-13
 - EPDOC_SET_FILTER method, 2-12
 - EPDOC_SET_PAGES method, 2-11
 - methods call-back, 2-13
 - methods, 2-8
 - oop, 2-5
 - pagination property, 2-6
- EPDOC_ENQ_PAGE
 - EPDOC class method, 2-12
- EPDOC_GOTO_PAGE
 - EPDOC class method, 2-12
- EPDOC_PARA_START
 - EPDOC class method call back, 2-13
- EPDOC_POS_FILTER
 - EPDOC class method, 2-12
- EPDOC_SENSE_CHARS
 - EPDOC class method call back, 2-13
- EPDOC_SET_FILTER
 - EPDOC class method, 2-12
- EPDOC_SET_PAGES
 - EPDOC class method, 2-11
- EPFDOC class
 - EPFDOC_PARA_START call back method, 2-15
 - EPFDOC_SENSE_CHARS call back method, 2-15
 - methods call-back, 2-15
 - oop, 2-14
- EPFDOC_PARA_START
 - EPFDOC class method call back, 2-15
- EPFDOC_SENSE_CHARS
 - EPFDOC class method call back, 2-15
- error handling
 - FORM library, 1-4
 - FORM library panics, 1-4
- FORM
 - call back methods, 1-5
 - class diagrams, 1-3
 - class hierarchy, 1-3
 - error handling, 1-4
 - error numbers panics, 1-4
 - library introduction, 1-1
 - long function parameters, 1-3
- FORM class
 - methods Series 3 notes, 4-53
- FORM classes
 - using, 1-1
- FORM library
 - function prototypes, 1-2
- form.dyl
 - ROM, 1-1
- formatted document content
 - classes, 2-1
- formatted text
 - classes, 3-1
- formatting
 - document FORM library, 1-1
 - printing FORM library, 1-1
- FORMDOC class
 - FORMDOC_ENQ_PAGE method, 2-5
 - FORMDOC_PARA_START method, 2-3
 - FORMDOC_SENSE_CHARS method, 2-3
 - FORMDOC_SENSE_PDATA method, 2-4
 - FORMDOC_SENSE_PLABEL method, 2-4
 - methods, 2-3
 - oop, 2-2
- FORMDOC_ENQ_PAGE
 - FORMDOC class method, 2-5
- FORMDOC_PARA_START
 - FORMDOC class method, 2-3
- FORMDOC_SENSE_CHARS
 - FORMDOC class method, 2-3
- FORMDOC_SENSE_PDATA
 - FORMDOC class method, 2-4
- FORMDOC_SENSE_PLABEL
 - FORMDOC class method, 2-4
- library
 - FORM DYL introduction, 1-1
 - form.dyl ROM, 1-1

-
- line-table
 - polytext classes, 7-1
 - long parameters
 - FORM functions, 1-3
 - FORM library functions, 1-3
 - measurement units
 - points, 1-2
 - printer units, 1-2
 - twips, 1-2
 - method function
 - FORM long parameters, 1-3
 - FORM prototypes, 1-2
 - methods
 - CALIMG class, 6-5
 - call back FORM library, 1-5
 - EPDOC class call-back, 2-13
 - EPDOC class, 2-8
 - EPFDOC class call-back, 2-15
 - FORM class Series 3 notes, 4-53
 - FORMDOC class, 2-3
 - PAGELAY class call back, 4-46
 - PAGES class, 4-21
 - PDR class, 4-37
 - PRINTER class, 4-6
 - PRNLAY class, 4-52
 - PRVPDR class, 5-5
 - PTFLAT class, 7-10
 - PTROOT class deferred, 7-8
 - PTROOT class, 7-4
 - PTSEG class, 7-13
 - SCRIMG class, 3-19
 - SCRLAY class, 3-8
 - WDR class, 4-29
 - WRAP class, 3-14
 - mixin classes
 - oop, 1-5
 - oop
 - calendar classes, 6-1
 - CALIMG class, 6-1
 - CALIMG class methods, 6-5
 - document layout classes, 3-1
 - document printing classes, 4-1
 - EPDOC class, 2-5
 - EPDOC class call-back methods, 2-13
 - EPDOC class methods, 2-8
