1 Introduction

1.1 Purpose of this document

The purpose of this document is to provide initial programming information for the OpenPaige developers. Comments are welcome, as are useful example code submissions. Questions and answers from OpenPaige developers may be used in subsequent editions of the manual.

This function sets a new tab that applies to the specified selection.

1.2 How to use this manual

The OpenPaige technology is quite extensive, so we recommend that you do not simply dive into the middle of this manual and start implementing complex features.

Our advice is to implement this software by following these gradient steps:

1. Follow the information in chapter 2, "Up & Running". During this phase, ignore all other information in the manual.
2. Follow chapter 3, "Beyond the Defaults", which discusses implementation of additional, common features above and beyond the bare minimum covered in #1 above.
3. If you need to implement virtual memory, do that by following chapter 4, "Virtual Memory".
4. Implement all remaining simple functionality not covered in #1 or #2 above, such as text formatting (fonts and styles), paragraph formatting (indents and justification) and possibly tab settings and color. See chapter 8, "Style Basics".
5. Depending on what you wish to accomplish with OpenPaige, find section(s) that deal with your particular requirements — we have tried to break down this manual into the most likely application requirements.

You should also consult the index to locate the topic(s) of interest quickly.

Generally, we have placed the parts of OpenPaige that most users will want and that are the most

straight forward in the front. As you move to the back of the manual, the functionality will become more complex.

CAUTION

It is important to remember that no user will need the entire functionality. If you are contemplating a complex feature, or one in which you will need detailed knowledge of OpenPaige or working in the chapters toward the rear of the manual, please contact OpenPaige Tech Support via electronic mail for an evaluation and suggestions on how you can easily accomplish your goal. We can often suggest the easiest way to do something if we are consulted before you are buried in buggy code. Also knowing what you are doing and why you are doing it "that way" helps us to build better features.

1.3 Implementation Tips & Hints

- If you are a Word Solution Engine customer: the OpenPaige technology is very different than DataPak's Word Solution Engine. We therefore recommend strongly to "forget" all you know about Word Solution in order to understand the implementation of OpenPaige.
- Use the index to find small items, and Summary of Functions for quick-reference to function syntax.
- Consult the demo program. The OpenPaige package you received includes all the source files for the "demo" which contains a wealth of information and examples. If you think something does not work correctly, before reporting a bug or otherwise reach an impasse, consult that area of the demo against the way you have implemented the code. One of the first questions we will ask when you contact our Technical Support is, "Does it work correctly in the demo?"

NOTE (WINDOWS USERS)

If you are using the OpenPaige API directly, consult the source files in the Control directory (the "demo" simply uses the OpenPaige Custom Control; the Custom Control source files show how to access the API).

NOTE

You may contact our Technical Support service if the above suggestions fail to help. However, we do not accept any telephone support questions whatsoever. All questions must be submitted by email; we will always attempt to handle your questions as quickly and as thoroughly as possible. You can email your support questions to .

1.4 Certain Conventions

Since OpenPaige is designed to be a multi-platform, multi-application processing editing library, we have had to make certain conventions in how the functions are described.

"FAR" Pointers

Certain platforms require pointers which are outside the current segment to be designated as `far` pointers, such as Windows. Other platforms, such as the Macintosh do not require this. For the Macintosh, `PG_FAR` has been declared as nothing and these differences can be ignored.

`pascal` keyword

The `pascal` keyword has been left out of the function definitions in this document; the actual header file(s) will contain that keyword. All external OpenPaige functions are declared using the Pascal calling conventions.

Redefinition of types

To maintain compatibility across all platforms, certain new types have been declared as follows:

Unicode Version

OpenPaige Type	Typedef'd From
`pg_short_t`	unsigned short
`pg_char`	unsigned short
`pg_char_ptr`	pointer to `pg_char`

pg_bits8	unsigned char
pg_bits8_ptr	pointer to pg_bits8
pg_boolean	short
pg_error	short (for error codes)
memory_ref	unsigned long
PGSTR	pg_char_ptr

Non-Unicode Version

OpenPaige Type	Typedef'd From
pg_short_t	unsigned short
pg_char	unsigned char
pg_char_ptr	pointer to pg_char
pg_bits8	(same as pg_char)
pg_bits8_ptr	(same as pg_char_ptr)
pg_boolean	short
pg_error	short (for error codes)
memory_ref	unsigned long
PGSTR	pg_char_ptr

NULL Reference

Frequent use of the term M_NULL exists throughout this manual. This is an OpenPaige macro that simply expands to (value of) zero. It is used for indicating a "null" for an OpenPaige memory reference.

Machine Definitions

A single header file, cpudefs.h controls basic definitions for the platform in which the source files are intended.

1.5 Debug Mode

Windows users can ignore the "debug mode" libraries described below. This method of debugging applies only to Macintosh versions.

OpenPaige is compiled in both "debug" and "non debug" modes. Two sets of libraries are provided for this purpose.

When you use the "debug" libraries, you must also include `pgdebug.c` in your project. This lets you break into a source-level debugger to learn why OpenPaige is raising an exception. To use the OpenPaige debugger, open `pgdebug.c` and place a break point at the suggested spot (source comments indicate the spot).

If you break into the debugger, the message parameter is a `pascal` string.

Source code users: Debug mode is controlled by a single `#define` in `CPUDefs.h`, `#define PG_DEBUG`.

Debug mode slows the performance down substantially. It is recommended to use OpenPaige in debug mode during your development, but to turn it off for your final release, if for no other reason than increased performance.

2 Up & running

2.1 OpenPaige Custom Control

If you are using OpenPaige for a new application, or integrating OpenPaige for the first time, it would be wise to consider implementing the OpenPaige Custom Control. Documentation for this subset of OpenPaige is contained in a separate, smaller manual.

The Custom Control can potentially save you substantial amounts of development time, particularly to get "up and running" quickly. To make this decision, consider the following:

- Using the Control immediately eliminates the need to know very little — or any — of the detailed information in this Programmer's Guide.
- Most of the samples we provide use the Control (not the direct API).
- You can still call the OpenPaige API directly, when and if you need to.

If you decide to use the OpenPaige Custom Control, you do not need to read this manual any further! Immediately proceed to the OpenPaige Control manual; use this (larger) Programmer's

Guide only when/if you need to call the API directly.

2.2 Bare necessities

This section provides the bare minimum code to get up and running with OpenPaige. This minimum functionality assumes one single default font and style, a single rectangle for display and word wrapping, no scrolling, nothing fancy.

CAUTION

Be sure to consult the release note and individual installation instructions included in each release. OpenPaige installation will change with versions and even interim releases. This makes checking the latest notes on the disk critical.

2.3 Libraries & Headers

Regardless of whether you are a source code user or an object-code-only user, all source files in your application that call OpenPaige functions must include, at a minimum:

```
#include "Paige.h"
```

As for the OpenPaige software itself, the minimum configuration is given below.

Windows

The Windows version provides several library options; choose the appropriate libraries based upon the information provided below.

NOTE

Most libraries include the option between DLL(s) and static libraries.

Windows 3.1

Multilingual (will handle double-byte codes such as Kanji)

```
(DLL Version Only)

PGML16.DLL (Main OpenPaige)
PGMLCT16.DLL (Custom control)
```

Non-Multilingual (no requirements for double-byte codes)

```
DLL Libraries
^^^^^^^^^^^^^
PAIGE.DLL (Main OpenPaige)
PGCNTL.DLL (Custom control)

Static Libraries
^^^^^^^^^^^^^^^^
PG16LIB.LIB (Main OpenPaige)
PGCTL16.LIB (Custom control)
```

Windows NT (XP, 10, 11)

Unicode

```
DLL Libraries
^^^^^^^^^^^^^
PGUNICOD.DLL (Main OpenPaige)
PGUNICTL.DLL (Custom control)

Static Libraries
^^^^^^^^^^^^^^^^
PGUNILIB.LIB (Main OpenPaige)
PGUNCTLB.LIB (Custom control)
```

Non-Unicode

```
DLL Libraries
^^^^^^^^^^^^^

Paige32.DLL (Dynamic Linked Library for main (OpenPaige)
Pgentl32.DLL (Dynamic Linked Library for custom control)

Static Libraries
^^^^^^^^^^^^^^^^

PGLIB32.LIB (Main OpenPaige)
PGCTLLIB.LIB (Custom control)
```

Multilingual

All versions for Windows 95 and NT are multilingual-compatible.

Borland Libraries (DLL libraries only)

```
Single Thread
^^^^^^^^^^^^^

PAIGE32B.DLL (Main OpenPaige)
PGCTL32B.DLL (Custom Control)

Multithread
^^^^^^^^^^^

PG32BMT.DLL (Main OpenPaige)
PGC32BMT.DLL (Custom control)
```

Program Linking with DLL Libraries

When using any of the DLL libraries, add the file with the same name plus the ".LIB" extension. For example, if using `PAIGE.DLL` for the runtime library, add `PAIGE.LIB` to your project.

Macintosh

Macintosh Object Code Users

If you are using Think C or Metrowerks CodeWarrior, add all libraries to your project from the "Debug Libraries" OR "Runtime Libraries" folder (not both). Running in debug mode is suggested for general development, while non-debug is suggested for performance testing (for speed) and/or for final release of your product. Debug mode will reduce the program's performance substantially.

If you are using Metrowerks CodeWarrior, you must be sure to remove all previous versions of header files. Compiler complaints may be the result of CodeWarrior finding the incorrect header or object file.

The source code package includes "make" files for building OpenPaige libraries with MSVC++. If you need to create your own project file to build an OpenPaige library, the following information may prove useful:

1. Include `.C` files from the `pgsource` directory. None of them should be excluded.
2. Include `pgdebug` from the `pgdebug` directory. (**NOTE:** This file compiles to zero bytes of code unless `#define PG_DEBUG` is present in `CPUDEFS.H` [see “Compiler Options" below].)
3. Include the following `.C` files from `pgplatfo` regardless of the target platform: `pgio.c`, `pgmemmgr.c`, `pgosutl.c`, `pgscrap.c`.
4. Depending upon your target platform, include the following files from `pgplatfo`: `pgwin.c` and `pgdll.c` (the latter if compiling as a DLL) for Windows, and `pgmac.c` and `pgmacput.c` for Macintosh.
5. For **Windows 3.1** you may be asked to include a `.DEF` file. With MSVC 1.5x you can ask to generate a default `.DEF`, in which case you should choose to do so and rebuild.
6. The OpenPaige source code is not always friendly to certain C++ compilers due to `void*` type casting (or lack thereof). In most cases, you can work around this problem by compiling your project as straight C with an output for static or dynamically-linked library, then include that library in your main project. For Metrowerks CodeWarrior (Macintosh) you can work around this problem by turning OFF the option, "Invoke C++ Compiler".
7. To compile for Unicode, define `UNICODE` and `_UNICODE` in the preprocessor option(s). Do not define these constants in the header file(s) or you won't necessary achieve an accurate Unicode library.
8. If you compile for **Windows 3.1-Multilingual**, you must also include the following Windows library (for National Language Support): `OLENLS.LIB`.

Compiler Options

All options for different target platforms and library types are controlled in `CPUDEFS.H`. Generally, only the first several lines in `CPUDEFS.H` need to be changed to compile for different platforms. The following guidelines should be followed:

Compiling for Windows 3.1

```
#define WIN16_COMPILE (should be ON)
#define WIN32_COMPILE (should be OFF)
```

Compiling for Windows 3.1-Multilingual (double-byte)

In addition to above:

```
#define WIN_MULTILINGUAL (should be added to the file or preprocessor)
```

Compiling for Windows NT (7, 8, 10, 11)

```
#define WIN16 COMPILE (should be OFF)
#define WIN32_COMPILE (should be ON)
```

NOTE

There are other miscellaneous options that may imply a requirement to be enabled (by their names) such as `WIN95_COMPILE`. Do not turn these on, regardless of platform! Enable only `WIN32_COMPILE` for all 32-bit versions.

You do not need to define anything other than WIN32_COMPILE to support doublebyte multilingual editing for Windows NT and Windows 95; that support is generated automatically.

For Unicode, you must define `UNICODE` and `_UNICODE` in your preprocessor options of the compiler. (If no preprocessor option, `#define UNICODE` somewhere in your sources or headers to allow all system header files to recognize the Unicode option).

DLL versus Static Library (all platforms)

To compile as a DLL:

```
#define CREATE_MS_DLL (should be ON)
```

If compiling as a static library or non-DLL:

```
#define CREATE_MS_DLL (should be OFF)
```

Debug versus Runtime

OpenPaige has a built-in debugger which can be enabled by compiling with the following:

```
#define PG_DEBUG (OpenPaige debugger compiles if ON)
```

When this is defined, all OpenPaige exceptions or debugging errors jump into the code in `pgdebug.c`.

NOTE

Compiling with `PG_DEBUG` will dramatically reduce the performance!

Special Resource (Macintosh only)

A special resource has been provided on your OpenPaige disc which the Macintosh-specific code within OpenPaige uses to initialise default character values (such as arrow keys, backspace characters, invisible symbols, etc.). You may copy and paste this resource into your application's resource and you may modify its contents if you want different defaults.

This resource is not required to use OpenPaige successfully. If it is missing, initialisation simply sets a hard-coded set of defaults.

See also Changing Globals.

2.4 Software Startup

Some place early in your application you need to initialise the OpenPaige software; the recommended place to do so is after all other initialisations have been performed for the main menu, Mac Toolbox, etc. To initialise, you need to reserve a couple blocks of memory that OpenPaige can use to store certain global variables (OpenPaige does not use any `globals` and therefore requires you to provide areas it can use to store required global structures).

To initialise OpenPaige you must call two functions in the order given:

```
#include "Paige.h"
(void) pgMemStartup (pgm_globals_ptr mem_globals, long max_memory);
(void) pgInit (pg_globals_ptr globals, pgm_globals_ptr mem_globals);
```

Calling `pgMemStartup` initialises OpenPaige's allocation manager. This call must be made first before `pgInit`. The `mem_globals` parameter must be a pointer to an area of memory which you provide. The usual (and easiest) method of doing this is to define a global variable that will not relocate or unload during the execution of your program, such as the following:

```
pgm_globals     memrsrv; // ←—somewhere that will NOT unload
```

You do not need to initialise this structure to anything—`pgMemStartup` initialises this structure appropriately.

`max_memory` should contain the maximum amount of memory OpenPaige is allowed to use before purging memory allocations. If you want OpenPaige to have access to all available memory (which is *strongly recommended*), pass 0 for `max_memory`.

For example, suppose you only wanted to use 200 kB of memory for all OpenPaige documents, combined. In this case, you would pass 200000 to pgInit. If you don't care, or want it to use all memory available, you would pass 0.

After `pgMemStartup`, call `pgInit`, which initialises every other part of OpenPaige.

`globals` is a pointer to an area of memory which you provide. The usual (and easiest) method of doing this is to define a global variable that will not relocate or unload during the execution of your program, such as the following:

```
pg_globals  paigersrv; // ⟵ somewhere that will NOT unload
```

The structure `pg_globals` is defined in `paige.h` (and shown in Changing Globals). You do not need to initialise this structure to anything—OpenPaige will initialise the globals structure as required. It is only necessary that you provide the space for this structure and pass a pointer to it in `pgInit`.

`mem_globals` parameter in `pgInit` must be a pointer to the same structure passed to `pgMemStartup`.

MFC NOTE

The best place to initialise OpenPaige in the constructor of the `CWinApp` derived class. Also the best place to put the OpenPaige globals and memory globals is in the `CWinApp` derived class.

EXAMPLE

```
(.H)

class MyWinApp : public CWinApp
{
    ...
    public:
    pgm_globals m_MemoryGlobals;
    pg_globals m_Globals;
    ...
}

(.CPP)

MyWinApp::MyWinApp()
{
    pgMemStartup(&m_MemoryGlobals, 0);
    pgInit(&m_Globals, &m_MemoryGlobals);
```

```
    ...
}
```

TECH NOTE

pgInit crashes

It is possible to crash in pgInit. This is very rare however. Here are the main possibilities:

- A wrong library is linked in, i.e. version mismatch. (This includes all "updates" from compiler vendors who have changed the format of their object code libraries).
- It is called without calling MemStartup.
- You are out of memory. OpenPaige can require up to 60 kB to build itself and get ready to accept text.
- **Windows 3.1 platform only**: you are building a DLL with a memory model mismatch. The PAIGE DLL was built for large modal; try building your DLL the same.

2.5 OpenPaige Shutdown

For applications that require a shutdown of all allocations it has created, call the following functions, in the order shown, before terminating your application:

```
(void) pgShutdown (pg_globals_ptr globals);
(void) pgMemShutdown (pgm_globals_ptr mem_globals);
```

globals and mem_globals parameters must be pointers to the same structures given to pgInit and pgMemStartup, respectively. After pgShutdown, you must not call any OpenPaige functions (except for pgInit). After pgMemShutdown, all allocations placed in globals are de-allocated.

CAUTION

All pg_refs and all memory references allocated anywhere by OpenPaige become invalid after pgShutdown, so make sure this is the very last OpenPaige function you call.

CAUTION (WINDOWS USERS)

Be sure to call both `pgShutdown` and `pgMemShutdown`, in that order, before `EXIT`, or you will have memory leaks and resources that are never released.

NOTES

- `pgShutdown` and `pgMemShutdown` actually dispose every memory allocation made by OpenPaige since `pgMemStartup`; you therefore don't really need to dispose any `pg_ref`s, `shape_ref`s or other OpenPaige allocations.
- You must not call either shutdown function if you are using the OpenPaige Control.
- For Macintosh applications, the shutdown procedure is completely unnecessary if you will be doing an `ExitToShell` using the app version. Mac developers working with code resource libraries will still need to call `pgShutdown` and `pgMemShutdown`.
- For Microsoft Foundation Class applications, the appropriate method to shut down OpenPaige is to override `CxxAppxExitInstance()` and call `::pgShutdown` and `::pgMemShutdown`.

- The best place to shutdown OpenPaige is in the destructor of the `CWinApp` derived class. Example:

```
(.CPP)
MyWinApp::~MyWinApp()
{
     ...
     pgShutdown(&m_Globals);
     pgMemShutdown(m_MemoryGlobals);
}
```

2.6 Creating an OpenPaige Object

By "OpenPaige object" is meant a single item that can edit, display and otherwise manipulate a block of text, large or small.

Calling `pgNew`, below, returns a reference of type `pg_ref`. This `pg_ref` can then be passed to all the other functions given in this manual.

```
(pg_ref) pgNew (pg_globals_ptr globals, generic_var def_device, shape_ref
vis_area, shape_ref page_area, shape_ref exclude_area, long attributes);
```

The above function returns a new `pg_ref`; the `pg_ref` can then be passed to other functions to insert text and edit text.

`globals` parameter must be a pointer to the same `pg_globals` structure you passed to `pgInit` at startup time.

Attributes are described in Attribute Settings and Changing Attributes, but can be set here as well.

`def_device` parameter defines what graphics port this OpenPaige object should draw to by default; what is actually passed to `def_device` can slightly vary between platforms as follows:

Macintosh & PowerPC

If `def_device` is `NULL` then current `GrafPort` is used as the default device; if `def_device` is non-`NULL` and not "-1" it is assumed to be a `GrafPtr` and that port is used for subsequent drawing.

Windows (PC)

If `def_device` is `0L` then the current window of focus is used as the default window where drawing will occur (e.g., `GetFocus` is used to determine the window); if `def_device` is non-`NULL` and not `-1` it is assumed to be type `HWND` and that window is used for subsequent drawing.

This `HWND` in the def_device is NOT a Device Context.

Essentially, the `dev_device` should be the window (or child window) that is receiving the message to create the OpenPaige object, e.g. `WM_CREATE`.

CAUTION

If you pass `MEM_NULL` to `def_device`, OpenPaige will obtain the window of current focus. You should only use this method if your document window is known to be the window of focus, otherwise passing `MEM_NULL` can result in a crash.