 - EPDOC document filter class, 2-6
 - EPDOC pagination property class, 2-6
 - EPFDOC class, 2-14
 - EPFDOC class call-back methods, 2-15
 - FORM class methods Series 3 notes, 4-53
 - formatted document content classes, 2-1
 - formatted text classes, 3-1
 - FORMDOC class, 2-2
 - FORMDOC class methods, 2-3
 - PAGELAY class, 4-46
 - PAGELAY class call back methods, 4-46
 - PAGES class, 4-14
 - PAGES class methods, 4-21
 - PDR class, 4-34
 - PDR class methods, 4-37
 - polytext classes, 7-1
 - print preview classes, 5-1
 - PRINT PREVIEW classes, 5-1
 - PRINTER class, 4-2
 - PRINTER class diagram, 4-2
 - PRINTER class methods, 4-6
 - PRINTER environment variables, 4-5
 - printer layout SCRLAY class, 3-2
 - PRINTER measurement units, 4-2
 - PRNLAY class, 4-49
 - PRNLAY class methods, 4-52
 - PRNTPRV class, 5-9
 - PRVPDR class, 5-2
 - PRVPDR class methods, 5-5
 - PTFLAT class, 7-9
 - PTFLAT class methods, 7-10
 - PTROOT class, 7-2
 - PTROOT class deferred methods, 7-8
 - PTROOT class methods, 7-4
 - PTSEG class, 7-12
 - PTSEG class methods, 7-13
 - screen layout SCRLAY class, 3-2
 - SCRIMG class, 3-15
 - SCRIMG class methods, 3-19
 - SCRLAY class, 3-2
 - SCRLAY class methods, 3-8
 - SCRLAY data structures class, 3-7
 - SCRLAY font width tables class, 3-8
 - SCRLAY screen diagram, 3-19
 - SCRLAY special characters class, 3-7
 - SCRLAY text line diagram, 3-21
 - text formatting classes, 3-1
 - WDR class, 4-25
 - WDR class methods, 4-29
 - WRAP class, 3-14
 - WRAP class methods, 3-14
 - page dimensions
 - diagram, 4-20
 - PAGELAY class
 - methods call back, 4-46
 - oop, 4-46
 - PAGELAY_MDONE method, 4-47
 - PAGELAY_MREAD method, 4-46
 - PAGELAY_MDONE
 - PAGELAY class method, 4-47
 - PAGELAY_MREAD
 - PAGELAY class method, 4-46
 - PAGES class
 - AO_ABRUN method, 4-24
 - AO_INIT method, 4-21
 - AO_QUEUE method, 4-24
 - AO_RUN method, 4-23
 - methods, 4-21
 - oop, 4-14
 - pagination
 - EPDOC class property, 2-6
 - panics
 - FORM error numbers, 1-4
 - parallel port
 - letter types, 4-8
-

- PDR class
 - DESTROY method, 4-37
 - methods, 4-37
 - oop, 4-34
 - PDR_ADD_COMMAND method, 4-41
 - PDR_DESTROY method, 4-42
 - PDR_END method, 4-42
 - PDR_FONT method, 4-44
 - PDR_INIT method, 4-37
 - PDR_LINE method, 4-43
 - PDR_PAGE method, 4-43
 - PDR_PRINT method, 4-38
 - PDR_RIGHT method, 4-44
 - PDR_START method, 4-42
 - PDR_STYLE method, 4-45
 - PDR_TEXT method, 4-43
- PDR_ADD_COMMAND
 - PDR class method, 4-41
- PDR_DESTROY
 - PDR class method, 4-42
 - PRVPDR class method, 5-7
- PDR_END
 - PDR class method, 4-42
- PDR_FONT
 - PDR class method, 4-44
 - PRVPDR class method, 5-8
- PDR_INIT
 - PDR class method, 4-37
 - PRVPDR class method, 5-5
- PDR_LINE
 - PDR class method, 4-43
- PDR_PAGE
 - PDR class method, 4-43
 - PRVPDR class method, 5-8
- PDR_PRINT
 - PDR class method, 4-38
 - PRVPDR class method, 5-6
- PDR_RIGHT
 - PDR class method, 4-44
- PDR_START