Microsoft Foundation Classes (MFC)

The best place to put pgNew() is in the OnCreate() member of the CView derived class. It is important to call the CView::OnCreate() **before** calling pgNew(). Examples follow:

```
(.H)
class MyView : public CView
{
...
public:
    pg_ref m_Paige;
...
}

(.CPP)
int MyView::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    pgm_globals_ptr memory_globals = ((MyWinApp*)AfxGetApp())->m_MemoryGlobals;
    int return_value = 0;
    CRect client_rect;
    rectangle client_paige_rect;
    if(CView::OnCreate(lpCreateStruct == -1)
        return -1;
    ASSERT(m hWnd);
    ASSERT(isWindow(m_hWind));

    // Non-OpenPaige initialisation here!

GetClientRect(&client_rect);
RectToRectangle(&client_rect, &client_paige_rect);

shape_ref window = pgRectToShape(AfxGetMemoryGlobals(), &rect);

PG_TRY(AfxGetApp()->m_MemoryGlobals // See Chapter 19 of the OpenPaige manual.
{
    m_Paige = pgNew(AfxGetApp()->m_Globals,(generic_var)(LPVOID)m_hWnd, window,
window, MEM_NULL, 0);
};

PG_CATCH
{
    return_value = -1;
};
```

```
    PG_ENDTRY;

    pgDisposeShape(window);

    return return_value;
}
```

All Platforms

If def_device is -1 then no device is assumed (which implies you will not be drawing anything and/or will specify a drawing port later). If you need to pass -1 for the def_device parameter, you can use the following predefined macro:

```
#define USE_NO_DEVICE (generic_var) -1 // pgnew is with no device
```

If def_device is neither -1 nor a null pointer it is assumed to be an OpenPaige drawing port to be used for the default (see graf_device, pgSetDefaultDevice).

For "Up & Running", pass a null pointer for def_device (for Macintosh and PowerPC) or the HWND associated with the current message for Windows-PC.

Parameters vis_area, page_area and exclude_area define the literal shapes for which text will display, wrap and jump over, respectively. Each of these define how the text will appear within the OpenPaige object as follows:

vis_area defines the visible area that shows text, or the "hole" in which it displays. This area may be physically smaller than the document containing the text; any physical area of the screen that is outside the boundary of vis_area will *clip* (mask) the text from view.

page_area defines the container in which text will wrap and flow. It is referred to as the page area since it literally defines the page size of your document. The width of page_area also defines the boundaries for which text must wrap. The page_area can be any size, larger or smaller than vis_area.

exclude_area is an optional shape which defines an area or areas in which text must avoid. In other words, if a line of text were to intersect any part of the exclude_area, it must jump over that area in some way to avoid it.

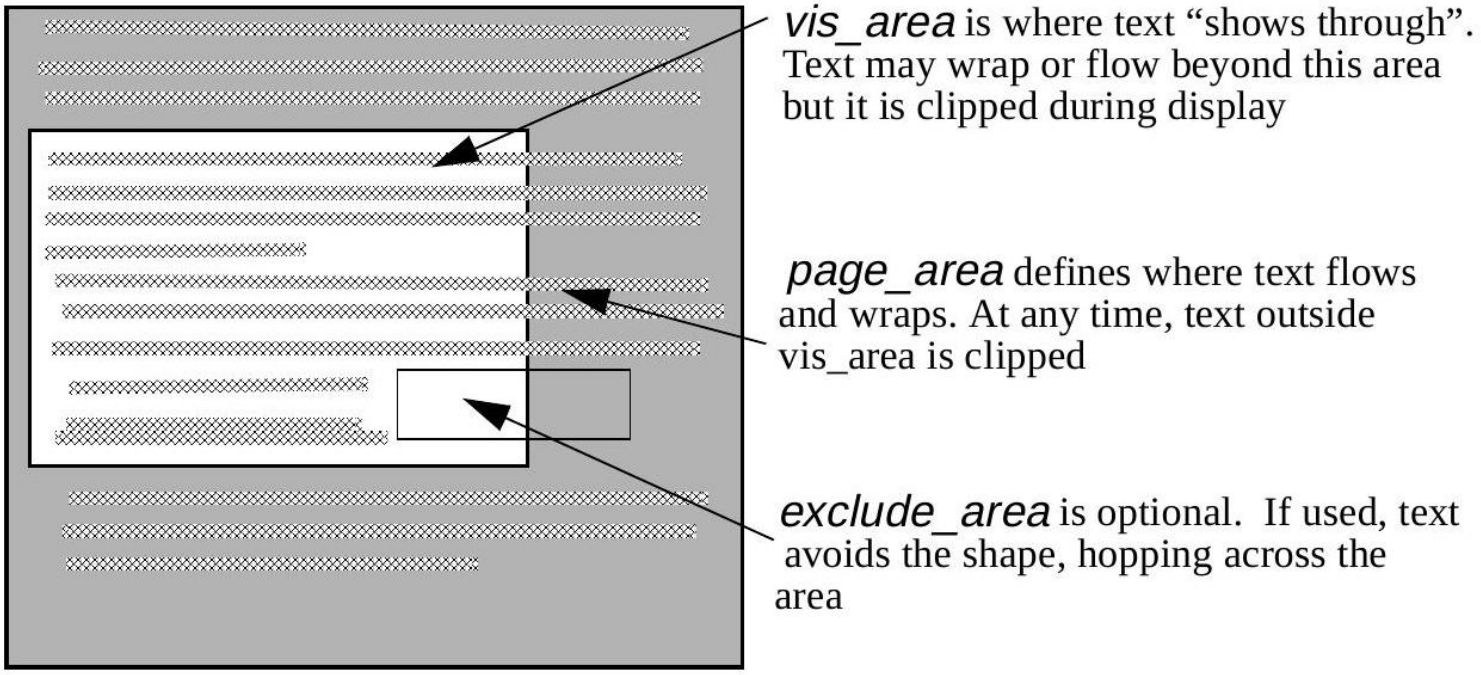
For pgNew, you can pass MEM_NULL for exclude_area, but you must pass a valid shape_ref for

vis_area and page_area.

See "Up & Running Shapes" on how to create a shape_ref.

attributes can contain different bit settings which define specific characteristics for the OpenPaige object. For the purpose of getting "Up & Running" quickly, pass 0 for this parameter (or see “Changing Attributes” on page 3-1).

The initial font and text format used by the pg_ref returned from pgNew will be taken from pg_globals. To change what font, style or paragraph format that a new pg_ref assumes, set the appropriate information in pg_globals after calling pgNew.



MEM_NULL Definition

The value MEM_NULL is a defined value in OpenPaige header files that you should use to imply a "null" shape_ref or memory_ref—see "The Allocation Mgr" on page 25-1.

Error checking pgNew

OpenPaige provides excellent error checking for pgNew. See "Exception Handling" on page 26-1.

2.7 Up & Running Shapes

To avoid a lengthy discussion at this time regarding OpenPaige shapes, we will assume at this time you wish to display text within a simple rectangle (as opposed to some other non-rectangular shape or multiple "container" rectangles).

Creating a shape using rectangle

The easiest way to create a new shape is to use the following function:

```
(shape_ref) pgRectToShape (pgm_globals_ptr mem_globals, rectangle_ptr rect);
```

This returns a new `shape_ref` (which can be passed to one of the `area` parameters in `pgNew`). The globals parameter must be a pointer to the same structure given in `pgInit` and `pgNew`.

The *rect* parameter is a pointer to a structure consisting of a top-left and bottom-right coördinate that encloses a rectangle. The coördinate and rectangle definitions are as follows:

```
typedef struct
{
    long    v;  // vertical position
    long    h;  // horizontal position
}
co_ordinate;

typedef struct
{
    co_ordinate top_left;   // Top-left of rect
    co_ordinate bot_right;  // Bottom-right of rect
}
rectangle, *rectangle_ptr;
```

Hence, if you set a rectangle to the desired dimensions and pass a pointer to that rectangle in `pgRectToShape`, a new memory reference is returned which contains a shape of that rectangle.

NOTE

The reason pgNew requires a shape_ref instead of rectangles is that an OpenPaige object can have non-rectangular shapes for any of its three areas.

For further information regarding shapes, particularly non-rectangular shapes, see "All About Shapes" on page 12-1.

Disposing a Shape

The pgNew function makes a copy of the shape you pass to its parameters. Once you have received a new pg_ref you can dispose the shape. To do so, call:

```
void pgDisposeShape (shape_ref the_shape);
```

Rect to Rectangle

Two utilities exist that make it easier to create OpenPaige rectangles:

```
\#include "pgTraps.h"
(void) RectToRectangle (Rect PG_FAR *r, rectangle_ptr pg_rect);
(void) RectantleToRect (rectangle_ptr pg_rect, co_ordinate_ptr offset, Rect
PG_FAR *r);
```

RectToRectangle converts Rect r to rectangle pg_rect. The pg_rect parameter must be a pointer to a rectangle variable you have declared in your code.

RectangleToRect converts pg_rect to r. Also, if offset is non-null, the resulting Rect is offset by the amounts of the co_ordinate (for example, if offset.h and offset.v were 10, -5 the resulting Mac Rect would be the values in pg_rect with left and right amounts offset by 10 and top and bottom amounts offset by -5.

NOTE (Windows)

Type Rect is identical to type RECT, and both can be used interchangeably.

NOTE (Macintosh)

Since a Mac Rect has a ±32K limit for all four sides, OpenPaige rectangle sides larger than 32K will be intentionally truncated to about 30K.

About Windows, Graphic Ports and Origins

Although OpenPaige is designed to be platform-independent, is does assume a target graphics device that all drawing is transferred to.

When a `pg_ref` is created, the default target device is set to whatever is appropriate for the running platform. For Macintosh, the default device is the current `GrafPort` set when `pgNew` is called.

NOTE (Word Solution Engine for Macintosh)

Unlike WSE, OpenPaige "remembers" what port it should draw to and all subsequent drawing will occur in that port unless you specifically override it.

For the purpose of getting "Up & Running", just make sure you create your window first and have it set as the current port before calling `pgNew`. In subsequent sections, we will provide different ways to change the target port.

Origins

OpenPaige does not care what a window's origin is set to (top-left `co_ordinate` values). OpenPaige only cares about the `area` parameters you provide for `pgNew`; remember, OpenPaige doesn't really know what a window is and doesn't know anything about origins. OpenPaige simply and only follows the coördinates you have set for `vis_area`, `page_area` and `exclude_area`. If your `page_area` shape passed to `pgNew`, for instance, had a top-left of -10000,-9999, the first character of the first line will be drawn at that coördinate location regardless of where the top-left of your window might physically exist. In other words, OpenPaige coördinates are always relative to the associated window's coördinates.

2.8 Attribute Settings

As mentioned earlier, pgNew will accept certain characteristics defined in the "attributes" parameter. The current version supports the following:

```
#define NO_WRAP_BIT          0x00000001  // Wraps only on <CR> or <LF>
#define NO_LF_BIT            0x00000002  // Do not add font
#define NO_DEFAULT_LEADING   0x00000004  // Do not add font leading
#define NO_EDIT_BIT          0x00000008  // No editing (display only)
#define EXTERNAL_SCROLL_BIT  0x00000010  // App controls scrolling
#define COUNT_LINES_BIT      0x00000020  // Keep track of line/para count
#define NO_HIDDEN_TEXT_BIT   0x00000040  // Do not display hidden text
#define SHOW_INVIS_CHAR_BIT  0x00000080  // Show control characters
#define SMART_QUOTES_BIT     0x00000800  // Do "smart quotes"
#define NO_SMART_CUT_BIT     0x00001000  // Do not do "rt cut/paste"
#define NO SOFT HYPHEN BIT   0x00002000  // Ignore soft hyphens
#define NO_DUAL_CARET_BIT    0x00004000  // Do not show dual carets
#define SCALE VIS BIT        0x00008000  // Scale vis area when scaling
#define BITMAP_ERASE_BIT     0x00010000  // Erase page(s) with bitmap drawing
#define TABS ARE WIDTHS BIT  0x10000000  // Fixed-width tab characters
#define LINE_EDITOR_BIT      0x40000000  // Document is line editor mode
```

NO_WRAP_BIT turns off word wrapping (which means a line of text will continue horizontally until a carriage-return or line-feed character is encountered).

NO_LF_BIT causes OpenPaige to ignore line-feed characters. The usual purpose of this setting is for imported text that contains both CR and LF at the end of every line; setting the NO_LF_BIT attribute will cause LF characters to be invisible and have no effect of any kind.

NO_DEFAULT_LEADING prevents any extra leading reported by the system for font attributes. In Windows, *extra leading* is the external leading value reported by GetTextMetrics; in Macintosh, it is the leading value reported by GetFontInfo. By default, OpenPaige adds the extra leading to every line unless this attribute is set.

NO_EDIT_BIT disables editing. In effect, if NO_EDIT_BIT is set, the "caret" will not blink and the user can't insert characters.

EXTERNAL_SCROLL_BIT tells OpenPaige that your application will control all scrolling. (This fairly complex subject is discussed elsewhere.)

COUNT_LINES_BIT tells OpenPaige to keep track of line and paragraph numbers, in which case you can use the line and paragraph numbering features in OpenPaige (see "Line and Paragraph Numbering"). Please note that constantly counting lines and paragraphs, particularly if the document is large and contains wordwrapping with style changes, can consume considerable

processing time. Hence, COUNT_LINES_BIT has been provided to enable/disable this feature.

NO_HIDDEN_TEXT_BIT suppresses the display of all text that is "hidden" (OpenPaige will accept a hidden text attribute as a style). If this bit is not set, hidden text is displayed with a grey strike-through line; if it is set, the text is completely invisible and ignored for line width computations.

SHOW_INVIS_CHAR_BIT causes all invisible characters (control codes such as CR and LF) to be displayed using special character symbols. These symbols are defined in pg_globals (see "Changing Globals").

EX_DIMENSION_BIT tells OpenPaige to include the exclusion area as part of the "document height".

NO_WINDOW_VIS_BIT – Do not respect window's clipped area.

SMART_QUOTES_BIT – Do "smart quotes" (curly quotation marks).

NO_SMART_CUT_BIT – Do not do "smart cut/paste"

NO_SOFT_HYPHEN_BIT – Ignore soft hyphens

NO_DUAL_CARET_BIT – Do not show dual carets

SCALE_VIS_BIT tells OpenPaige to scale the vis_area along with the text when scaling has been enabled. By default, the vis_area is left alone when an OpenPaige document is scaled, leaving the text "behind" the visual boundaries reduced or enlarged. In certain cases—particularly when employing multiple pg_refs into the same document as "edit boxes"—you need this attribute set; for single pg_ref documents that fill all or most of the window, you generally do not want this attribute set.

BITMAP_ERASE_BIT tells OpenPaige to erase area(s) on the page using offsetting bitmap drawing, otherwise the same portions of the screen are erased directly. The purpose of this attribute is to draw "background" graphics in the window when/if OpenPaige needs to erase the screen.

TABS_ARE_WIDTHS_BIT causes all characters to display as no more or less than "wide" blanks. For example, if this attribute is not set, a character aligns the character(s) that follow to the next logical tab stop; if this attribute is set, the a tab character is simply a fixed-width space (the default tab spacing per OpenPaige globals).

LINE_EDITOR_BIT tells OpenPaige that you intend to maintain the document as a "line editor", defined as one where words will not wrap and all lines remain the same height. If OpenPaige knows this in advance, it can bypass the usual "pagination" functions and you can achieve substantially increased performance for line editors.

NOTE

If you set LINE_EDITOR_BIT you must not set any attributes to wrap the text, nor should you vary the point size(s) or attempt any irregular page shapes or page breaks. You can still produce multi-styled text as long as the text height(s) are consistently the same.

Any (or all) of the above settings can exist at once.

NOTE

You can always change these attributes after an OpenPaige object is created (see section 3.1, "Changing Attributes").

Example – pgNew

```
/* This creates a new OpenPaige object */
#include <Paige.h>
#include "pgTraps.h"
extern pg_globals paige_rsrv;

// Routine: Open_Window
// Purpose: Open our window
/* Note: the window has already been made and will be shown and selected
immediately after this function */

void Open_Window(WindowPtr win_ptr)
{
    if (win_ptr=≠nil) /* See if opened OK */
    {
    pg_ref result;
    shape_ref vis, wrap;
    rectangle rect;

    /* this sets vis_area and wrap_area to the shape of the window itself */

    RectToRectangle(win_ptr->portRect, &rect);
    vis = pgRectToShape(&paige_rsrv, &rect);
    wrap = pgRectToShape(&paige_rsrv, &rect);
    result = pgNew(&paige_rsrv, NULL, vis, wrap, NULL, EX_DIMENSION_BIT);
```

```
    }       /* End of IF */
}
```

2.9 Disposing an OpenPaige Object

Once you are completely through with a `pg_ref` (e.g., user closes the window), dispose it with:

```
(void) pgDispose (pg_ref pg);
```

This function disposes all data structures within `pg`; the `pg_ref` will no longer be valid.

Be certain you have not shut down the OpenPaige library before disposing a `pg_ref`, or you will crash.

NOTE (Microsoft Foundation Classes)

The best place to destroy the OpenPaige object is in the OnDestroy() member of your `CView` derived class. Example:

```
(.CPP)

void PGView::OnDestroy()
{
    pgDispose(m_Paige);
    CView::OnDestroy();
}
```

2.10 Getting the "Globals" Pointer

If you need to obtain the pointer to `pg_globals` (originally given to `pgInit` and to `pgNew`), you can get it from a `pg_ref` using the following:

```
(pg_globals_ptr) pgGetGlobals (pg_ref pg);
```

The typical use for pgGetGlobals is to obtain a pointer to pgGlobals in places where the original global structure, given to pg_init, is not easily accessible.

FUNCTION RESULT: This function returns the globals pointer as saved in pg.

To change globals, see section 3.8, "Changing Globals".

2.11 Displaying

To draw the text in a pg_ref to a window, use the following function:

```
(void) pgDisplay (pg_ref pg, graf_device_ptr target_device, shape_ref
vis target, shape_ref wrap_target, co_ordinate_ptr offset_extra, short
draw_mode);
```

The pg_ref's contents are drawn to the target_device. If, however, you pass a null pointer to target_device the text will be drawn to the default device set during pgNew. (For the purposes of getting "Up & Running", we will assume you want to draw to the default device, which will typically be a window that was created prior to pgNew, so pass a null pointer).

vis_target and wrap_target parameters are optional shapes which will temporarily redefine the OpenPaige object's vis_area and wrap_area, respectively. Using these two parameters, you can temporarily control and/or change the way an OpenPaige object will display. Text gets clipped to vis_target, or, if vis_target is a null pointer, to the original vis_area, and text will wrap within wrap_target, or, if wrap_target is MEM_NULL, within the original wrap_area. (For the purposes of getting "Up & Running", pass MEM_NULL for these two parameters.)

If offset_extra is non-null, all drawing is offset by the amounts in that coördinate (all text is offset horizontally by offset_extra → h and vertically by offset_extra → v. If offset_extra is a null pointer, no extra offset is added to the text.

The draw_mode parameter defines the way text should be transferred to the target device. The draw_mode selections are shown below.

See "Display Proc" about how to add ornaments to the text display.

NOTE

You do not need to specify any drawing device for pgDisplay if you intend to display in the window given to pgNew. In this case, just pass NULL to the target_device parameter.

If for some reason you need to redirect the display to some other window or device (such as a bitmap), you can create a graf_device record for that purpose and pass a pointer to that structure for the target_device.

Creating a graf_device for this purpose is the same as the graf_device record used for pgPrintToPage. See "Printing in Windows".

Draw Modes

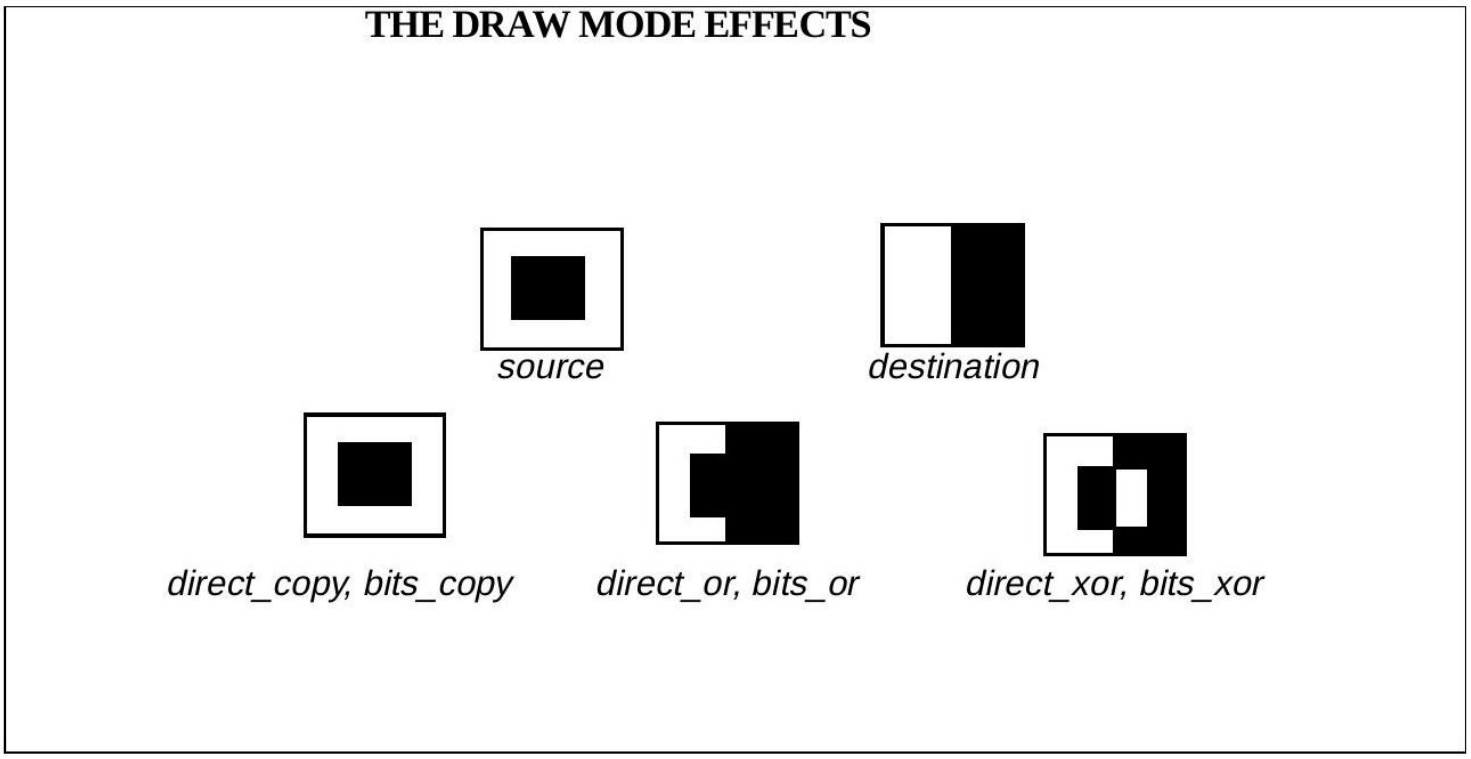
```
typedef enum
{
    draw_none,          // Do not draw at all
    best_way,           // Use most efficient method(s)
    direct_copy,        // Directly to screen, overwrite
    direct_or,          // Directly to screen, "OR"
    direct_xor,         // Directly to screen, "XOR"
    bits_copy,          // Copy offscreen
    bits_or,            // Copy offscreen in "OR" mode
    bits_xor,           // Copy offscreen in "XOR" mode
    bits_emulate_copy   // Copy "fake" offscreen
    bits_emulate_or // "Fake" offscreen in "OR" mode
    bits_emulate_xor    // "Fake" offscreen in "XOR" mode
};
```

"Bits-emulate" Mode

The drawing modes bits_emulate_copy, bits_emulate_or, and bits_emulate_xor are identical to bits_copy, bits_or and bits_xor save that no bitmaps are used and the drawing is directly to the screen. Unlike the non-bitmap drawing modes, however, OpenPaige's standard callback hooks are called to allow modification to its "bitmap", which in this case is the direct screen. Bitmap modification is typically used to render background images, patterns, and other forms of graphics.

NOTE

Unless you need to create a special or unusual effect, always pass `direct_or` or `bits_emulate_or` when responding to `WM_PAINT` (Windows) or an update event (Macintosh), and `best_way` for all other functions requiring a `draw_mode`.



Additional draw modes require the developer to use the custom draw hook and draw his own. See "`text_draw_proc`" for information on how to do custom drawing.

A value of `draw_none` will disable all drawing and visual scrolling. In other words, if the OpenPaige document changes in some way, nothing would change on the screen until the application re-displayed the OpenPaige text contents. The "draw nothing" feature is used only for special cases where an application wants to change without drawing anything yet.

Responding to `WM_PAINT` Event (Windows)

```
{
PAINTSTRUCT ps;
BeginPaint(hWnd, &ps);
pgDisplay(pg, NULL, MEM_NULL, MEM_NULL, NULL, direct_or);
}
EndPaint(hWnd, &ps);
```

To display the OpenPaige object in MFC, use `OnPaint()`. Do not try to use `OnDraw()` or it will not draw correctly.

EXAMPLE

```
(.CPP)

void PGView::OnPaint()
{
CWnd::OnPaint();

// If you don't use the OnEraseBkgnd() member of the MFC class,
// you must erase the background of the window first.

pgDisplay(m_Paige, NULL, MEM_NULL, MEM_NULL, NULL, bits_emulate_or);
}
```

2.12 Key Insertion

OpenPaige actually makes very little distinction between keyboard entry and any other text insertion, and in both cases the following function is used:

```
(pg_boolean) pgInsert (pg_ref pg, pg_char_ptr data, long length, long position,
short insert_mode, short modifiers, short draw_mode);
```

This function will insert length bytes pointed to by data. The insertion will occur at byte offset position if it is positive or zero; if position is `CURRENT_POSITION` (a `#define`d constant of -1), the insertion occurs at the current insertion point.

The `insert_mode` parameter defines the type of data being inserted, which can be any of the following:

```
typedef enum
{
    key_insert_mode,        // Typing insertion
    key_buffer_mode,        // Typing-buffer insertion
    data_insert_mode,       // Raw data insertion
```

```
}
```

For keyboard entry, pass `key_insert_mode` or `key_buffer_mode`; for any other data insertion, pass `data_insert_mode`.

The difference between the two "key" insert modes and `data_insert_mode` is that a key insertion can contain special controls such as arrow keys and backspace (delete). For `data_insert_mode`, the bytes will be inserted as is.

If `key_insert_mode` is used, the new character(s) will draw immediately if `draw_mode` is nonzero.

If `key_buffer_mode` is used, character(s) will be buffered (temporarily saved) and drawn later by OpenPaige; the purpose of this mode is to avoid "getting ahead" of keyboard entry on complex document entry. It is also useful for Macintosh double-byte script entry, in which the text is entered all at once from a floating palette window.

NOTE (Windows)

The `key_buffer_mode` is usually meaningless in the Windows environment; instead, you should always use `key_insert_mode` when processing keyboard characters. Using `key_buffer_mode` (where chars are stored and inserted later) requires a call to `pgIdle` which, under the Windows messaging system, would require you to set up a "timer" message that occurs every few milliseconds, which is probably not implemented in most applications.

If keys are buffered, OpenPaige will display the new text during the first `pgIdle` function call (see “Blinking Carets & Mouse Selections”).

NOTE

"Arrows" and other control codes are defined (and changeable) in the `pg_globals` record (see "Changing Globals"); these special controls will be processed correctly for `key_insert_mode` and `key_buffer_mode` only.

The `modifiers` parameter can change the way the `pg_ref` will respond to special control characters for `key_insert_mode` (`modifiers` is ignored for the other insertion modes). In the current version, the following value is supported:

```
#define EXTEND_MOD_BIT 0x0001   // Extend the selection
```

If `modifiers` is `EXTEND_MOD_BIT`, the selection range is extended if an arrow key is "inserted." Other selection modifier bits are explained in "Modifiers".

The `draw_mode` for `pgInsert` performs identically to `pgDisplay` and can be any of the verbs defined for drawing. If you just want to insert but not display, pass `draw_none` for `draw_mode`. If `key_buffer_mode` is used for insertion, the `draw_mode` is saved and used later when the text is displayed.

For keyboard insertions, the recommended `draw_mode` is `best_way`.

CAUTION (Macintosh)

> Mac developers should not confuse these modifier bits with the modifiers given in the event record. There is no similarity. The modifiers shown here are the ones OpenPaige supports.

NOTE

The insertion will assume either the text format of the current insertion point OR the format of the last style/font/format change, whichever is more recent. This is true even if you specify an insert position other than the current point. If you want to force the insertion to be a particular font or style, simply call the appropriate function to change the text format prior to your insertion.

FUNCTION RESULT

The function returns TRUE if the text and/or highlighting in `pg` changed in any way. Note that no change occurs only if `key_buffer_mode` is passed as the insert mode, in which case the characters are stored and not drawn until the next call to `pgIdle`. Another situation that will not change anything visually is passing `draw_none` as the `draw_mode`. In both cases, `pgInsert` would return FALSE. The purpose of this function result is for the application to know whether or not it should update scrollbar values or scroll to the insertion point, etc. (i.e., it is a waste of processing time to check or change scroll positions if nothing changed on the screen).

Running Unicode

If you are using the Unicode-enabled OpenPaige library, the "data" to be inserted is expected to be one or more 16-bit characters. The data size in this case is assumed to be a character count (not a byte count). This is due to the fact that if UNICODE is defined in your preprocessor or header files (which it should be for a true Unicode-enabled application), a `pg_char_ptr` changes from a byte pointer to a 16-bit character pointer.

For example, to insert the Unicode value 0x0041 (letter "A") you would pass the value of 1 in the `length` parameter even though the character size is technically 2 bytes long.

TECH NOTE: Insert Positions

The specified insertion position is a zero-relative byte offset. Note that this is a byte—not a "character" offset (characters in OpenPaige can be more than one byte), rather a byte offset from the beginning of all text in `pg`, starting at zero.

EXCEPTION: The pure Unicode version measures everything as 16 bit characters. Hence, the insertion point in this case is a character position.

If one or more characters are currently selected (selection range ≥ one character), those characters are deleted before the insertion occurs. Note that if the specified insertion position were `CURRENT_POSITION`, the insertion will occur to the immediate left of the previously selected text (which will have been deleted).

After the insertion, the new insertion position in `pg` is advanced to length `bytes` from the original specified position. Example: If 100 bytes were inserted at text position 500 when `pgInsert` returns the current insertion position will be 600.

APPLIED STYLE(S) AND INSERTION

If `pgInsert` occurs at the current insertion point, whatever the last style and/or font that was applied to that insertion point will be applied to the next insertion.

For example, suppose all text in `pg` is currently "Helvetica" font, and `pg` has a single insertion point (not a selected range of characters). Before inserting new text, a call is made to `pgSetFontInfo` with "Times Roman" font; the very next subsequent `pgInsert` would apply Times Roman—not Helvetica—to the new text.

However, if the insertion occurs somewhere other than the current insertion, the font/style that is applied will be whatever font/style applies to that position in text.

Hence, to implement the insertion of specific, multi-stylized text, the logic to perform should be as follows:

```
pgSetStyleInfo(...) - and/or - pgSetFontInfo(...);
pgInsert(..., CURRENT POSITION,...);
pgSetStyleInfo(...) - and/or - pgSetFontInfo(...);
pgInsert(..., CURRENT_POSITION,...); etc.
```

NOTE: For repetitive insertions, the insertion point will automatically advance the number of bytes you insert, so normally you should not need to set a new position if you are doing repetitive, sequential insertions.

WARNING: If you need to apply a specific font or style to a text insertion (such as in the logic above), do not set the insertion point after you set the style/font or that style/font attribute may be lost. If you must set position, do so BEFORE calling pgSetFont/nfo or pgSetStyleInfo.

EXAMPLE

WRONG WAY:

```
pgSetStyleInfo(...);
pgSetSelection(pg, 0, 0); //<-- previous style setting is lost!
pgInsert(...);
```

RIGHT WAY:

```
pgSetSelection(pg, 0, 0);
pgSetStyleInfo(...); // <-- Style gets applied to next insertion
pgInsert(...);
```

TECH NOTE: Nothing happens

> Nothing seems to happen when I insert text.

If you are doing inserts with `key_insert_mode`, OpenPaige won't do anything if the `pg_ref` is deactivated. That might be the problem. If so, you need to use `data_insert_mode`, not

key_insert_mode, and it will then work; pgInsert does nothing.

2.13 Keyboard Editing with MFC (Windows)

To get *Up and Running* with basic keyboard editing you must add the following code to your MFC view class:

```
(.H)
// Declare the following private variables.
short m_KeyModifiers;

(.CPP)

// Respond to the windows message WM_KEYDOWN...
void PGView::OnKeyDown(UINT nChar, UINT nRepCnt, UINT nFlags)
{
pg_globals_globals = ((MyWinApp*)AfxGetApp())->m_Globals;
pg_short_t verb;

    switch(nChar)
    {
        case VK_SHIFT:
            m_KeyModifiers |= EXTEND_MOD_BIT;
            break;
        case VK_CONTROL:
            m_KeyModifiers |= CONTROL_MOD_BIT;
            break;
        case VK_LEFT:
            SendMessage(WM_CHAR, globals->left_arrow_char);
            break;
        case VK_UP:
            SendMessage(WM_CHAR, globals->up_arrow_char);
            break;
        case VK_RIGHT:
            SendMessage(WM_CHAR, globals->right_arrow_char);
            break;
        case VK_DOWN:
            SendMessage(WM_CHAR, globals->down_arrow_char);
            break;
        case VK_HOME:
            verb = begin_line_caret;
            if(m_KeyModifiers & CONTROL_MOD_BIT)
```

```
                verb = home_caret;
            if(m_KeyModifiers & EXTEND_MOD_BIT)
                verb |= EXTEND_CARET_FLAG;
            pgSetCaretPosition(mPaige, verb, TRUE);
            pgScrollToView(m_Paige, CURRENT_POSITION, 0, 0, TRUE,
bits_emulate_or;
            break;
        case VK_END:
            verb = end_line_caret;
            if(m_KeyModifiers & CONTROL_MOD_BIT)
                verb = doc_bottom_caret;
            if(m_KeyModifiers & EXTEND_MOD_BIT)
                verb |= EXTEND_CARET_FLAG;
            pgSetCaretPosition(m_Paige, verb, TRUE);
            pgScrollToView(m_Paige, CURRENT_POSITION, 0, 0, TRUE,
bits_emulate_or;
            break;
        case VK_PRIOR:
            SendMessage(WM_VSCROLL, SB_PAGEUP);
            break;
        case VK_DELETE:
            if(m_KeyModifiers & EXTEND_MOD_BIT)
            {
                long start, end;
                pg_ref scrap;
                pgGetSelection(m_Paige, &start, &end);
                if(start == end)
                    return;
                scrap = pgCut(m_Paige, &start, &end);
                assert(scrap);
                OpenClipboard();
                pgPutScrap(scrap, 0, pg_void_scrap);
                CloseClipboard();
                pgDispose(scrap);
                scrap = MEM_NULL;
                SetChanged();
            }
            else
            {
                SendMessage(WM_CHAR, globals->fwd_delete_char);
            }
        case VK_NEXT:
            SendMessage(WM_VSCROLL, SB_PAGEDOWN);
            break;
            pgScrollToView(m_Paige, CURRENT_POSITION, 0, 0, TRUE,
```

```
bits emulate or);
            break;
    }
}
            break;
        case VK_INSERT:
        {
        if(m_Key_Modifiers & CONTROL_MOD_BIT
            {
                long start, end;
                pg_ref scrap;
                pgGetSelection(m_Paige, &start, &end);
                if(start == end)
                    return;
                scrap = pgCopy(m_Paige, NULL);
                assert(scrap);
                OpenClipboard();
                pgPutScrap(scrap, 0, pg_void_scrap);
                CloseClipboard();
                pgDispose(scrap);
            }
        else if(m_KeyModifiers & EXTEND_MOD_BIT)
            {
                pg_ref scrap = MEM_NULL;
                OpenClipboard();
                scrap = pgGetScrap(globals, 0, HookEmbedProc);
                CloseClipboard();
                if scrap
                {
                    pgPaste(m_Paige, scrap, CURRENT_POSITION, false, best_way);
                    pgDispose(scrap)
                }
            }
        }
        pgScrollToView(m_Paige, CURRENT_POSITION, 0, 0, TRUE, bits_emulate_or);
        break;
    }
}

// Respond to the Windows message WM_KEYUP...
void MyView::OnKeyUp(UINT nChar, UINT nRepCnt, UINT nFlags)
{
    switch(nChar)
    {
        case VK_SHIFT:
```

```
            m_KeyModifiers &= (~EXTEND_MOD_BIT);
            break;
        case VK_CONTROL:
            m_KeyModifiers &= (~CONTROL_MOD_BIT);
            break;
    }
}

// Respond to the Windows message WM_CHAR...
void MyView::OnChar(UINT nChar, UINT nRepCnt, UINT nFlags)
{
    pg_char the_char = (pg_char)nChar;
    pgInsert(mPaige, &the_char, 1, CURRENT_POSITION, key_insert_mode,
m_KeyModifiers, best_way);
    pgScrollToView(m_Paige, CURRENT_POSITION, 0, 0, TRUE, bits_emulate_or);
}
```

2.14 Pending Buffer Insertions

As mentioned in `pgInsert`, if `key_buffer` mode is used, the characters get stored in an internal buffer and get inserted during the next `pgIdle`.

There might be an occassion, however, that requires immediate insertion of anything pending in this buffer. To do so, call the following:

```
(pg_boolean) pgInsertPendingKeys (pg_ref pg);
```

Calling this function will immediately "empty" any pending characters, inserting and displaying them as appropriate. If there aren't any pending characters, `pgInsertPendingKeys` does nothing. The function returns TRUE if one or more characters were inserted.

NOTE

The display mode used when OpenPaige displays the pending buffer will be the original display mode passed to `pgInsert`.

2.15 Blinking Carets & Mouse Selections

Caret blinking (Macintosh only)

To cause the "caret" to blink in a `pg_ref`, call the following as often as possible:

```
(pg_boolean) pgIdle (pg_ref pg);
```

NOTE (Macintosh):

`pgIdle` should be called repeatedly while you are waiting for an event.

The `pg` parameter must be a valid `pg_ref` (can not be a null pointer).

FUNCTION RESULT: The function returns TRUE if character(s) were inserted and displayed that were stored previously from `pgInsert` calls with `key_buffer_mode`. This will only happen if you had called `pgInsert`, passing `key_buffer_mode` as the data transfer parameter. A result of TRUE or FALSE from `pgIdle` can help your application know whether or not it should update scrollbar values (since new text has been inserted). For Windows `key_buffer_mode` is not usually necessary, see “Key Insertion”.

NOTE (Windows)

You do not need to call `pgIdle()` since the blinking caret is maintained by the OS. Calling `pgIdle` by "accident" however is harmless.

Clicking & Dragging

Clicking and dragging is accomplished by using the following function:

```
(long) pgDragSelect (pg_ref pg, co_ordinate_ptr location, short verb, short
modifiers, long track_refcon, short auto_scroll);
```

To change the insertion point in a pg_ref (i.e., in response to a mouse click), call pgDragSelect with the location parameter set to the location of the "click." The coördinate values must be local to the window's coördinate system (relative to the top-left window origin).

For **Macintosh**, location should be the same as the "where" member of the EventRecord, converted to local coördinates.

For **Windows**, location is usually the coördinates given to you in lParam when responding to WM_LBUTTONDOWN, WM_LBUTTONDBLCLK, or WM_MOUSEMOVE.

The verb parameter defines what action should occur, which must be one of the following:

```
enum
{
    mouse_down, // First-time click
    mouse_moved,    // Mouse moved
    mouse_up,       // Mouse button released
}
```

NOTE: pgDragSelect() does not retain control at any time—it always returns control immediately regardless of what verb is passed.