 - PDR class method, 4-42
 - PRVPDR class method, 5-7
- PDR_STYLE
 - PDR class method, 4-45
- PDR_TEXT
 - PDR class method, 4-43
- phrase
 - polytext classes, 7-1
- points
 - printer measurement units, 1-2
- polytext class
 - buffer, 7-1
 - line-table, 7-1
 - phrase, 7-1
- polytext classes
 - oop, 7-1
- port parallel
 - letter types, 4-8
- port serial
 - letter types, 4-8
- port type
 - printer, 4-8
- PR_CLOSE_WDR
 - PRINTER class method, 4-10
- PR_GET_HD
 - PRINTER class method, 4-10
- PR_GET_PARAMS
 - PRINTER class method, 4-9
- PR_INIT
 - PRINTER class method, 4-6
- PR_OPEN_PORT
 - PRINTER class method, 4-10
- PR_OPEN_WDR
 - PRINTER class method, 4-10
- PR_PAGINATE
 - PRINTER class method, 4-11
- PR_PORT_DATA
 - PRINTER class method, 4-7
- PR_PREVIEW
 - PRINTER class method, 4-13
- PR_PREVIEW_DATA
 - PRINTER class method, 4-14
- PR_PREVIEW_END
 - PRINTER class method, 4-13
- PR_PREVIEW_START
 - PRINTER class method, 4-12
- PR_PRINT
 - PRINTER class method, 4-11
- PR_SENSE_MODEL
 - PRINTER class method, 4-9
- PR_SENSE_PORT
 - PRINTER class method, 4-8
- PR_SET_HD
 - PRINTER class method, 4-9
- PR_SET_MODEL
 - PRINTER class method, 4-7
- PR_SET_PORT_TYPE
 - PRINTER class method, 4-7
- PR_STORE_FILE
 - PRINTER class method, 4-6
- PR_STORE_SRCHAR
 - PRINTER class method, 4-6
- print manager
 - environment variables, 4-5
- print preview
 - classes, 5-1
- PRINT_PREVIEW class
 - oop, 5-1
- PRINTER class
 - DESTROY method, 4-6
 - diagram, 4-2
 - document printing, 4-1
 - environmet variables, 4-5
 - measurement units, 4-2
 - methods, 4-6
 - oop, 4-2
 - PR_CLOSE_WDR method, 4-10
 - PR_GET_HD method, 4-10
 - PR_GET_PARAMS method, 4-9
 - PR_INIT method, 4-6

-
- PR_OPEN_PORT method, 4-10
 - PR_OPEN_WDR method, 4-10
 - PR_PAGINATE method, 4-11
 - PR_PORT_DATA method, 4-7
 - PR_PREVIEW method, 4-13
 - PR_PREVIEW_DATA method, 4-14
 - PR_PREVIEW_END method, 4-13
 - PR_PREVIEW_START method, 4-12
 - PR_PRINT method, 4-11
 - PR_SENSE_MODEL method, 4-9
 - PR_SENSE_PORT method, 4-8
 - PR_SET_HD method, 4-9
 - PR_SET_MODEL method, 4-7
 - PR_SET_PORT_TYPE method, 4-7
 - PR_STORE_FILE method, 4-6
 - PR_STORE_SRCHAR method, 4-6
 - text printing, 4-1
 - printer layout
 - SCRLAY class, 3-2
 - printer model
 - get type, 4-9
 - printer port
 - device types, 4-8
 - printing
 - formatting FORM library, 1-1
 - page dimensions diagram, 4-20
 - PRINTING
 - PREVIEW class, 5-1
 - PRNLAY class
 - methods, 4-52
 - oop, 4-49
 - SL_PRINT_POS method, 4-52
 - SL_PRINT_READ method, 4-52
 - PRNTPRV class
 - oop, 5-9
 - PRNTPRV_MDONE method, 5-9
 - PRNTPRV_MDONE
 - PRNTPRV class method, 5-9
 - PRVPDR class
 - methods, 5-5
 - oop, 5-2
 - PDR_DESTROY method, 5-7
 - PDR_FONT method, 5-8
 - PDR_INIT method, 5-5
 - PDR_PAGE method, 5-8
 - PDR_PRINT method, 5-6
 - PDR_START method, 5-7
 - PT_ADD_PHRASE
 - PTROOT class method, 7-4
 - PT_APPEND
 - PTFLAT class method, 7-11