For the first click, pass mouse_down in verb.

In a Macintosh-specific application, while the user is holding down the mouse button, wait for the mouse location to change and, if it does, call pgDragSelect with the new location but with verb as mouse_moved.

In a Windows-specific application, call pgDragSelect(mouse_moved) in response to a WM_MOUSEMOVE if the mouse button is still down.

When the mouse button is released, pass the final location and mouse_up for verb.

NOTE

It is important to call pgDragSelect with mouse_up after the user releases the mouse button *even if the mouse never moved from its original location*. This is because OpenPaige performs certain housekeeping chores when mouse_up is given.

The modifiers parameter controls the way text is selected. For "normal" click/drag, pass zero for

this parameter; for added effects (such as responding to double-clicks, shift-clicks, etc.), see “Modifiers".

If `auto_scroll` is "TRUE", OpenPaige will automatically scroll the document if `pgDragSelect` (with `verb` as `mouse_moved`) has gone beyond the `vis_area`. See "All About Scrolling". For getting "Up & Running", you can pass TRUE for this parameter.

`track_refcon` is used when and if OpenPaige makes a call to the track-control-callback function. If a style is a "control" (the control bit set for the style class bits field), OpenPaige calls the tracking control function hook and passes the `track_refcon` to the app. In other words, this value is application-defined and OpenPaige does nothing with it. For getting "Up & Running", you can pass 0 for this parameter.

FUNCTION RESULT

For "normal" mouse tracking, ignore the function result of `pgDragSelect`. The only time the function result is significant is when you have customized a style to be a "control" (information is available on "control" styles under "Customizing OpenPaige”). If you have not customized OpenPaige in any way, `pgDragSelect` will always return zero.

Modifiers

The following bit settings are supported for the modifiers parameter in this release:

```
#define EXTEND_MOD_BIT      0x0001  // Extend the selection
#define WORD_MOD_BIT        0x0002  // Select whole words only
#define PAR_MOD_BIT         0x0004  // Select whole paragraphs only
#define LINE_MOD_BIT        0x0008  // Highlight whole lines
#define VERTICAL_MOD_BIT    0x0010  // Allow vertical selection
#define DIS_MOD_BIT         0x0020  // Enable discontinuous selection
#define STYLE_MOD_BIT       0x0040  // Select whole style range
#define WORD_CTL_MOD_BIT    0x0080  // Select "words" delimited by ctrl chars
#define NO_HALF_CHARS_BIT   0x0100  // Do not go left/right on half chars
#define CONTROL_MOD_BIT     0x0200  // Word advance for arrows
```

Various combinations of these bits can generally be set to create the desired effect such as word selections, paragraphs selections, etc., save that vertical selection does not work with the other

modifiers. If misused regardless, it will produce unpredictable results.

The following is a description of how text is highlighted in response to each of these bits:

`EXTEND_MOD_BIT` will extend the selection for `verb` of `mouse_down` (otherwise the previous selection is removed). For Macintosh, this is the same as "shift-click" (but you need to determine that from your application and set this bit).

`WORD_MOD_BIT` will select whole words, otherwise only single characters are selected.

`PAR_MOD_BIT` will select whole paragraphs.

This is different than `LINE_MOD_BIT` (below) since a paragraph could contain several lines if word wrapping exists.

`LINE_MOD_BIT` will select whole lines. This differs from `PAR_MOD_BIT` since a paragraph might consist of many lines.

`VERTICAL_MOD_BIT` allows vertical selection. This bit really causes a rectangular region that selects all characters intersecting that region and will not follow any particular character. `VERTICAL_MOD_BIT` is mainly useful for tables and tabular columns.

`DIS_MOD_BIT` allows discontinuous selections. If this bit is set, the previous selection remains and a new selection range is started (OpenPaige can have multiple selection ranges).

`STYLE_MOD_BIT` causes whole style ranges to become selected. This is similar to word/paragraph/line highlighting except style changes are considered the delimiters (which also means the whole document could be selected in one click if only one style exists).

`WORD_CTL_MOD_BIT` causes text between control characters to be selected. This is similar to word/paragraph/line highlighting except control codes are considered the delimiters.

> **NOTE:** In OpenPaige "control codes" or "control characters" are not necessarily limited to standard ASCII symbols. Control characters in the OpenPaige context are defined in `pg_globals` (see "Changing Globals").

`NO_HALF_CHARS_BIT` controls whether or not dragging can change the selection point half way into a character. Normally, if this bit is not set, once the mouse moves half way into a character, that character is considered to be "selected" (or unselected if moving in the opposite direction). Setting this bit, however, instructs `pgDragSelect` not to select the character until it has completely crossed over its area.

`CONTROL_MOD_BIT` is used mainly with arrow keys. This causes the selection to advance to the next word (right arrow) or to the previous word (left arrow).

For additional information about highlighting and selection range(s), see "All About Selection".

2.16 Click & Drag using Microsoft Foundation Classes (Windows)

To get Up and Running with simple mouse drag select in MFC, use the following code as a starting point:

```
(.H)
/* Declare the following private variables. Make sure to set m_Dragging to
FALSE in the construct to avoid the unintialized variable bug!! *

short m_MouseModifiers;
BOOL m_Dragging;

(.CPP)
// Respond to the Windows message WM_LBUTTONDOWN...
void MyView::OnLButtonDown(UINT nFlags, CPoint point)
{
    CView::OnLButtonDown(nFlags, point);
    co_ordinate mouse { point.y, point.x };
    SetCapture();
    m_Dragging $=$ TRUE;
    if(nFlags & MK_SHIFT)
        m_MouseModifiers =+ EXTEND_MOD_BIT;
    if(nFlags & MK_CONTROL)
        m_MouseModifiers =+ PAR_MOD_BIT;
        pgDragSelect(m_Paige, &mouse, mouse_down, m_MouseModifiers, 0, TRUE);
}

// Respond to the Windows WM_MOUSEMOVE message...
void MyView::OnMouseMove(UINT nFlags, CPoint point)
{
    CView::OnMouseMove(nFlags, point);
    co ordinate pg_mouse = {point.y, point.x};
    if(m_Dragging)
    {
        pgDragSelect(m_Paige, &mouse, mouse_up, m_MouseModifiers, 0, FALSE);
        m MouseModifiers = 0;
        ReleaseCapture();
        m_Dragging = FALSE;
```

```
    }
}
```

Responding to Windows mouse events

```
case WM_LBUTTONDBLCLK:
pg_modifiers = pg_modifiers = WORD_MOD_BIT;

// fall through to WM_LBUTTONDOWN

case WM_LBUTTONDOWN:
if(pgRef)
{
        co_ordinate pg_mouse;
        mouse_contact = TRUE;
        SetCapture(hWnd);
        pg_mouse.h = lParam & OxFFFF;
        pg_mouse.v = ((lParam & OxFFFF0000) >> 16);

    if (wParam & MK_SHIFT)
    pg_modifiers =+ EXTEND_MOD_BIT;

    if (wParam & MK CONTROL)
    pg_modifiers =+ DIS_MOD_BIT;
    pgDragSelect(pgRef, &pg_mouse, mouse_down, pg_modifiers, 0, TRUE);
}
return 0 ;

case WM_LBUTTONUP:
    if(pgRef)
    {
        co_ordinate pg_mouse;
        pg_mouse.h = lParam & OxFFFF;
        pg_mouse.v = ((lParam & OxFFFF0000) >> 16);
        mouse_contact = FALSE;
        pgDragSelect(pgRef, &pg_mouse, mouse_up, pg_modifiers, 0, FALSE);
        pg_modifiers = 0;
        ReleaseCapture();
    }
return 0;

case WM_MOUSEMOVE:
```

```
    if(mouse contact)
        pgDragSelect(pgRef, &pg_mouse, mouse_moved, pg_modifiers, 0, TRUE);
    else
    {
        pg_view = pgPtInView(pgRef, &pg_mouse, NULL);

        if (pg_view & WITHIN_TEXT)
            SetCursor(LoadCursor(NULL, IDC_IBEAM));
        else
            SetCursor(LoadCursor(MULL, IDC_ARROW));
    }
}
return 0;
```

TECH NOTE

Turn automatic scroll off

To prevent selecting/scrolling you would simply pass FALSE for `pgDragSelect` so it doesn't try to auto-scroll. As far as not letting the user select text outside the visual area, I would simply check to see if the coördinate that will get passed to `pgDragSelect` is outside of the view area and if it is, just force it to some other point that is within the view area.

In fact, you wouldn't even need to turn off auto-scroll if you forced the coördinate to always be within the visual area. Remember, you have complete control over `pgDragSelect` (control always comes back to you unlike, say, `TrackControl` on Macintosh) so there is no reason you can't adjust the "mouse" point for each pass.

(Mac-specific) Problems with mouse clicks -1

> I have big troubles handling mouse clicks in the openPaige object within my class library. If I get a click (with `GetMouse(&hitPt)`) and do the following (testing a response to a simple click)...

Your test code sample should work. Therefore, I have to conclude there is something wrong with the mouse point you obtain with `GetMouse()`.

I would guess that you are doing a `GetMouse()` without regards to the current `GrafPort`. Since

GetMouse() returns a LOCAL point (based on current port's coördinates), if you don't have the correct GrafPort set you will get some other coördinate system. Worst case, you are getting "global" coördinates which will be completely different than what you expect.

Or, another possibility might have to do with the window's "origin". I know that some class libraries muck with this to create scrolling effects.

What you need to do is to check what the actual values of point.h and point.v really are. I know that pgDragSelect works; in fact, you should see the caret immediately appear at the point you give for mouse_down verb.

BTW, the usual (best) way for dragging the mouse in a pg_ref is to get the click right out of the EventRecord.where field (first doing a GlobalToLocal on it). That is by far the most accurate -- but I do not know if that EventRecord is easily available in your class library.

2.17 Activate/Deactivate

To deactivate a pg_ref (to cause highlighting or the "caret" to disappear), call the following function:

```
(void) pgSetHiliteStates (pg_ref pg, short front_back_state, short perm_state,
pg_boolean show_hilite);
```

In a "window" environment, where different windows can overlap, it is usually desirable to disable any OpenPaige objects that are not contained in the front most window. To do so, pgSetHiliteState can be called to turn off the highlighting or the "caret."

An OpenPaige object, however, contains two highlight states, one for "front/back" activate and deactivate and one to disable a pg_ref in both states. For "normal" applications, you will only be changing the front/back highlight state (activate or deactivate a pg_ref according to its window position). The purpose of the alternate highlight state is to provide a way to disable a pg_ref completely regardless of its window position.

The front_back_state should be one of the following values:

```
typedef enum
{
    no_change_verb, // State stays the same
```

```
    activate verb,  // Set to activate mode
    deactivate_verb,    // Set to deactivate mode
}
```

The perm_state parameter provides an alternate highlight state setting; this parameter can also be any of the above. For getting "Up & Running," however, pass no_change_verb for this parameter.

If show_hilite is "TRUE", the highlighting (or caret) will redraw according to pg's new state. A "FALSE" value will activate or deactivate pg internally (by setting special flags within the pg_ref) but the highlighting or caret will remain unchanged. For getting "Up & Running", always pass TRUE for should_draw.

See also "Additional Selection Support" and "Activate/Deactivate with shape of selection still showing".

Responding to WM_SETFOCUS and WM_KILLFOCUS messages

```
{
case WM_KILLFOCUS:
    pgSetHiliteStates(pgRef, deactivate_verb, no_change_verb, TRUE);

case WM SETFOCUS:
    pgSetHiliteStates(pgRef, activate_verb, no_change_verb, TRUE);
}
```

Getting the Highlight State

If you want to know what state a pg_ref is in, call the following:

```
(void) pgGetHiliteStates (pg_ref pg, short PG_FAR *front_back_state, short
PG_FAR *perm_state);
```

The front/back highlight state will be returned in front_back_state and the alternate state in perm_state. Both parameters will be set to either activate_verb or deactivate_verb.

NOTES

1. If the highlight status is already set to what is specified in your parameters (e.g., if you are deactivating a `pg_ref` that is already deactivated or vice versa), this function does nothing.
2. A `pg_ref` returned from `pgNew` is set to an active state.
3. If a `pg_ref` is in a deactivate state, the caret will not blink even if you call pgIdle and highlighting will not draw.

TECH NOTE

Why two activate states?

One is for regular activate/deactivate for a window; the other is to FORCE deactivation regardless of the window's front/behind state. Haven't you ever been in a situation where you want to deactivate selections but the window is still in front? Using two possible states, it becomes easier to do that. The two states are logically "AND'd" logic for activation: both must be true or the document is deactivated.

MFC NOTE

IMPORTANT: You must activate and deactivate the OpenPaige object in the MFC OnSetFocus() and OnKillFocus() before any of the functions in this chapter will work.

Example:

```
(.CPP)

// Respond to Windows message WM SETFOCUS...
void MyView::OnSetFocus(CWnd* pOldWnd)
{
    CView::OnSetFocus(pOldWnd);
    pgSetHiliteStates(m_Paige, activate_verb, no_change_verb, TRUE);
}

// Respond to Windows message WM_KILLFOCUS...
void MyView::OnKillFocus(CWnd* pNewWnd)
```

```
{
    pgSetHiliteStates(m_Paige, deactivate_verb, no_change_verb, TRUE);
    CView::OnKillFocus(pNewWnd);
}
```

3 BEYOND THE DEFAULTS

The purpose of this section is to explain some of the more common additions and/or changes to the "bare minimum" implementation discussed in the previous section, "Up & Running."

3.1 Changing Attributes

There will be situations where you want to change the attributes of an OpenPaige object after it is created (these are the bits initially passed to `pgNew` for the "attributes" parameter). There are also situations where you want to examine the current attributes (to check mark a menu item, for instance). To do so, use the following:

```
(long) pgGetAttributes (pg_ref pg);
(pg_boolean) pgSetAttributes (pg_ref pg, long attributes);
```

To obtain the current attribute bits, call `pgGetAttributes`.

FUNCTION RESULT: The function result will be the current setting(s) of pg.

To change the attributes, call `pgSetAttributes` with attributes set to the new bit value(s).

OpenPaige "attributes" are defined as bit settings which can be a combination of any bit values shown below:

```
#define NO_WRAP_BIT         0x00000001  // Wraps only on CR or LF
#define NO_LF_BIT           0x00000002  // <LF> char ignored
#define NO_DEFAULT_LEADING  0x00000004  // Do not add font leading
#define NO_EDIT_BIT         0x00000008  // No editing (display only)
#define EXTERNAL_SCROLL_BIT 0x00000010  // App controls scrolling
#define COUNT_LINES_BIT     0x00000020  // Track line/para count
```

```
#define NO_HIDDEN_TEXT_BIT  0x00000040  // Do not display hidden text
#define SHOW_INVIS_CHAR_BIT 0x00000080  // Show invisible character(s)
#define EX_DIMENSION_BIT    0x00000100  // Exclude width/height
#define NO_WINDOW_VIS_BIT   0x00000200  // Do not respect clipped area
#define SMART_QUOTES_BIT    0x00000800  // Do "smart" quotes
#define NO_SMART_CUT_BIT    0x00001000  // Do not do "smart" cut/paste
#define NO_SOFT_HYPHEN_BIT  0x00002000  // Ignore soft hyphens
#define NO_DUAL_CARET_BIT   0x00004000  // Do not show dual carets
#define SCALE_VIS_BIT       0x00008000  // Scale vis_area when scaling
#define BITMAP_ERASE_BIT    0x00010000  // Erase page(s) with bitmap drwg
#define TABS_ARE_WIDTHS_BIT 0x10000000  // Tab chars are merely wides
#define LINE_EDITOR_BIT     0x40000000  // Doc is line editor mode
```

These are described under "Attribute Settings".

FUNCTION RESULT: After calling `pgSetAttributes`, the function result will be "TRUE" if `pg` should be redrawn. The only time "TRUE" is returned is when one or more attributes have been set that will affect the way text is drawn or the way word wrap is computed.

WARNING: Before setting attributes, first get the current settings from the function `pgGetAttributes` and change the bits you require and pass that whole long value to `pgSetAttributes`. Otherwise, the `view only` bits will get changed erroneously.

Additional attributes can be set for more advanced features using the following `set` and `get` functions:

```
(pg_boolean) pgSetAttributes2 (pg_ref pg, long attributes); (long)
pgGetAttributes2 (pg_ref pg);
```

To obtain the current, extended attribute bits, call `pgGetAttributes2`.

FUNCTION RESULT: The function result will be the current setting(s) of the extended attributes of `pg`.

To change the extended attributes, call `pgSetAttributes2` with attributes set to the new bit value(s).

OpenPaige "extended attributes" are defined as bit settings which can be a combination of any of the following.

```
#define KEEP_READ_STYLES    0x00000200  // Keep existing style_infos for
pgReadDoc()
```

```
#define KEEP_READ_PARS      0x00000400  // Keep existing par_infos for
pgReadDoc()
#define KEEP_READ_FONTS     0x00000800  // Keep existing font_infos for
pgReadDoc()
#define CHECK_PAGE_OVERFLOW 0x00002000  // Constantly check page overflow
#define NO_HAUTOSCROLL      0x00080000  // Do not autoscroll horizontally
#define NO_VAUTOSCROLL      0x00100000  // Do not autoscroll vertically
```

KEEP_READ_STYLES tells OpenPaige to not remove existing style_info records from the pg_ref when a file is read. Normally, all existing style records are replaced with the styles read from an OpenPaige file. This attribute is used to retain the existing styles.

KEEP_READ_PARS tells OpenPaige to not remove existing par_info records from the pg_ref when a file is read. Normally, all existing paragraph records are replaced with the paragraph records read from an OpenPaige file. This attribute is used to retain the existing paragraph records.

KEEP_READ_FONTS tells OpenPaige to not remove existing font_info records from the pg_ref when a file is read. Normally, all existing font records are replaced with the fonts read from an OpenPaige file. This attribute is used to retain the existing fonts.

CHECK_PAGE_OVERFLOW tells OpenPaige to constantly test the position of the last character in the document and, if it overflows the bottom of the page_area, sets an internal field to the number of characters that have overflowed. The purpose of this attribute is to allow an application to implement features that require "page overflow checking", but since this requirement requires constant pagination and extra processing, set this attribute only when absolutely necessary.

NO_HAUTOSCROLL, NO_VAUTOSCROLL tells OpenPaige not to automatically scroll horizontally or vertically, respectively, when pgDragSelect() is called.

"Auto-checking" page overflow

Setting CHECK_PAGE_OVERFLOW with pgSetAttribute2() causes OpenPaige to continuously check the situation where character(s) flow below the boundaries of the page area. If this attribute is set, the overflow_size member within the pg_ref get set to the number of characters that overflow the page.

Or, if overflow_size is set to -1, a single carriage return is causing the overflow (i.e., the text overflows but the overflow is a "blank" line).

NOTE: The auto-checking for page overflow is meaningless if your pg_ref is set for repeating

pages, or if your pg_ref is set to a variable page size. The only time overflow checking will work (or make any sense) is for fixed-size, nonrepeating page shapes.

Checking page overflow

NOTE: You should not implement this code if your pg_ref is set for repeating pages, or if your pg_ref is set for a variable document height.

```
/* Call the function below after doing anything that can change the size of the
document. This included insertions, deletions, style and font changes (which
can cause new word wrapping) and page size changes. This function returns the
number of characters that are overflowing the page area of pg. */

/* Note: CHECK_PAGE_OVERFLOW must be set with pgSetAttributes2(pg). */

long CheckPageOverflow (pg_ref pg)
{
    paige_rec_ptr pg_rec;
    long_overflow_amount;

    pg_rec = UseMemory(pg);
    overflow_amount = pg_rec→overflow_size;
    UnuseMemory(pg);

    return overflow_amount;
}
```

TECH NOTE: Carriage return/line feeds causing problems

Regarding LF/CR characters, OpenPaige handles both of them as a "new line" except a CR. It also starts a new paragraph, but for LF it just does a line feed.

Note that lines that terminate both in LF and CR will cause "two" lines on the screen -- at least in OpenPaige default mode.

You can turn that off, however, if you want LF/CR to be treated as only one líne feed.' To do so, just

set NO_LF_BIT in the OpenPaige attribute flags during pgNew. When this attribute is set, OpenPaige ignores all LFs embedded in the text (they become invisible).

Note that I haven't mentioned what the values are for LF and CR, because those are whatever values sit in OpenPaige globals. Also as he mentioned, MPW will compile \r etc. differently than Symantec so watch out for that. See "Changing Globals" and "CR/LF Conversion".

3.3 A Different Default Font, Style, Paragraph

Any time a new pg_ref is created, OpenPaige sets the initial style_info, font_info and par_info (style, font and paragraph format) to whatever exists in the corresponding field from pg_globals.

Hence, to set default style, font or paragraph format, simply change the respective information in pg_globals (see example below).

To change the default style information, change field(s) in pg_globals.def_style; to change the default font, change field(s) in pg_globals.def_font; to change the default paragraph format, change field(s) in pg_globals.def_par.

You can also set the default low-level callback "hook" functions for style or paragraph records, and even the general OpenPaige functions by placing a pointer to the new function in the respective pg_globals field. See "Customizing OpenPaige".

For example, if you wanted to override the draw-text callback function always for all styles, you would change the default draw-text function in the default style found in pg_globals before your first call to pgNew (but after pgInit:

```
pg_globals.def_style.procs.draw = myTextDrawProc;
```

... where myTextDrawProc is a low-level callback to draw text (see "Setting Style Functions"). If you did this, every new style_info record created by OpenPaige will contain your callback function.

The default hooks for general callbacks not related to styles or paragraph formats are in pg_globals.def_hooks.

See a complete description of style_info, font_info and par_info records under "Style

Basics".

Change defaults after they are created using `pgInit`.

These changes will apply to all `pgNew`s that are called later.

```
void ApplInit() // Initialisation of the App
{
    pgMemStartup(&mem_globals, 0);
    pgInit(&paige_rsrv, &mem_globals);

    /* change to make the default for all pg_refs created herein after
    9 point instead of 12 point is a fraction with hi word being a
    point is a fraction with hi word being the whole point value /*

    paige_rsrv.def_style.point = 0x00090000;
}
```

Default tab spacing

You can also change the default spacing for tabs (the distance to the next tab if no specific tab stops have been defined in the paragraph format). To do so, change `globals.def_par.def_tab_space`.

```
/* The following code changes the default tab spacing (for all subsequent
pg_refs) to $32. */

pgMemStartup(&mem_globals, 0);
pgInit(&paige_rsrc, &mem_globals);
paige_rsrv.def_par.def_tab_space $=32$;
```

3.4 Graphic Devices

As mentioned earlier, a newly created OpenPaige object will always draw to a default device; in a Macintosh environment, for instance, the default device will be the current port that is set before

calling pgNew. In a Windows environment, the default device will be an HDC derived from GetDC(hWnd), where hWnd is the window given to pgNew.

Setting a device

It is possible that you will want to change that default device once an OpenPaige object has been created. To do so, call the following function:

```
(void) pgSetDefaultDevice (pg_ref pg, graf_device_ptr device);
```

The device parameter is a pointer to a structure which is maintained internally (and understood) by OpenPaige. (Generally, you won't be altering its structure directly but the record layout is provided at the end of this section for your reference).

The contents and significance of each field in a graf_device depends on the platform in which OpenPaige is running. However, a function is provided for you to initialise a graf_device regardless of your platform:

```
(void) pgInitDevice (pg_globals_ptr globals, generic_var the_port, long
machine_ref, graf_device_ptr device);
```

The above function sets up an OpenPaige graphics port which you can then pass to pgSetDefaultDevice (you can also use pgInitDevice to set up an alternate port that can be passed to pgDisplay).

The globals parameter is a pointer to the same structure you passed to pgInit.

The actual (but machine-dependent) graphics port is passed in the_port; what should be put in this parameter depends on the platform you are working with, as follows:

- **Macintosh** (and PowerMac) — the_port should be a GrafPtr or CGrafPtr; machine_ref should be zero.
- **Windows** (all OS versions) — the_port should be an HWND and machine_ref should be MEM_NULL. Or, if you only have a device context (but no window), the_port should be MEM_NULL and machine_ref the device context. See sample below.

The device parameter must be a pointer to an uninitialised graf_device record. The function will initialise every field in the graf_device; you can then pass a pointer to that structure to pgSetDefaultDevice.

NOTES

1. If you specified a window during `pgNew()` and want the `pg_ref` to continue displaying in that window, the "default device" is already set, so you do not need to use these functions. The only reason you would/should ever set a default device is if you want to literally change the window or device context the `pg_ref` is associated with.

2. OpenPaige makes a copy of your `graf_device` record when you call `pgSetDefaultDevice`, so the structure does not need to remain static. But the graphics port itself (`HWND` or `HDC` for Windows, or `GrafPtr` for Mac) must remain "open" and valid until it will no longer be used by OpenPaige.

3. If you need to temporarily change the `GrafPtr` (Macintosh) or device context (Windows), see "Quick & easy `set-window`".

CAUTION: Do not set the same `graf_device` as the "default device" to more than one `pg_ref`. If you need to set the same window or device context to more than one `pg_ref`, create a new `graf_device` for each one.

Setting up a `graf_device` for Windows

EXAMPLE 1: Setting up a `graf_device` from a Window handle (HWND)

```
graf_device device;

pgInitDevice(&paige_rsrv, (generic_var)hWnd, MEM_NULL, &device);
pgSetDefaultDevice(pg, &device);

//... other code, draw, paint, whatever.

pgCloseDevice(&paige_rsrv, &device);
```

EXAMPLE 2: Setting up a `graf_device` from a device context only (HDC):

```
graf_device device;

pgInitDevice(&paige_rsrv, MEM_NULL, (generic_var)hDC, &device);
pgSetDefaultDevice(pg, &device);

//... other code, draw, paint, whatever.
```

Setting default device on the Macintosh

```
/* This function accepts a pg_ref (already created) and a Window pointer. The
Window is set to pg's default drawing port, so after a call to this function,
all drawing will occur in a new window. */

void set_new_paige_port (pg_ref pg, WindowPtr new_port)
{
    graf_device paige_port;
    pgInitDevice(&paige_rsrv, new_port, 0, &paige_point);
    pgSetDefaultDevice(pg, &paige_port);
}

/* Done. OpenPaige makes a copy of paige_port so it does not need to be static
*/
```

If you want to obtain the current default device for some reason, you can call the following:

```
(void) pgGetDefaultDevice (pg_ref pg, graf_device_ptr device);
```

The device is copied to the structure pointed to by `device`.

Disposing a device

If you have initialised a `graf_device`, followed immediately by `pgSetDefaultDevice()`, you do not need to deïnitialise or dispose the `graf_device`.

If, however, you have initialised a `graf_device` that you are keeping around for other purposes, you must eventually dispose its memory structures. To so call the following:

```
(void) pgCloseDevice (pg_globals_ptr globals, graf_device_ptr device);
```

This function disposes all memory structure created in device when you called `pgInitDevice`. The `globals` parameter should be a pointer to the same structure given to `pgInit`.

NOTES:

1. `pgCloseDevice` does not close or dispose the `GrafPort` (Macintosh) or the HWND/HDC (Windows) — you need to do that yourself.
2. You should never dispose a device you have set as the default device because `pgDispose` will call `pgCloseDevice`. The only time you would use `pgCloseDevice` is either when you have set up a `graf_device` to pass as a temporary pointer to `pgDisplay` (or a similar function that accepts a temporary port) in which OpenPaige does not keep around, OR when you have changed the default device (see note below).
3. Additionally: OpenPaige does not dispose the previous default device if you change it with `pgSetDefaultDevice`. Thus, if you change the default you should get the current device (using `pgGetDefaultDevice`), set the new device then pass the older device to `pgCloseDevice`.

Quick & easy `set-window`

In certain situations you might want to temporarily change the window or device context a `pg_ref` will render its text drawing. While this can be done by initialising a `graf_device` and giving that structure to `pgSetDefaultDevice()`, a simpler and faster approach might be to use the following functions:

```
generic_var pgSetDrawingDevice (pg_ref pg, const generic_var draw_device);
void pgReleaseDrawingDevice (pg_ref pg, const generic_var previous_device);
```

The purpose of `pgSetDrawingDevice` is to temporarily change the drawing device for a `pg_ref`. The `draw_device` parameter must be a `WindowPtr` (Macintosh) or a device context (Windows).

The function returns the current device (the one used before `pgSetDrawingDevice`).

NOTE: "device" in this case refers to a machine-specific device, not a `graf_device` structure.

You should call `pgReleaseDrawingDevice` to restore the `pg_ref` to its previous state. The `previous_device` parameter should be the value returned from `pgSetDrawingDevice`.

Temporarily changing the HDC (Windows)

```
/* This function forces a pg_ref to display inside a specific HDC instead of
the default. */

void DrawToSomeHDC (pg_ref pg, HDC hDC)
{
    generic_var old_dc;
    old_dc = pgSetDrawingDevice(pg, (generic_var)hDC);
    pgDisplay(pg, NULL, MEM NULL, MEM NULL, NULL, best_way);
    pgReleaseDrawingDevice(pg, old_dc);
}
```

Setting a Scaled Device Context (Windows only)

On a Windows platform, in certain cases you will want to preset a device context that needs to scale all drawing. However, using the standard function to set a device into an OpenPaige object (`pgSetDrawingDevice`) will not work in this case because OpenPaige will want to clear your mapping mode(s) and scaling factor(s).

The solution is to inform OpenPaige that you wish to set your own device context, but to include a scaling factor:

```
generic_var pgSetScaledDrawingDevice (pg_ref pg, const generic_var draw_device,
pg_scale_ptr scale);
```

This is identical to `pgSetDrawingDDevice()` except that it contains the additional parameter `scale` which specifies the scaling factor. For more information on OpenPaige scaling, see the appropriate section(s).

3.5 Colour Palettes (Windows-specific)

```
void pgSetDevicePalette (pg_ref pg, const generic_var palette); generic_var
pgGetDevicePalette (pg_ref pg);
```

These Windows-specific functions are used to select a custom palette into the device context of a `pg_ref`. To set a palette, call `pgSetDevicePalette()` and pass the `HPALETTE` in `palette`. If you want to clear a previous palette, pass `(generic_var)0`.

Setting a palette causes OpenPaige to select that palette every time it draws to its device context.

To obtain the existing palette (if any), call `pgGetDevicePalette()`

CAUTION: Do not delete the palette unless you first clear it from the `pg_ref` by calling `pgSetDevicePalette(pg, (generic_var)0)`.

CAUTION: If you change the default device (`pgSetDefaultDevice`), you need to set the custom palette again.

NOTE: OpenPaige does not delete the `HPALETTE`, even during `pgDispose()`. It is your responsibility to delete the palette.

3.6 Changing Shapes

You can change the `vis_area`, the `page_area` and/or the `exclude_area` of an OpenPaige object at any time (see "Creating an OpenPaige Object" for a description of each of these parameters):

```
(void) pgSetAreas (pg_ref pg, shape_ref vis_area, shape_ref page_area,
shape_ref exclude_area);
```

The `vis_area`, `page_area`, and `exclude_area` are functionally identical to the same parameters passed in `pgNew`. Of course, you could have passed any of these shapes in `pgNew`, but the purpose of `pgSetAreas` is to provide a way to change the visual area and/or wrap area and/or exclusion areas some time after an OpenPaige object has been created.

Any of the three "`_area`" parameters can be `MEM_NULL`, in which case that shape remains unchanged.

Subsequent drawing of `pg`'s text will reflect the changes, if any, produced by the changed shape(s).

A typical reason for changing shapes would be, for example, to implement a "set columns" feature. The initial OpenPaige object might have been a simple rectangle ("normal" document), but let us suppose that the user later wishes to change the document to three columns. To do so, you could set up a `page_area` shape for three columns and pass that new shape to `page_area` and null pointers for the other two areas. The OpenPaige object, on a subsequent `pgDisplay`, would rewrap the text and flow within these "columns."

NOTES

1. If your area(s) are simple rectangles, it may prove more efficient to use `pgSetAreaBounds()` in this chapter.
2. If you simply want to "grow" the `vis_area` (such as responding to a user changing the window's size), see "'Growing' The Visual Area" for information on `pgGrowVisArea`.
3. OpenPaige makes a copy of the new shape(s) you pass to `pgSetAreas`. You can therefore dispose these shapes any time afterwards.

For information on constructing various shapes, see "All About Shapes".

If you are implementing containers, see "Containers Support".

"Growing" The Visual Area

If you want to change the `vis_area` (area in which text displays) in response to a user enlarging the window's width and height, call the following:

```
(void) pgGrowVisArea (pg_ref pg, co_ordinate_ptr top_left, co_ordinate_ptr
bot_right);
```

The size of `vis_area` shape in `pg` is changed by adding `top_left` and `bot_right` values to `vis_area`'s top-left and bottom-right corners, respectively.

By "adding" is meant the following: `top_left.v` is added to `vis_area`'s top and `top_left.h` is added to `vis_area`'s left; `bot_right.v` is added to `vis_area`'s bottom and `bot_right.h` is added to `vis_area`'s right.

NOTE: This function adds to (or "subtracts" from, if coördinate parameters are negative) the visual

area rather than setting or replacing the visual area to the given coördinates.

Either `top_left` or `bot_right` can be null pointers, in which case they are ignored.

NOTE: This function only works correctly if `vis_area` is rectangular; if you have set a non-rectangular shape, you need to reconstruct your `vis_area` shape and change it with `pgSetAreas`.

Responding to WM SIZE message (Windows)

```
case WM_SIZE:
    if (pgRef)
    {
        rectangle vis_bounds;
        co_ordinate amount_to_grow;
        long old_width, new_width, old_height, new_height;
        pgAreaBounds(pgRef, NULL, &vis_bounds);
        new_width = (long) LOWORD(lParam);
        new_height = (long) HIWORD(lParam);
        old_width = vis_bounds.bot_right.h - vis_bounds.top_left.h;
        old_height = vis_bounds.bot_right.v - vis_bounds.top_left.v;
        amount_to_grow.h = new_width - old_width;
        amount_to-grow.v = new_height - old_height;
        pgGrowVisArea(pgRef, NULL, (co_ordinate_ptr) &amount_to_grow);
    }
break;
```

3.7 Getting information about shapes

Getting Current Shapes

To obtain any of the three shapes in an OpenPaige object, call the following:

```
(void) pgGetAreas (pg_ref pg, shape_ref vis_area, shape_ref page_area,
shape_ref exclude_area);
```

The `vis_area`, `page_area`, `exclude_area` must be pre-created `shape_ref`s (see below). Any of

them, however, can be MEM_NULL (in which case that parameter is ignored).

This function will copy the contents of pg's visual area, wrap area, and exclude area into vis_area, page_area and exclude_area, respectively, if that parameter is non-null.

Helpful hint: The easiest way to create a shape_ref is to call pgRectToShape passing a null pointer to the rect parameter, as follows:

```
shape_ref new_shape;
new_shape = pgRectToShape(&paige_rsrv, NULL);
```

The paige_rsrv parameter in the above example is a pointer to the same pg_globals passed to pgInit. By providing a null pointer as the second parameter, a new shape_ref is returned with an empty shape (all sides zero).

'Get/Set Areas' Trick

If you are using simple rectangles for the visual area or wrap (page) area in an OpenPaige object, and/or if you simply want to know the bounding rectangular area of either shape, use the following instead of pgGetAreas:

```
(void) pgAreaBounds (pg_ref pg, rectangle_ptr page_bounds, rectangle_ptr
vis_bounds);
```

When pgAreaBounds is called, page_bounds gets set to a rectangle that encloses the entire page_area and vis_bounds gets set to a rectangle that encloses the entire vis_area of pg.

If you don't want one or the other, either page_bounds or vis_bounds can be a null pointer.

This function is useful when you simply want the enclosing bounds of either shape because you do not need to create a shape_ref.

You can also set the page area and/or vis area by calling pgSetAreaBounds, which accepts a pointer to a rectangle in page_bounds and vis_bounds (of which either can be a null pointer). Note that this is faster and simpler than pgSetAreas, except that it only works provided that the shape(s) are single rectangles.

Direct Shape Access

You can also access the shape_ref s in an OpenPaige object directly using any of the following:

```
(shape_ref) pgGetPageArea (pg_ref pg);
(shape_ref) pgGetVisArea (pg_ref pg);
(shape_ref) pgGetExcludeArea (pg_ref pg);
```

These three functions will return the shape_ref for page_area, vis_area and exclude_area, respectively. Neither will ever return MEM_NULL (even if you provided MEM_NULL for exclude_area in pgNew, for instance, OpenPaige will still maintain a shape_ref for the exclusion, albeit an empty shape).

The purpose of these functions is for special applications that need to look inside of OpenPaige shape as quickly and as easily as possible.

CAUTION: These functions return the actual memory_ref's for each shape. You must therefore never dispose of them, nor should you alter their contents (or else OpenPaige won't know you have changed anything and word wrapping and display will fail). If you want to alter the contents of OpenPaige shapes, see "Containers Support" and "Exclusion Areas".

Getting Shape Rectangle Quantity

You can find out how many rectangles comprise any shape by calling the following:

```
(pg_short_t) pgNumRectsInShape (shape_ref the_shape);
```

The function will return the number of rectangles in the_shape.

NOTE: The result will always be at least 1, even for an empty shape. Any "empty" shape is still one rectangle whose boundaries are $0,0,0,0$. If you need to detect whether or not a shape is empty, call:

```
(pg_boolean)pgEmptyShape(the_shape); /* Returns TRUE if empty */
```

3.8 Changing Globals

As mentioned several times, your application provides a pointer to `pgInit` (and other places) to be used by OpenPaige to store certain global variables. This structure is initially set to certain default values, but you can make certain changes that apply to your particular application.

For example, OpenPaige globals contain the values for special control codes such as CR, LF, and arrow keys, but there are instances when you need to change some of these "characters" to a different value.

Another (more common) reason to change OpenPaige globals is to force a default text or paragraph format for all subsequent `pgNew()` calls.

Since your application maintains the globals record, there are no functions provided to change its contents; rather, you alter the structure's contents directly some time after `pgInit`.

NOTE: The entire OpenPaige globals structure can be viewed in `paige.h`. Only the members of this structure that you are allowed to alter are shown unless noted otherwise.

```
/* Paige "globals" (address space provided by app): */

struct pg_globals
{
    pgm_globals_ptr mem_globals;            // Globals for pgMemManager
    long            max_offscreen;          // Maximum memory for offscreen
    long            max_block_size;         // Maximum size of text block
    long            minimum_line_width;     // Minimum size line width
    long            def_tab_space;          // Default tab spacing for pgNew
    pg_short_t      line_wrap_char;         // <CR> character
    pg_short_t      soft_line_char;         // Soft <CR> character
    pg_short_t      tab_char;               // Tab character
    pg_short_t      soft_hyphen_char;       // Soft hyphen character
    pg_short_t      bs_char;                // Backspace character
    pg_short_t      ff_char;                // Form feed chr (for page breaks)
    pg_short_t      container_brk_char;     // Container break character
    pg_short_t      left_arrow_char;        // Left arrow
    pg_short_t      right_arrow_char;       // Right arrow
    pg_short_t      up_arrow_char;          // Up arrow
    pg_short_t      down_arrow_char;        // Down arrow
    pg_short_t      fwd_delete_char;        // Forward delete character
    pg_char         hyphen_char[4];         // Hard hyphen character
    pg_char         decimal_char[4];        // "." char (for decimal tabs)
                                            /* Visible surrogate for:
                                            ------------------------*/
    pg_char         cr_invis_symbol[4];     // carriage return
    pg_char         lf_invis_symbol[4];     // line feed
```

```
    pg_char         tab_invis_symbol[4];    // horizontal tab
    pg_char         end_invis_symbol[4];    // end-of-document
    pg_char         pbrk_invis_symbol[4];   // break-of-page
    pg_char         cont_invis_symbol[4];   // container break
    pg_char         space_invis_symbol[4];  // space
                                            //-----------------------
    pg_char         flat_single_quote[4];   // Single "typewriter" quote
    pg_char         flat_double_quote[4];   // Double "typewriter" quote
    pg_char         left_single_quote[4];   // Single left smart quote
    pg_char         right_single_quote[4];  // Single right smart quote
    pg_char         left_double_quote[4];   // Double left smart quote
    pg_char         right_double_quote[4];  // Double right smart quote
    pg_char         elipse_symbol[4];       // Char to draw for ellipse
    long            invis_font;             // Machine-specific invisible char
font
    pg_char         unknown_char[4];        // Used for unsupported characters
    long            embed_callback_proc;    // Used internally by embed_refs
    font_info       def_font;               // Default font for all pgNew's
    style_info      def_style;              // Default style for all pgNew's
    par_info        def_par;                // Default para for all pgNew's
    color_value     def_bk_color;           // Default background colour
    color_value     trans_color;            // Transparent colour (default is
white)
    pg_hooks        def_hooks;              // Default general hooks
    // miscellaneous fields not to be altered by app.
};
```

The following is a description for each field that you can change directly:

`max_offscreen` — defines the maximum amount of memory, in bytes, that can be used for offscreen bit map drawing. The purpose of this field is to avoid excessive, unreasonable offscreen bit maps for huge text on high-density monitors.