 - PTROOT class method deferred, 7-8
 - PTSEG class method, 7-13
 - PT_DISPLAY_LINE
 - PTROOT class method, 7-5
 - PT_FIND
 - PTROOT class method, 7-7
 - PT_INIT
 - PTFLAT class method, 7-10
 - PTROOT class method deferred, 7-8
 - PTSEG class method, 7-13
 - PT_INQUIRE
 - PTROOT class method, 7-8
 - PT_MOD_BY_ATTRIB
 - PTROOT class method, 7-7
 - PT_MOD_BY_NUM
 - PTROOT class method, 7-6
 - PT_PBUF
 - PTFLAT class method, 7-11
 - PTROOT class method deferred, 7-9
 - PTSEG class method, 7-14
 - PT_PUT_LINTAB
 - PTFLAT class method, 7-11
 - PTROOT class method deferred, 7-9
 - PTSEG class method, 7-13
 - PT_RESET
 - PTFLAT class method, 7-11
 - PTROOT class method deferred, 7-8
 - PTSEG class method, 7-13
 - PT_SET_FONTS
 - PTROOT class method, 7-5
 - PT_WRAP
 - PTROOT class method, 7-4
 - PTFLAT class
 - DESTROY method, 7-10
 - methods, 7-10
 - oop, 7-9
 - PT_APPEND method, 7-11
 - PT_INIT method, 7-10
 - PT_PBUF method, 7-11
 - PT_PUT_LINTAB method, 7-11
 - PT_RESET method, 7-11
 - PTROOT class
 - methods deferred, 7-8
 - methods, 7-4
 - oop, 7-2
 - PT_ADD_PHRASE method, 7-4
 - PT_APPEND deferred method, 7-8
 - PT_DISPLAY_LINE method, 7-5
 - PT_FIND method, 7-7
 - PT_INIT deferred method, 7-8
 - PT_INQUIRE method, 7-8
 - PT_MOD_BY_ATTRIB method, 7-7
 - PT_MOD_BY_NUM method, 7-6
 - PT_PBUF deferred method, 7-9
 - PT_PUT_LINTAB deferred method, 7-9
 - PT_RESET deferred method, 7-8
 - PT_SET_FONTS method, 7-5
 - PT_WRAP method, 7-4
 - PTSEG class
 - methods, 7-13
 - oop, 7-12
 - PT_APPEND method, 7-13
 - PT_INIT method, 7-13
 - PT_PBUF method, 7-14
 - PT_PUT_LINTAB method, 7-13
 - PT_RESET method, 7-13
 - resource files
 - WDR printing, 4-25
-

- ROM
 - form.dyl, 1-1
- screen layout
 - SCRLAY class, 3-2
- SCRIMG class
 - DESTROY method, 3-19
 - methods, 3-19
 - oop, 3-15
 - SI_DELPREP method, 3-26
 - SI_DOC_CHANGED method, 3-25
 - SI_DOC_RESET method, 3-25
 - SI_EMPHASIZE method, 3-22
 - SI_FWD_CHANGE method, 3-27
 - SI_GET_SELECT method, 3-22
 - SI_INIT method, 3-20
 - SI_MOVE_CURSOR method, 3-23
 - SI_PAN method, 3-22
 - SI_PARA_CHANGED method, 3-26
 - SI_REDRAW method, 3-25
 - SI_SCROLL method, 3-23
 - SI_SENSE method, 3-22
 - SI_SET method, 3-20
 - SI_STYLE_CHANGED method, 3-26
 - SI_VIEW method, 3-23
- SCRLAY class
 - data structures, 3-7
 - DESTROY method, 3-10
 - document layout, 3-1
 - font width tables, 3-8
 - methods, 3-8
 - oop, 3-2
 - screen diagram, 3-19
 - SL_BEGIN_READ method, 3-11
 - SL_DISCARD_LAYOUT method, 3-13
 - SL_FORMAT_LINE method, 3-12
 - SL_INIT method, 3-8
 - SL_LINE_ENDS method, 3-11
 - SL_PARA_CHANGED method, 3-13
 - SL_POS_TO_XL method, 3-10
 - SL_READ method, 3-12
 - SL_RESCALE method, 3-13
 - SL_SCROLL method, 3-12
 - SL_SENSE method, 3-10
 - SL_SET method, 3-9
 - SL_SET_LINES method, 3-13
 - SL_VIEW method, 3-12
 - SL_XL_TO_POS method, 3-11
 - special characters, 3-7
 - text layout, 3-1
 - text line diagram, 3-21