`max_block_size` — defines the largest size for contiguous text (OpenPaige breaks down text into blocks of `max_block_size` as the OpenPaige object grows).

`minimum_line_width` — pdefines the smallest width allowed, in pixels, for a line of text. The purpose of this field is for OpenPaige to decide when a portion of a wrap area is too small to even consider placing text.

`def_tab_space` — not used in version 1.3 and beyond. (To change default tab spacing, change `globals.def_par.def_tab_space`).

`line_wrap_char` through `down_arrow_char` — defines all the special characters recognized by

OpenPaige. Any of these can be changed to something else if you don't want the default values. See **Warning** below. See also "Double Byte Defaults".

`text_brk_char` — defines an alternate character to delineate text blocks (OpenPaige partitions large blocks of text into smaller blocks; by default, a block will break on a <CR> or <LF>, but if neither of those are found in the text, the `text_brk_char` will be searched for). For additional information, see "Anatomy of Text Blocks" in the Appendix.

`null_char` — defines a special character that, if inserted, merely causes word-wrap to recalculate and the `null_char` itself is not inserted.

`cr_invis_symbol` through `space_invis_symbol` — define all the character values to draw when OpenPaige is in "show invisibles" mode. Each character is represented by a null-terminated Pascal string (first byte is the length, followed by the byte(s) for the character, followed by a zero). Note that these characters can be zero, one or two bytes in length. See also “Double Byte Defaults”.

`flat_single_quote` through `right_double_quote` - define single and double quotation characters for "smart quotes" implementation. The "flat*" quote characters should be the standard ASCII characters for single and double quotes, while the "left*" and "right_" quote characters are to be substituted for "smart quotes" if that feature has been enabled.

`elipse_symbol` — contains the character to draw an ellipsis “..." symbol. However, this character definition has been provided only for future enhancement: the current version of OpenPaige does not use this character for any built-in feature.

`invis_font` — defines the font to be used for drawing invisibles. This is machine dependent. For Macintosh, this is the QuickDraw font ID that gets set for invisible characters. For Windows, this is a font `HANDLE` (which you can alter by replacing it with you own font `HANDLE`)

`unknown_char` — Contains the symbol to be used when importing unsupported characters. For example, importing a file with OpenPaige's import extension may include characters that to not cross over to any available character set, in which case unknown_char will be substituted.

WARNINGS

- *(Windows only)* If you replace the `invis_font` member with your own font object, do not delete the object that was there before, if any. Moreover, OpenPaige will not delete your `invis_font` object either, so you are responsible for deleting your own object before your application quits.
- The default machine-specific functions within OpenPaige are assuming ASCII control codes for the special character values in `pg_globals` (ASCII chars < 20).

`def_font`, `def_style`, `def_par` — define the default font, text and paragraph formatting,

respectively. Whenever you call `pgNew()`, these three structures are literally copied into the new `pg_ref`. Hence, to change the default(s) for text formatting, you simply change the members of these three structures prior to calling `pgNew()`.

`def_bk_color`, `trans_color` — define the default background colour to be used for drawing all text and the colour that is considered "transparent", respectively. The background colour is not necessarily the same as the window's background colour (OpenPaige will make the necessary adjustment if window colour does not equal the `pg_ref`'s background colour). By "transparent" color is meant which colour is considered the normal screen background colour (default is white).

The purpose of defining the transparent colour is to inform OpenPaige when and if the background of its drawing needs to be "erased" with a different colour other than the regular background of the window. If the background colour for an OpenPaige object is set to the same value as `trans_color` in `pg_globals`, OpenPaige won't do any special color filling of background since it assumes normal erasing of the window will take care of it (for instance, responding to WM_PAINT). If OpenPaige's background color is not the same as `trans_color`, then the `pg_ref`'s background shape will be pre-filled with a different color other than the window's default.

`def_hooks` — define the default function pointers to be used for a `pg_ref`'s general hooks. Essentially, `pgNew` copies these pointers. (DSI and other developers can change these defaults for special extensions).

Default Values

After you have called $p g I n i t$, the following defaults are set for all the fields mentioned above:

Global Field	About the field	Windows	Macintosh
`max_offscreen`	bit map size (bytes)	48,000	48,000
`max_block_size`	max paragraph size in number of ` ` characters	4096	4096
`minimum_line_width`	in pixels	16	16
`line_wrap_char`	carriage return character	0x0D	0x0D
`soft_line_char`	soft carriage return char	0x0A	0x0A
`tab_char`	tab char	0x09	0x09
`hyphen_char`	hard hyphen char	0x2D	0x2D
`soft_hyphen_char`	soft hyphen char	0x1F	0x1F
`decimal_char`	decimal point char	0x2E	0x2E
`bs_char`	back space (delete) char	0x08	0x08
`lf_char`	line feed char	0x0C	0x0C

container_brk_char	container break char	0x0E	0x0E
left_arrow_char	left arrow key	0x1C	0x1C
right_arrow_char	right arrow key	0x1D	0x1D
up_arrow_char	up arrow key	0x1E	0x1E
down_arrow_char	down arrow key	0x1F	0x1F
text_brk_char	alternative carriage return char (form feed)	0x1B	0x1B
fwd_delete_char	forward delete key	0x7F	0x7F
elipse_symbol	displayed when OpenPaige encounters an unknown symbol	''	0x85
flat_single_quote	straight apostrophe – '	0x27	0x27
flat_double_quote	straight double quote – "	0x22	0x22
left_single_quote	curly left quote – '	0x91	0xD4
right_single_quote	curly right quote – '	0x92	0xD5
left_double_quote	curly left quotes – “	0x93	0xD2
right_double_quote	curly right quotes – "	0x94	0xD3
cr_invis_symbol	carriage return when invisibles are displayed	¶ (0xB6)	¶ (0xA6)
lf_invis_symbol	line feed when invisibles are displayed	¼ (0xB5)	¼ (0xB9)
tab_invis_symbol	tab when invisibles are displayed	0x95	0x13
end_invis_symbol	end of document when invisibles are displayed	× (0xB5)	× (0xB0)
cont_invis_symbol	container break when invisibles are displayed	| (0xA5)	| (0xAD)
space_invis_symbol	space symbol when invisibles are displayed	. (0x2E)	. (0x2E)
invis_font	font in which invisibles are displayed	default font*	0 (Chicago)
def_font	font (name) used for pgNew()	“System”	Application font
def_style	text format used for pgNew()	Plain, 12 point	Plain, 12 point
def_par	paragraph format used for pgNew ()	Indents all zero, tab spacing 24 pixels	Indents all zero, tab spacing 24 pixels
def*bk*color	background color used to fill page area for all pg_refs	white	white
trans_color	color assumed also to be window’s background	white	white

If the default font is zero, then OpenPaige creates a font object using the default found in `pg_globals` record that was created with `pgNew`. If you want to change this you can change the default font in the `pg_globals`.

NOTE (Macintosh): The `pgdf` Resource: During initialisation, the machine-specific code for Macintosh searches for a special resource to determine the character defaults (above). If it does not find this resource, the values given above are used. Hence, you can change the defaults by changing the contents of this resource:

TABLE #2 MACINIOSH RESOURCE TYPE & ID

Resource Type	Resource ID
"pgdf'	128

The OpenPaige package we provide should contain this resource as well as a ResEdit template to change its contents.

Double Byte Defaults

Each character default in `pgGlobals` can be "double byte" such as Kanji, if necessary. Although this manual references these defaults as "characters," in truth these global values are ALL double-byte, that is they are unsigned integers. An ASCII CR, for instance, is considered to be 0x000D and not 0x0D, etc. To set a double byte default, such as a Kanji decimal for instance, simply place the whole 16-bit value into the appropriate global field.

TECH NOTE (CR/LF Conversion)

> I have read all the stuff so far about carriage return line feeds. What exactly do I have to do to make sure my documents are portable between the PC which uses `<CR><LF>`, and the Mac which uses only a `<CR>`?

OpenPaige normally formats text using only CR for paragraph endings (NOT CR/LF), hence for documents created from scratch on any of the platforms, where all text has been entered by the user via the keyboard, documents between platforms are generally portable with respect to CR/LF or just CR.

The only time this can become even remotely an issue is when raw text is inserted which contains both CR and LF, which if left "as is" would cause OpenPaige to draw two line feeds for each paragraph ending (one for CR and one for LF).

To avoid this situation, the NO_LF_BIT should be set as one of the "flag" bits in pgNew (or, if the pg_ref has already been created, NO LF BIT can be set by calling pgGetAttributes, ORing NO_LF_BIT to the result and setting that value with pgSetAttributes. By setting this bit, OpenPaige will essentially ignore all LF characters and they will become virtually invisible.

See also "Carriage return/line feeds causing problems".

3.9 Cloning an OpenPaige Object

To create a new OpenPaige object based on an existing pg_ref's vis_area, page_area, exclude_area and attributes, use the following:

```
(pg_ref) pgDuplicate (pg_ref pg);
```

FUNCTION RESULT: This function returns a new pg_ref, completely independent of, but using the same shapes and attributes as, pg. No text is copied and the default text formatting is used.

3.10 Storing Arbitrary References and Structures

You can store any arbitrary long value or pointer into a pg_ref any time you want, and with as many different values as you want by using the following:

```
(void) pgSetExtraStruct (pg_ref pg, void PG_FAR *extra_struct, long ref_id);
(void PG_FAR *) pgGetExtraStruct (pg_ref pg, long ref_id);
```

By "storing" an arbitrary value within a pg_ref is meant that OpenPaige will save longs or pointers — which only have significance to your application — which can be retrieved later at any time.

To store such items, call pgSetExtraStruct, passing your long (or pointer) in extra_struct and a unique identification number in ref_id. The purpose of this UID is to reference that item later in pgGetExtraStruct.

However, if the value in `ref_id` is already being used by an "extra struct" item within `pg`, the old value is overwritten with `extra_struct`. (Hence, that is how you can "change" a value that had previously been stored).

To retrieve an item stored with `pgSetExtraStruct`, call `pgGetExtraStruct` passing the wanted ID in `ref_id` (which must be the same number given to `unique_id` for that item originally given to `pgSetExtraStruct`).

See "OpenPaige "Handler" Functions".

TECH NOTE (Removing ExtraStruct)

> Why is there no `pgRemoveExtraStruct()`?

Probably because of the way it was implemented and what it is/was intended for doesn't make sense to do a "remove."

An "extra `struct`", as far as OpenPaige is concerned, is a single element of an array of `long`s. Each of these `long`s are treated as `refcon` values that an application can use for whatever.

Literally, the list of extra `struct`s are maintained internally as `long[n]` where `n` is the number of extra `struct`s added.

The array number itself, e.g. 0, 1, 2, etc., is the "ID number" of the extra `struct`. That is what makes each one unique, really. Hence you can see why we could not really "delete" one of these elements since that would cause all subsequent extra `struct` elements to be a different "ID" number.

For example, if a `pg_ref` holds elements 0, 1, 2, 3, and 4 (all with same corresponding ID numbers), deleting 2 would make 3 become 2 and 4 become 3.

We realise a more elaborate system could have been implemented that contained indirect pointers, or some other scheme that is closer to what (I think) you are suggesting, so extra `struct`s could be deleted.

But, the original purpose of this feature was simply to add extra `refCon` possibilities. It might make more sense if we called the function something like `pgReserveAnotherLongRefCon`.

Finding a Unique ID

If you aren't sure whether or not an ID number is unique for a `pg_ref`, or if you simply want to get

an ID number that you know is unique, call the following:

```
(long) pgExtraUniquelD (pg_ref pg);
```

The number this function returns will always be positive and is guaranteed to have not yet been used for `pgSetExtraStruct` with this `pg_ref`.

CAUTION: OpenPaige has no idea what you are storing with `pgSetExtraStruct`, and therefore will not dispose any memory allocations that you might have attached to "extra `struct`" storage. Be sure to dispose any such allocations before disposing the `pg_ref` or you will end up with a memory leak.

NOTE: Once you have stored something with `pgExtraStruct`, that item (and unique reference) stays in the `pg_ref` and never gets "removed" unless you explicitly do another `pgSetExtraStruct` using the same ID (in which case the previous item associated with that ID will get overwritten).

EXAMPLE (How to use and extra struct)

```
/* This function adds a WindowPtr to the OpenPaige object using
   the extra struct feature and returns the ID of that struct */

short add_window_to_pg (pg_ref pg, WindowPtr w_ptr)
{
    short unique_id;
    unique_id = pgExtraUniqueID(pg);
    pgSetExtraStruct(pg, w_ptr, unique_id);
    return unique_id;
}

/* Later, the extra struct can be accessed using the ID returned above. */

WindowPtr window_with_pg;
window_with_pg = pgGetExtraStruct(pg, unique_id);
```

3.11 Cursor Utilities

If you want to know if a point (co_ordinate) sits on top of editable text (to change the mouse symbol to something else, for instance), call the following:

```
(short) pgPtInView (pg_ref pg, co_ordinate_ptr point, co_ordinate_ptr
offset_extra);
```

Given an arbitrary window coördinate (relative to that window's coördinate system) in point, pgPtInView returns information about what part of pg, if any, includes that point.

The offset_extra parameter is an optional pointer to a coördinate that holds values to temporarily offset everything in pg before checking intersections of the point. In other words, if offset_extra is non-null, this visual area in pg will first be offset by offset_extra.h and offset_extra.v amounts before checking the containment of point in vis_area; the wrap area will also be offset by this amount before checking if the wrap area contains the point, and so on.

If offset_extra is a null pointer, everything is checked as-is.

FUNCTION RESULT: The function result will be a word containing different bits set (or not) indicating what intersects the point as follows:

```
#define WITHIN_VIS_AREA     0x0001  // Point within vis_area
#define WITHIN_WRAP_AREA    0x0002  // Point within page_area
#define WITHIN_EXCLUDE_AREA 0x0004  // Point within exclude_area
#define WITHIN_TEXT         0x0008  // Point within actual text
#define WITHIN_REPEAT_AREA  0x0010  // Point is in repeating gap of page
#define WITHIN_LEFT_AREA    0x0020  // Point is left of document
#define WITHIN_RIGHT_AREA   0x0040  // Point is right of document
#define WITHIN_TOP_AREA     0x0080  // Point is above top of document
#define WITHIN_BOTTOM_AREA  0x0100 // Point is below bottom of document
```

WITHIN_VIS_AREA means the point is within the bounding area of vis_area.

WITHIN_WRAP_AREA means the point is somewhere within the page_area shape.

WITHIN_EXCLUDE_AREA means the point is somewhere within the exclude_area.

WITHIN_TEXT means the point is somewhere within "real" text. This differs from WITHIN_WRAP_AREA since it is possible to have a large page_area shape with very little text (in which case, WITHIN_TEXT will only be set if the point is over the portion that displays text).

Each bit gets set notwithstanding the other settings. For example, WITHIN_EXCLUDE_AREA and

WITHIN_WRAP_AREA can both be set, even though text cannot flow into the exclude_area.

Another setting that can be returned is WITHIN_TEXT set but WITHIN_VIS_AREA not set, which really means the point is over text that falls outside of vis_area. The function result is simply the setting for each case individually, so it is your responsibility to examine the combination of bits to determine what action you should take, if any.

NOTE: The best time to turn the cursor to an "i-beam" is when pgPtInView returns WITHIN_VIS_AREA and WITHIN_TEXT at the same time and pg is in an active state.

3.12 Getting Text Size and Height

To obtain the total size of text in an OpenPaige object (in bytes), call the following:

```
(long) pgTextSize (pg_ref pg);
```

FUNCTION RESULT: This function returns the total size of text (byte size) in pg.

To find out how "tall" the text is, call the following:

```
(long) pgTotalTextHeight (pg_ref pg, pg_boolean paginate)
```

FUNCTION RESULT: This function returns the distance between the top of the first line of text to the bottom of the lowest line, in pixels.

NOTE: The lowest line is not necessarily the last line in the document: had pg had a non-rectangular shape, such as parallel columns, the last (ending) line could have been vertically above some of the lines in other areas of the shape. Hence, pgTotalTextHeight really returns the bounding height between the highest and lowest points.

If paginate is "TRUE," all the text from top to bottom is recalculated (word wrap recomputed), if necessary. If paginate is "FALSE," the total text height returned is computed with the latest information available within pg. In essence, this would be OpenPaige's "best guess."

For example, suppose a large document changed from 12 point text to 18 point text and you wanted to know how tall the document had become. To get the exact height, to the nearest pixel, you should pass TRUE for paginate, otherwise OpenPaige might not have computed all the text to return an exact answer. However, computing large amounts of text can consume a great deal of

time, which is why the choice to "paginate" or not has been provided.

NOTES:

1. If you will be using the built-in scrolling support in OpenPaige, you probably never need to get the height of an OpenPaige object — see "All About Scrolling". If you do need an exact height for other reasons, see "Getting the Max Text Bounds".
2. The "height" returned from this function does not consider any extra structures that aren't embedded in the text stream. For example, if you have implemented headers, footers, footnotes, or any other page "ornaments" their placement will not be considered in the text height computation.

4 Virtual Memory

4.1 Initialising Virtual Memory

OpenPaige supports a "virtual memory" system in which memory allocations made by OpenPaige can be spooled to a disk file in order to free memory for new allocations.

However, your application must explicitly initialise OpenPaige virtual memory before it is operational; this is because disk file reading and writing is machine-dependent, hence your application needs to provide a path for memory allocations to be saved.

To do so, call the following function somewhere early when your application starts up and after `pgInit`:

```
#include "pgMemMgr.h"
void InitVirtualMemory (pgm_globals_ptr globals, purge_proc purge_function,
long ref_con);
```

The `globals` parameter is a pointer to a field in `pg_globals` (same structure you gave to `pgInit`). For example, if your `pg_globals` structure is called `paige_rsrv`, this parameter would be passed as follows:

```
&paige_rsrv.mem_globals
```

Parameters

- `purge_function` — a pointer to a function that will be called by OpenPaige to write (save) and purge memory allocations and/or to read (restore) purged allocations. However, OpenPaige will use its own function for `purge_proc` if you pass a null pointer for `purge_proc`. Otherwise, if you need to write your own, see "Providing Your Own Purge Function" for the definition and explanation of this function.
- `ref_con` — contains the necessary information for the purge function to read and write to the disk and what you pass to `ref_con` depends on the platform you are operating and/or whether or not you are using the standard purge function (`purge_function null`).

How to set up virtual memory (Macintosh)

```
// This function inits VM by setting up a temp file in System folder

pg_globals paige_rsrv;  // Same globals as given to pgInit, pgNew
void init_paige_vm(void)
{
    SysEnvRec theWorld;
    sysEnvirons(2, &theWorld);  // Get system info for "folder"

    // Get whatever temp file name to use (in this example I get a STR#)

    GetIndString(temp_file_name, MISC_STRINGS, TEMP_FILE_STR);
    Create(temp_file_name, theWorld.sysVRefNum, TEMP_FILE_TYPE);
    FSOpen(temp_file_name, theWorld.SysVRefNum, &vm_file);
    InitVirtualMemory(&paige_rsrv.mem_globals, NULL, vm_file);

    // Leave temp file open until quit (see below)
}

// Before quit, "shut down" VM by closing temp file

void uninit_paige_vm(void)
{
    FSClose(paige_rsrv.purge_ref_con);  // VM file stored here
```

```
}
```

4.2 The scratch file

Assuming you will be passing a null pointer to `purge_proc`, letting OpenPaige use the built-in purge function, the steps to initialise virtual memory fully are as follows:

1. First, call `pgMemStartup` to initialise the OpenPaige Memory Allocation manager, and pass the maximum memory you want OpenPaige using to `max_memory` before allocations begin purging. If you want OpenPaige to use whatever is available, pass 0 for `max_memory` (see `pgMemStartup` in the index for additional information).
2. Create a file that can be used as a "temp" file and open it with read/write access.
3. Call `InitVirtualMemory`, passing the file reference from §2 in the `ref_con` parameter. (For **Macintosh** platform, this should be the file `refnum` of the opened file; for **Windows** platform, this should be the `int` returned from `OpenFile` or `GetTempFile`, etc.).
4. Keep the scratch file open until you shut down the Allocation Manager with `pgMemShutdown`.

NOTE: It is your responsibility to close and/or delete your temp file after your application session with OpenPaige has terminated.

If you are writing your own purge function, however, `ref_con` can be anything you require to initialise virtual memory I/O, such as a file reference or a pointer to some structure of your own definition.

After calling the above function, memory allocations will be "spooled" to your temp file as necessary to create a virtual memory environment.

The value originally passed to `pgMemStartup` — `max_memory` — dictates the maximum memory available for the OpenPaige Allocation Manager before allocations must be purged. This is a logical partition, not necessarily physical (i.e., you might have 2 GiB available but only want OpenPaige to use 50 MiB, in which case you would pass 52428800 to `max_memory` in `pgMemStartup`).

Providing Your Own Purge Function

In most cases you can use the purging utilities provided in the Allocation Manager, see "Purging Utilities". However, you can bypass the built-in memory purge function, if necessary. For complete details, see "Writing Your Own Purge Function".

5 Cut, Copy, Paste

This section explains how to implement Cut, Copy, Paste and Undo, including additional methods to copy "text only."

5.1 Copying and Deleting

```
(pg_ref) pgCut (pg_ref pg, select_pair_ptr selection, short draw_mode);
(pg_ref) pgCopy (pg_ref pg, select_pair_ptr selection);
(void) pgDelete (pg_ref pg, select_pair_ptr delete_range, short draw_mode);
```

To perform a "Cut" operation — for which text is copied then deleted — call `pgCut`. The selection parameter is an optional pointer to a pair of text offsets from which to delete text. This is a pointer to the following structure:

```
typedef struct
{
    long begin; // Beginning offset of some text portion
    long end;   // Ending offset of some text portion
}
select_pair, *select_pair_ptr;
```

The `begin` field of a `select_pair` defines the beginning text offset and the `end` field defines the ending offset. Both offsets are byte offsets, not character offsets. Text offsets in OpenPaige are zero-based (first offset is zero). The last character "end" is included in the selection.

FIGURE 3 SELECTION BEGIN AND END EXPLAINED

NOTE: All offsets are byte counts. In the case of characters, they are each one byte.

If the selection parameter in `pgCut` is a null pointer, the current selection in `pg` is used instead (which is usually want you want).

FUNCTION RESULT: The function result of `pgRef` is a newly created OpenPaige object containing the copied text and associated text formatting. You can then pass this `pg_ref` to `pgPaste`, below.

`draw_mode` can be the values as described in "Draw Modes":

```
draw_none,      // Do not draw at all
best_way,       // Use most efficient method(s)
direct copy,    // Directly to screen, overwrite
direct_or,      // Directly to screen, "OR"
direct xor,     // Directly to screen, "XOR"
bits_copy,      // Copy offscreen
bits or,        // Copy offscreen in "OR" mode
bits_xor        // Copy offscreen in "XOR" mode
```

NOTES:

1. The `pg_ref` returned from `pickup` is a "real" OpenPaige object, which means you need to eventually dispose of it properly using `pgDispose`.
2. Shapes from the source `pg_ref` are used to "clone" the resulting `pg_ref` from a copy or cut regardless of the selection range. For example, if the source `pg_ref` that gets copied contained a `page_area` shape with dimensions 10, 10, 580, 800, the resulting `pg_ref` will have the same `pg_area` shape. The same is true for `vis_area` and `exclude_area`.

CAUTION: If there is nothing to copy (no selection range exists), both `pgCut` and `pgCopy` will return `MEM_NULL`.

CAUTION: It is wise never to display the resulting `pg_ref` unless you first set a default graphics device to target the display. For example, doing a `pgCopy` then drawing to a "clipboard" window later could result in a crash. This can happen if the original window containing the copied `pg_ref` has been closed (rendering an invalid window attached to the copied reference). Hence, before drawing to such a "clipboard", use `pgSetDefaultDevice`. See "Setting a device".

The `pgCopy` function is identical to `pgCut` except that no text is deleted, only a `pg_ref` is returned which is the copy of the specified text and formatting and no `draw_mode` is provided (because the

source pg_ref remains unchanged).

OpenPaige provides excellent error checking for out-of-memory situations with pgCopy. See "Exception Handling".

The pgDelete function is the same as pgCut in every respect except that a "copy" is neither made nor returned. Use this function when you simply want to delete a selection range but not make a copy (such as a "Clear" command from a menu).

5.2 Pasting

```
(void) pgPaste (pg_ref pg, pg_ref paste_ref, long position, pg_boolean
text_only, short draw_mode);
```

The pgPaste function takes paste_ref (typically obtained from pgCut or pgCopy) and inserts all of its text into pg, beginning at text offset position (which is a byte offset). The paste_ref's contents remain unchanged.

The position parameter, however, can be CURRENT_POSITION (value of -1) in which case the paste occurs at the current insertion point in pg. After the paste the insertion point advances the number of characters that were inserted from paste_ref.

If text_only is "TRUE," only the text from paste_ref is inserted — no text formatting is transferred.

draw_mode can be the values as described in "Draw Modes".

```
draw_none,      // Do not draw at all
best_way,       // Use most efficient method(s)
direct_copy,    // Directly to screen, overwrite
direct_or,      // Directly to screen, "OR"
direct_xor,     // Directly to screen, "XOR"
bits_copy,      // Copy offscreen
bits_or,        // Copy offscreen, "OR" mode
bits_xor        // Copy offscreen, "XOR" mode
```

NOTES:

1. If there is already selected text in `pg` (the target `pg_ref`), it is deleted before the paste occurs.
2. Only text and styles are affected in the target `pg_ref` — shapes remain unchanged.

TECH NOTE: `pgPaste` custom styles

> I need to know when a custom style gets inserted into a particular `pg_ref`. That is, if a style is duplicated in an undo or clipboard context, I need to know when the style is inserted into the style table for the "real" `pg_ref`.

There are several ways to do this. Which method you choose depends on when you need to know, i.e. if you need to know the instant it occurs versus knowing somewhere in your app following a `pgPaste` or `pgUndo`.

By "instant it occurs" I mean when processing a style with one of the hooks, for instance. If that's what you need, one good way is to use the `duplicate` function. By mere virtue of getting called at all you know that OpenPaige is adding that style for one reason or another.

If you need to find out if that style exists at any arbitrary time, one way is to use `pgFindStyleInfo`. This function searches all style change(s) in the text to find the first occurrence of a particular style. One useful feature in `pgFindStyleInfo` is that you can set up a "mask" to only compare certain specific fields in your style. I assume your custom style will contain some kind of unique value for you to identify it, in which case this function is probably exactly what you want.

Then there is the "hack" method which looks dangerous, but isn't really. This method is to look at the whole style info list directly, which should remain compatible with all future OpenPaige versions *and* it is even portable between Windows and other platforms! This is done as follows:

```
paige_rec_ptr pg_rec;   // actual struct inside pg_ref
style_info_ptr styles;  // will be pointer to styles
long num_styles;        // will be number of styles avail

pg_rec = UseMemory(pg); // do this to get paige struct
num_styles = GetMemorySize(pg_rec->t_formats);  // number of style_info

styles = UseMemory(pg_rec->t_formats);  // points to first style
```

```
/* At this point: styles = pointer to first style_info and num_styles contains
number of styles. Hence, you can get next style as styles[1], ++styles, etc. To
find your particular style, just walk through and look for it. */

// Once you're through, you MUST do:

UnuseMemory(pg_rec->t_formats);
UnuseMemory(pg);
```

5.3 Copying Text Only

```
text_ref pgCopyText (pg_ref pg, select_pair_ptr selection, short data_type);
```

FUNCTION RESULT: This function returns a memory allocation containing a copy of the text in `pg`, beginning at the specified offset as follows: if selection is nonnull, it is used to determine the `selection` range (see "Copying and Deleting" for information about `select_pair` structure). If selection is a null pointer, the current selection range is used.

NOTE: The `memory_ref` returned from `pgCopyText` will have a "record size" set to one byte. In other words, a `GetMemorySize()` will return the number of bytes copied (which might be different to the number of characters, since OpenPaige can theoretically contain multibyte chars).

The `data_type` parameter specifies which type of text to copy which can be one of the following:

```
typedef enum
{
    all_data,               // Return all data
    all text chars,         // All text that is writing script
    all_roman,              // All Latin ASCII chars
    all visible data,       // Return all visible data
    all_visible_text_chars, // All visible text that is writing script
    all_visible_roman       // All visible Latin ASCII chars
};
```

If `data_type` is all_data, every byte in the specified range is copied; if `all_text_chars`, all single byte text is copied (which excludes only custom characters that aren't really "text"); for `all_roman` [sic!], only ASCII characters of Latin script are copied (as opposed to some other script such as

Chinese or Arabic).

The function result is typed as a text_ref which is a memory allocation created by the OpenPaige Allocation Manager.

NOTE: "Single byte text" in the above sense does not refer to single or double byte scripts such as Roman vs. Kanji. The `all_text_chars` data type will in fact include double-byte script. The only type excluded in this case is embedded graphics, controls, or some other customized text stream that really isn't text.

See also "Examine Text".

TECH NOTE: No zeros at the end of pgCopyText

I got my text in a `text_ref` with `pgcopyText`, but there is no 0 at the end!

1. Can I simply add a zero at the end to create a zero delimited string?
2. How do I know where the end is?

Point one: yes, but you *must* be careful since the `memory_ref` is only guaranteed to have allocated the number of bytes in the selection sent to `pgCopyText`. So if you want to append a zero, you should use `AppendMemory`, then put in the value.

```
memory_ref the_text;
the_text = pgCopyText(pg, &the_selection, all_data);

/* put a zero on the end so the parser doesn't walk off the end of the text */

AppendMemory(the_text, sizeof(pg_char), true);
UnuseMemory(the_text);
```

Point two: you can find the size of the text with `GetMemorySize()`, which will return the number of "records", which, in this case, will be the number of characters. Alternatively, you know the number of characters going into `pgCopyText` by knowing the selection range(s).

6 Undo and Redo

OpenPaige provides an variety of functions to fully support multi-kinds of "undo" for most situations. OpenPaige provides a convenient method of building custom undos which can be incorporated into your own application as well.

6.1 Concept of Undo

The concept of OpenPaige "undo" support is as follows: Before you do anything to an OpenPaige object that you want to be undoable, call `pgPrepareUndo` if you are about to do a `pgCut`, `pgDelete`, `pgPaste`, or any style, font or paragraph formatting change. The function result can then be given to `pgUndo` which will cause a reversal of what was performed.

For setting up an undo for `pgCut` or `pgDelete`, pass `undo_delete` for the verb parameter and a null pointer for `paste_ref`, for setting up an undo for `pgPaste`, pass `undo_paste` for the verb and the `pg_ref` you intend to paste from in `paste_ref`. For formatting changes (setting different fonts and styles or paragraph formats), pass `undo_format` for verb and null pointer for `paste_ref`.

6.2 Prepare Undo

To implement these features you must make the following function call prior to performing something that is undoable:

```
(undo_ref) pgPrepareUndo (pg_ref pg, short verb, void PG_FAR *insert_ref);
```

FUNCTION RESULT: This function returns a special memory allocation which you can give to `pgUndo` (below) to perform an Undo.

The `verb` parameter defines what you are about to perform, which can be one of the following:

```
typedef enum
{
    undo_none,              // Null undo ("can't undo")
    undo_typing,            // Undo key entry except bksp and forward delete
    undo_backspace,         // Undo backspace key
    undo_delete,            // Undo clear/cut/delete
    undo_fwd_delete,        // Undo forward delete
    undo_paste,             // Undo paste/insert
    undo_format,            // Undo text style, para format, or font
```

```
    undo_insert,            // Undo some other form of insertion
    undo_page_change,       // Undo page area change
    undo_vis_change,        // Undo vis area change
    undo_exclude_change,    // Undo exclusion area change
    undo_doc_info,          // Undo setDocInfo change
    undo_embed_insert,      // Undo embed_ref insertion
    undo_app_insert         // Undo insert with position parameter
};
```

"About to perform" means that you are about to do something you wish to be undoable later on. This includes performing a deletion, insertion, or text formatting change of any kind.

6.3 The `insert_ref` Parameter

For `undo_paste`, `insert_ref` must be the `pg_ref` you intend to paste (the source "scrap"); for `undo_insert`, `insert_ref` must be a pointer to the number of bytes to be inserted.

The `undo_app_insert verb` is identical to `undo_insert` except you must specify the insert location (`undo_insert` assumes the current text position). To do so, `insert_ref` must be a pointer to an array of two long words, the first element should be the text position to be inserted and the second element the insertion size, in bytes.

For `undo_typing`, `undo_backspace` and `undo_fwd_delete`, `insert_ref` should be the previous undo_ref you received for any `pgPrepareUndo` — or NULL if none.

NOTE: `insert_ref`, in this case, is an `undo_ref` — not a pointer to one — so you must coerce the `undo_ref` as `(void PG_FAR *)`.

For all other undo preparations, `insert_ref` should be NULL.

Insert 100 bytes

If you are about to insert, say, 100 bytes, you would call `pgPrepareUndo` as follows:

```
long length;
length = 100;
pgPrepareUndo(pg, undo_insert, (void PG_FAR *) &length);
```

```
/* The following function inserts a key into pg and returns the undo ref that
can be used to perform "Undo typing". The last_undo is the previous undo_ref,
or MEM_NULL if none. */

undo_ref insert_width_undo (pg_ref pg, pg_char the_key, undo_ref last_undo)
{
    undo_reffunction_result;

    if (the_key ⩾ ' ') // if control char
    {
        if(the_key == FWD_DELETE_CHAR)
            function_result = pgPrepareUndo(pg, undo_fwd_delete, (void PG_FAR
*) last_undo);
        else
            function_result = pgPrepareUndo(pg, undo_typing, (void *) undo);
    }
    else if (the_key == BACKSPACE_CHAR)
        function result = pgPrepareUndo(pg, undo backspace, (void *) undo);
        pgInsert(pg, (pg_char_ptr) &the_key, sizeof(pg_char), CURRENT_POSITION,
key_insert_mode, 0, best_way);
        return function_result;
}
```

For undo_paste, insert_ref must be the pg_ref you are about to paste (same as before).

For all other undo verbs, insert_ref is not used (so can be NULL).

6.4 Additional Undo verbs

undo_page_change can be used before changing the page shape, undo_vis_change before changing the visual area, and undo_exclude_change before changing the exclusion area.

The undo_doc_info verb can be given before changing anything in pg_ref's doc_info. For example, you could do “Undo Page Setup” with this undo verb.

The undo_embed_insert verb can be used before inserting an embed_ref (see chapter on Embedded Objects). Note, unlike undo_insert and undo_app_insert, the insert_ref parameter should be NULL for undo_embed_insert.

Undoing "Containers"

When you use `undo_page_change`, OpenPaige will set up an undo (and restore upon redo) both "container" rectangles and the associated refcons. You can therefore perform a full Undo Container Change.

6.5 Performing the Undo

To perform the actual Undo operation, pass an undo_ref to the following:

```
(undo_ref) pgUndo (pg_ref pg, undo_ref ref, pg_boolean requires_redo, short
draw_mode);
```

The `ref` parameter must be an `undo_ref` obtained from `pgPrepareUndo`.

If `requires_redo` is "TRUE," `pgUndo` returns a new `undo_ref` which can be used for a "Redo".

For example, if the `undo_ref` passed to this function performed an "Undo Cut," and `requires_redo` is given as TRUE, the function will return a new `undo_ref` which, if given to `pgUndo` again, would perform a "Redo Cut." Undo/Redo results can be toggled back and forth this way virtually forever.

`draw_mode` can be the values described in "Draw Modes":

```
draw_none,      // Do not draw at all
best way,       // Use most efficient method(s)
direct_copy,    // Directly to screen, overwrite
direct or,      // Directly to screen, "OR"
direct_xor,     // Directly to screen, "XOR"
bits_copy,      // Copy offscreen
bits_or,        // Copy offscreen, "OR" mode
bits_xor        // Copy offscreen, "XOR" mode
```

Generally, if you want the OpenPaige object to redraw, pass `best_way` for `draw_mode`.

NOTES

1. pgUndo returns a new undo_ref, which is a completely different allocation to the undo_ref you passed to it. It is your responsibility to dispose all undo_refs.
2. When an Undo is performed, it does not matter what the selection point (or selection range) is in pg at the time — pgUndo will restore whatever selection range(s) existed at the time the undo_ref was created. For example, if the user performs an action for which you created an undo_ref, such as a Paste, and then he selects some other text or clicks at a different location, pgUndo still works correctly, given that the original insertion point for the Paste is recorded in the undo_ref.

6.6 Disposing undo_refs

Once you are through using an undo_ref, dispose it by calling the following function:

```
(void) pgDisposeUndo (undo_ref ref);
```

The ref parameter must be a valid undo_ref (received from pgPrepareUndo or pgUndo); or, ref can be MEM_NULL (in which case pgDisposeUndo() does nothing).

NOTES

1. MEM_NULL is allowed intentionally, so that you can blindly pass your application's last "undo-able" operation that can be set initially to MEM_NULL.
2. There are a few cases where you should not dispose an undo_ref — see following.

Disposing the Previous Prepare-Undo

If you are implementing single-level undo support (user can only undo the last operation), you would normally need to dispose the "old" undo_ref (the one returned from the previous pgPrepareUndo()) before preparing for the next undo. For undo_typing, undo_fwd_delete, and undo_backspace, you must not dispose the "old" undo_ref — these are the lone exceptions to the "dispose-old-undo" rule.

The reason for this is that you give OpenPaige the "old" undo_ref as the insert_ref parameter; for undo_typing, undo_backspace, and undo_fwd_delete, the undo_ref given in insert_ref is

either disposed or returned back to you as the function result.

Never dispose the "previous" undo_ref when preparing for any of these "character" undos (undo_typing, undo_backspace and undo_fwd_delete). In all other cases, it is OK to dispose the previous undo_ref.

6.7 Undo Type

```
short pgUndoType (undo_ref ref);
```

This returns what type of undo_ref will perform.

FUNCTION RESULT: The function returns one of the undo verbs listed above under pgPrepareUndo, or a negative complement of a verb.

If the undo_ref is intended for a redo (returned from pgUndo, the verb will be its negative complement. For example, if pgUndoType() returns undo_paste, a call to pgUndo() would essentially perform a "Redo Paste".

A good use for this function is to set up a menu item for the user to indicate what can be undone.

NOTE

If you want to record more information about an Undo operation than the undo verbs listed above, use pgSetUndoRefCon, of which an explanation follows.

6.8 Undo RefCon

```
(void) pgSetUndoRefCon (undo_ref ref, long refCon);
(long) pgGetUndoRefCon (undo_ref ref);
```

These two functions allow you set (or get) a long reference inside an undo_ref.