- serial port
 - letter types, 4-8
- Series 3
 - FORM class notes, 4-53
- SI_DELPREP
 - SCRIMG class method, 3-26
- SI_DOC_CHANGED
 - SCRIMG class method, 3-25
- SI_DOC_RESET
 - SCRIMG class method, 3-25
- SI_EMPHASIZE
 - SCRIMG class method, 3-22
- SI_FWD_CHANGE
 - SCRIMG class method, 3-27
- SI_GET_SELECT
 - SCRIMG class method, 3-22
- SI_INIT
 - SCRIMG class method, 3-20
- SI_MOVE_CURSOR
 - SCRIMG class method, 3-23
- SI_PAN
 - SCRIMG class method, 3-22
- SI_PARA_CHANGED
 - SCRIMG class method, 3-26
- SI_REDRAW
 - SCRIMG class method, 3-25
- SI_SCROLL
 - SCRIMG class method, 3-23
- SI_SENSE
 - SCRIMG class method, 3-22
- SI_SET
 - SCRIMG class method, 3-20
- SI_STYLE_CHANGED
 - SCRIMG class method, 3-26
- SI_VIEW
 - SCRIMG class method, 3-23
- SL_BEGIN_READ
 - SCRLAY class method, 3-11
- SL_DISCARD_LAYOUT
 - SCRLAY class method, 3-13
- SL_FORMAT_LINE
 - SCRLAY class method, 3-12
- SL_INIT
 - SCRLAY class method, 3-8
- SL_LINE_ENDS
 - SCRLAY class method, 3-11
- SL_PARA_CHANGED
 - SCRLAY class method, 3-13
- SL_POS_TO_XL
 - SCRLAY class method, 3-10
- SL_PRINT_POS
 - PRNLAY class method, 4-52
- SL_PRINT_READ
 - PRNLAY class method, 4-52
- SL_READ
 - SCRLAY class method, 3-12
- SL_RESCALE
 - SCRLAY class method, 3-13
- SL_SCROLL
 - SCRLAY class method, 3-12
- SL_SENSE
 - SCRLAY class method, 3-10
- SL_SET
 - SCRLAY class method, 3-9
- SL_SET_LINES
 - SCRLAY class method, 3-13
- SL_VIEW
 - SCRLAY class method, 3-12
- SL_XL_TO_POS
 - SCRLAY class method, 3-11

- text display
 - polytext classes, 7-1
- text formatting
 - classes, 3-1
- twips
 - printer measurement units, 1-2
- WDR
 - resource files printing, 4-25
- WDR class
 - DESTROY method, 4-29
 - methods, 4-29
 - oop, 4-25
 - WDR_COUNT_MODELS method, 4-30
 - WDR_FONT_HEIGHT method, 4-31
 - WDR_GET_WIDTH_TABLE method, 4-32
 - WDR_INIT method, 4-29
 - WDR_LOAD_RECORD method, 4-34
 - WDR_OPEN_PRINT method, 4-33
 - WDR_SEARCH_HEIGHT method, 4-32
 - WDR_SEARCH_TYPEFACE method, 4-31
 - WDR_SENSE_MODEL method, 4-30
 - WDR_SENSE_MODEL_NAME method, 4-30
 - WDR_SENSE_WIDTH method, 4-33
 - WDR_SET_MODEL method, 4-30
 - WDR_TWIPS_TO_XY method, 4-33
 - WDR_TYPEFACE method, 4-30
- WDR_COUNT_MODELS
 - WDR class method, 4-30
- WDR_FONT_HEIGHT
 - WDR class method, 4-31
- WDR_GET_WIDTH_TABLE
 - WDR class method, 4-32
- WDR_INIT
 - WDR class method, 4-29
- WDR_LOAD_RECORD
 - WDR class method, 4-34
- WDR_OPEN_PRINT
 - WDR class method, 4-33
- WDR_SEARCH_HEIGHT
 - WDR class method, 4-32
- WDR_SEARCH_TYPEFACE
 - WDR class method, 4-31
- WDR_SENSE_MODEL
 - WDR class method, 4-30
- WDR_SENSE_MODEL_NAME
 - WDR class method, 4-30
- WDR_SENSE_WIDTH
 - WDR class method, 4-33
- WDR_SET_MODEL
 - WDR class method, 4-30
- WDR_TWIPS_TO_XY
 - WDR class method, 4-33
- WDR_TYPEFACE
 - WDR class method, 4-30
- WRAP class
 - AO_INIT method, 3-14
 - AO_QUEUE method, 3-15
 - AO_RUN method, 3-15
 - methods, 3-14
 - oop, 3-14