The ref parameter must be a valid undo_ref; for pgSetUndoRefCon, refCon can be anything. pgGetUndoRefCon returns whatever has been set in ref.

6.9 Customizing undo

OpenPaige has a low-level hook for which you can use to implement modified undo actions, or you can completely customize an undo regardless of its complexity. See the chapter "Customizing OpenPaige" for more information.

6.10 Multilevel Undo

Your application can theoretically provide multiple-level Undo support by simply preparing a "stack" of undo_refs returned from pgPrepareUndo. Given that each undo_ref is independent of the next (i.e. there are no data structures within an undo_ref that depend on other undo_refs or even pg_refs), an application can keep as many of these around as desired to achieve "Undo of Undo" and "Undo of Undo of Undo," etc.

Supporting a multilevel Undo (being able to undo the last several operations) simply involves "stacking" the undo_refs returned from pgPrepareUndo.

CAUTION

When you set up for "Undo Typing" (be it for a regular insertion, backspace or forward delete), OpenPaige might return the same undo_ref that was given to pgPrepareUndo, and/or it might delete the previous undo_ref passed to the insert_ref parameter. In this case, make sure you check for this situation and handle it.

Example

```
/* The following code places consecutive undo_refs into an array so multi-level
"Undo" can be supported. While we only show stacking a maximum of 16, it can of
course be bigger. */
```

```
undo_ref stacked_refs[16];
short stack_index = 0;          // Begins with "no undos".

/* We call "PrepareUndo" from several places in the program. The verb is the
undo_verb to be performed. */

void PrepareUndo(pg_ref pg, short_verb)
{
    undo_ref new_undo, previous_undo;
    previous_undo = MEM_NULL;   // Assume no previous undo.
    if (verb == undo_typing || verb == undo_fwd_delete || verb ==
undo_backspace)
    if (stack index > 0)        // There is a previous undo.
    if (pgUndoType(stacked_refs[stack_index - 1] == verb))
        otherparam = stacked refs[stack index - 1];
        new_undo = pgPrepareUndo(pg, verb, (void PG_FAR *))
        previous_undo;

    // Check to see if OpenPaige returned the same undo_ref.

    if(!previous_undo || new_undo =≠ previous_undo_
        ++stack_index;

    stacked_refs[stack_index - 1] = new_undo;
}
```

CLIPBOARD SUPPORT

OpenPaige provides a certain degree of automatic support for the external clipboard, regardless of platform.

7.1 Writing to the Clipboard

```
void pgPutScrap(pg_ref the_scrap, pg_os_type native_format, short scrap_type);
```

This function writes the appropriate data to the external clipboard for other applications to read (including your own application). The data to be written is contained in the*scrap; usually, the*scrap would have been returned earlier from $p g \operatorname{Copy}($) or pgCut().

The scrap_type parameter indicates the preferred format within `pg` to write to the clipboard. If `scrap_type` is `pg_void_scrap` (value of zero), OpenPaige will write whatever format(s) are appropriate, including its own native type.

If `scrap_type` is non-zero it must be one of `pg_native_scrap` (the OpenPaige native format), `pg_text_scrap` (ASCII text), or `pg_embed_scrap` (the contents of an `embed_ref`).

For `pg_embed_scrap`, only `embed_mac_pict` (for Macintosh) and `embed_meta_file` (for Windows) are supported, and only the first `embed_ref` found within `the_scrap` is written to the clipboard.

The `native_format` parameter should contain a platform-appropriate identifier for a native OpenPaige format. For the Macintosh platform, `pg_os_type` is an `OSType` parameter; for the Windows platform, `pg_os_type` is a `WORD` parameter (Win16) or `int` parameter (Win32). Note that the value you place in `native_format` depends upon the runtime platform, as follows:

Windows only

You must first register a new format type by calling `RegisterClipboardFormat()`, then use that format type for every call to `pgPutScrap()` and `pgGetScrap()`. The name of this format type can be arbitrary; however, to remain consistent we recommend the name used by the custom control, "OpenPaige".

NOTES

1. **IMPORTANT!** You must call `OpenClipboard()` before calling `pgPutScrap()`, then call `CloseClipboard()` after this function has returned. OpenPaige can't open the clipboard for you because it can't assume there is a valid `HWND` available within its structure.
2. All data from the clipboard is copied, i.e. the data within the `pg_ref` is not owned by the clipboard.

Macintosh only

For Macintosh, a `pg_os_type` is identical to `OSType`. The name of this format type can be

arbitrary; however, to remain consistent we recommend the name used by the custom control, `paig`.

All Platforms

For both Macintosh and Windows platform, the clipboard is cleared before any data is written. If it is successful, the data can be read from the clipboard by calling pgGetScrap(), below.

7.2 Reading from the Clipboard

```
pg_ref pgGetScrap (pg_globals_ptr globals, pg_os_type native_format,
embed_callback def_embed_callback);
```

This function checks the external clipboard for a recognizable format and, if found, returns a new `pg_ref` containing the data; the `pg_ref` can then be passed to `pgPaste`. This function will work for both Macintosh and Windows-based applications.

The `globals` parameter must be a pointer to the OpenPaige globals structure (same structure used for `pgNew()`).

The `native_format` parameter should contain the same native format type identifier that was given to `pgPutScrap()`. For example, if running on a Macintosh, the `native_format` might be `paig`. On a Windows machine, `native_format` would be the value returned from `RegisterClipboardFormat()`.

The `def_embed_callback` parameter is an optional function pointer to an `embed_ref` callback function. The purpose of providing this parameter is to initialise any `embed_ref`s read from the clipboard to use your callback function. If `def_embed_callback` is NULL it will be ignored (and the default callback used by OpenPaige will be placed into any `embed_ref`s read).

NOTE (Windows)

IMPORTANT: You must call `OpenClipboard()` before calling `pgGetScrap()`, then call `CloseClipboard()` after you are through processing the data. OpenPaige can't open the clipboard for you because it can't assume there is a valid `HWND` available within its structure.

Function Result

If a format is recognized on the clipboard, a new `pg_ref` is returned containing the clipboard data. If no format(s) are recognized, `MEM_NULL` is returned.

NOTE

It is your responsibility to dispose the `pg_ref` returned from this function.

7.3 Format Type Priorities

Windows

OpenPaige will check the clipboard for format types it can support in the following priority order:

1. OpenPaige native format (taken from `native_format` parameter).
2. Text (`CF_TEXT`).
3. Metafile (`CF_METAFILEPICT`)
4. Bitmap (`CF_BITMAP`)

If none of the above formats are found, `pgGetScrap()` returns MEM_NULL.

Macintosh

OpenPaige will check the clipboard for format types it can support in the following priority order:

1. OpenPaige native format (taken from `native_format` parameter).
2. Text (`TEXT`).
3. Picture (`PICT`).

If none of the above formats are found, `pgGetScrap` returns MEM_NULL.

7.4 Checking Clipboard Availability

```
pg_boolean pgScrapAvail (pg_os_type native_format);
```

This function returns TRUE if there is a recognizable format in the clipboard. No data is read from the clipboard — only the data availability is returned.

The native_formatshould be the appropriate clipboard format type for the OpenPaige native format (see `pgPutScrap()` above).

This function is useful for controlling menu items, e.g. disabling "Paste" if nothing is in the clipboard.

NOTE (Windows)

IMPORTANT: You should call `OpenClipboard()` before calling `pgScrapAvail()`, then call `CloseClipboard()` after this function has returned.

8 STYLE BASICS

OpenPaige maintains three separate text formatting runs (series of text formatting changes): styles (bold, italic, super/subscript, etc.), fonts (Helvetica, Times, etc.) and paragraph formats (indentations, tabs, justification, etc.).

Each of these three formats can be changed separately; any portion of text can be a combination of each of these formats. Setting each of those is described in detail in “Advanced Styles". This chapter, *Style Basics*, describes the easiest, quickest, and simplest way to set the style, font and paragraph format you want.

NOTE

Unlike a Windows font that defines the whole composite format of text, the term *font* as used in this chapter generally refers only to a typeface, or typeface name. OpenPaige considers a *font* to simply be a specific family such as Candara, Consolas, Corbel, etc., while distinguishing other formatting properties such as bold, italic, underline, etc. as the text *style*.

8.1 Simplified Fonts and Styles

The simplest way to change the text in a `pg_ref` to different fonts, style or color is to use the high-level utility functions provided with OpenPaige version 3.0. These utilities provide a "wrapper" around the lower-level OpenPaige functions that change styles, fonts and text colors.

The source code to the wrapper has also been provided for your convenience, so you can alter them as necessary to fit your particular application. Or, you can examine them as reference material as the need occurs to apply more sophisticated stylization to your document.

Installing the Wrapper

All the functions listed in this section can be installed by including the source file `pgHLevel.c` in your project and `pgHLevel.h` as its header file. These functions can be called from both Macintosh and Windows platforms and should work with all compilers that support standard C conventions.

NOTE

If your application requires more sophistication than provided in this high-level wrapper, and/or if you cannot use the wrapper for any reason, please see the chapter, “Advanced Styles”.

8.2 Selection range

Most of the functions in this chapter require a selection range, `select_pairs` and `CURRENT_SELECTION`.

The selection range defines the range of text that should be changed, or if you pass a null pointer the current selection range (or insertion point) in `pg` is changed.

```
typedef struct
{
    long begin; // Beginning offset of some text portion
    long end    // Ending offset of some text portion
}
```

```
select pair
typedef select_pair PG_FAR *select_pair_ptr;
```

The begin field of a select_pair defines the beginning text offset and the end field defines the ending offset. Both offsets are byte offsets, not character offsets. Text offsets in OpenPaige are zero-indexed (i.e., the first offset is zero).

8.3 Changing / Getting Fonts

Windows prototype

```
#include "pgHLevel.h"
void pgSetFontByName (pg_ref pg, LPSTR font_name, select_pair_ptr
selection_range, pg_boolean redraw);
```

Macintosh prototype

```
#include "pgHLevel.h"
void pgSetFontByName (pg_ref pg, Str255 font_name, select_pair_ptr
selection_range, pg_boolean redraw);
```

This function changes the text in pg to the specified font_name.

If selection_range is a null pointer, the text in pg currently selected is changed (or, if nothing is selected, the font is applied to the next key insertion).

If selection_range is not null, it must point to a select_pair record defining the beginning and ending text offsets to apply the font. (See also "Selection range").

If redraw is TRUE the changed text is redrawn if there was a selected range affected.

NOTE

Only the font is affected in the composite style of the specified text, i.e. the text will retain its current point size and its other style attributes; only the font family changes.

Macintosh prototype

```
#include "pgHLevel.h"
pg_boolean pgGetFontByName (pg_ref pg, Str255 font_name);
```

Windows prototype

```
#include "pgHLevel.h"
pg_boolean pgGetFontByName (pg_ref pg, LPSTR font_name);
```

This function returns the font name that is applied to the text currently highlighted in `pg` (or, if nothing is highlighted, the font that applies to the current insertion point is returned).

The font name is returned in `font_name`. However, if the text is selected and the text range has more than one font, `pgGetFontByName` returns FALSE and `font_name` is not certain.

8.4 Setting/Getting Point Size

Prototype (same for both Mac and Windows)

```
#include "pgHLevel.h"
void pgSetPointSize (pg ref pg, short point_size, select_pair_ptr
selection_range, pg_boolean redraw);
```

This function changes the text point size to the new size specified.

If `selection_range` is a null pointer, the text in `pg` currently highlighted is changed (or, if nothing is highlighted, the point size is applied to the next key insertion).

If `selection_range` is not null, it must point to a `select_pair` record defining the beginning and

ending text offsets to apply the size. (See also "Selection range").

If `redraw` is TRUE the changed text is redrawn if there was a selected range affected.

NOTE

Only the text size is affected in the composite style of the specified text, i.e. the text will retain its current font family and its other style attributes; only the point size changes.

Prototype (same for both Mac and Windows)

```
#include "pgHLevel.h"
pg_boolean pgGetPointsize (pg_ref pg, short PG_FAR *point_size);
```

This function returns the point size that is applied to the text currently selected in $p g$ (or, if nothing is selected, the point size that applies to the current insertion point is returned).

The point size is returned in `*point_size` (which must not be a null pointer). However, if the text is highlighted and the text range has more than one size, `pgGetPointsize` returns FALSE and `*point_size` is not certain.

8.5 Setting / Getting Styles

Setting easy styles

Prototype (same for Mac and Windows)

```
#include "pgHLevel.h"
void pgSetStyleBits (pg_ref pg, long style_bits, long set_which_bits,
select_pair_ptr selection_range, pg_boolean_redraw);
```

This function changes the text style(s) to the new style(s) specified. "Styles" refers to text drawing

characteristics such as bold, italic, underline, etc.

The style(s) to apply are represented in `style_bits`, which can be a composite of any of the following values:

```
#include "pgHLevel.h"
#define X_PLAIN_TEXT            0x00000000
#define X_BOLD_BIT              0x00000001
#define X_ITALIC_BIT            0x00000002
#define X_UNDERLINE_BIT         0x00000004
#define X_OUTLINE_BIT           0x00000008
#define X_SHADOW_BIT            0x00000010
#define X_CONDENSE_BIT          0x00000020
#define X_EXTEND_BIT            0x00000040
#define X_DBL_UNDERLINE_BIT     0x00000080
#define X_WORD_UNDERLINE_BIT    0x00000100
#define X_DOTTED_UNDERLINE_BIT  0x00000200
#define X_HIDDEN_TEXT_BIT       0x00000400
#define X_STRIKEOUT_BIT         0x00000800
#define X_SUPERSCRIPT_BIT       0x00001000
#define X_SUBSCRIPT_BIT         0x00002000
#define X_ROTATION_BIT          0x00004000
#define X_ALL_CAPS_BIT          0x00008000
#define X_ALL_LOWER_BIT         0x00010000
#define X_SMALL_CAPS_BIT        0x00020000
#define X_OVERLINE_BIT          0x00040000
#define X_BOXED_BIT             0x00080000
#define X_RELATIVE_POINT_BIT    0x00100000
#define X_SUPERIMPOSE_BIT       0x00200000
#define X_ALL_STYLES            0xFFFFFFFF
```

The `set_which_bits` parameter specifies which of the styles specified in `style_bits` to actually apply; the value(s) you place in `set_which_bits` should simply be the bits (as defined above) that you want to change.

The purpose of `set_which_bits` is to distinguish between a style you choose to force to "off" versus a style you choose to remain unchanged.

For example, suppose you want to change all the selected text to boldface but leave the other styles of the text unchanged. To do so, you would simply pass `X_BOLD_BIT` in both `style_bits` and `set_which_bits`.

However, suppose you want to force the selected text to ONLY bold (forcing all other styles off). In

this case, you would pass X_BOLD_BIT in style_bits and 0xFFFFFFFF (or X_ALL_STYLES) in set_which_bits.

Also note for "plain" text (forcing all styles OFF), you pass X_PLAIN_TEXT for style_bits and X_ALL_STYLES for set_which_bits.

If selection_range is a null pointer, the text in pg currently selected is changed (or, if nothing is selected, the style(s) are applied to the next key insertion).

If selection_range is not null, it must point to a select_pair record defining the beginning and ending text offsets to apply the style(s). (See also "Selection range").

If redraw is TRUE the changed text is redrawn if there was a selected range affected.

NOTE

Only the specified style attributes will affect the text, i.e. the selected text will retain its font family and point size, and all other style attributes that are not specified in set*which*bits.

NOTE (Macintosh)

The first six style definition bits are identical to QuickDraw's style bits. You might find it convenient to simply pass the QuickDraw style(s) to this function.

Getting Style Example

```
#include "pgHLevel.h"

/* The following code sets the text currently selected in pg to bold-italic but
leaves all other styles in the text alone. The text gets re-draw with the
changes if we had a highlight range.*/

long style_bits = X_BOLD_BIT | X_ITALIC_BIT;
pgSetStyleBits(pg, style_bits, style_bits, NULL, TRUE);

/* The following code sets the text currently selected in pg to bold-italic but
does NOT leave the other styles alone (forces text to bold-italic and turns off
```

```
all other styles). The text gets re-drawn with the changes if we had a
highlight range. */

long style_bits = X_BOLD_BIT | X_ITALIC_BIT; pgSetStyleBits(pg, style_bits ,
X_ALL_STYLES, NULL, TRUE);

 // The following code changes all the selected text to "plain"

 pgSetStyleBits(pg, X_PLAIN_TEXT, X_ALL_STYLES, NULL, TRUE);
```

Prototype (both Mac and Windows)

```
#include "pgHLevel.h"
void pgGetStyleBits (pg_ref pg, long PG_FAR *style_bits, long PG_FAR
*consistent_bits);
```

This function returns the style(s) that are applied to the text currently highlighted in `pg` (or, if nothing is highlighted, the style(s) that apply to the current insertion point are returned).

The style(s) are returned in `*style_bits` (which must not be a null pointer); the value of `*style_bits` will be a composite of one or more of the style bits as defined in `pgSetStyleBits` (above).

The `*consistent_bits` parameter will also get set to the style(s) that remains consistent throughout the selected text; if a style bit in `consistent_bits` is set to a "1", that corresponding bit value in `*style_bits` is the same throughout the selected text.

For example, if `*style_bits` returns with all 0's, yet `*consistent_bits` is set to all 1's, the selection is purely "plain text" (no styles are set). However, if `*style_bits` returned all 0's but `*consistent_bits` was *not* all 1's, the text is not "plain text," rather the bits that are 0 in `*consistent_bits` reveal that style is not the same throughout the whole selection.

NOTE: The `consistent_styles` parameter must not be a null pointer.

8.6 Setting/Getting Text Color

Windows prototypes

```
#include "pgHLevel.h"
void pgSetTextColor (pg_ref pg, COLORREF color, select_pair_ptr
selection_range, pg_boolean redraw);
void pgSetBKColor (pg_ref pg, COLORREF color, select_pair_ptr selection_range,
pg_boolean redraw);
```

Macintosh prototypes

```
#include "pgHLevel.h"
void pgSetTextColor (pg_ref pg, RGBColor *color, select_pair_ptr
selection_range, pg_boolean redraw);
void pgSetTextBKColor (pg_ref pg, RGBColor *color, select_pair_ptr
selection_range, pg_boolean redraw);
```

`pgSetTextColor` changes the foreground color of text in `pg` to the specified color; `pgSetTextBKColor` changes the background color of text in `pg` to the specified color.

If `selection_range` is a null pointer, the text in `pg` currently highlighted is changed (or, if nothing is highlighted, the color is applied to the next key insertion).

If `selection_range` is not null, it must point to a `select_pair` record defining the beginning and ending text offsets to apply the color. (See also "Selection range").

If `redraw` is TRUE the changed text is redrawn if there was a selected range affected.

NOTE

Only the text color is affected in the specified text, i.e. the text will retain its current font family, point size and its other style attributes.

Windows prototypes

```
#include "pgHLevel.h"
pg_boolean pgGetTextColor (pg_ref pg, COLORREF PG_FAR *color);
pg_boolean pgGetTextBKColor (pg_ref pg, COLORREF PG_FAR *color);
```

Macintosh prototypes

```
#include "pgHLevel.h"
pg_boolean pgGetTextColor (pg_ref pg, RGBColor *color);
pg_boolean pgGetTextBKColor (pg_ref pg, RGBColor *color);
```

`pgGetTextColor` returns the foreground color that is applied to the text currently highlighted in `pg` (or, if nothing is highlighted, the color that applies to the current insertion point is returned); `pgGetTextBKColor` returns the text background color.

The color is returned in `*color` (which must not be a null pointer). However, if the text is highlighted and the text range has more than one size, the function returns FALSE and `*color` is not certain.

8.7 Style Examples

Setting styles (Windows)

```
/* The following code shows an example of setting a new point size, a new font
and new style(s) taken from a "LOGFONT" structure. All new text characteristics
are applied to the text currently highlighted (or they are applied to the NEXT
pgInsert if no text is highlighted). Carefully note that we do not "redraw" the
text until the last function is called, otherwise we would keep "flashing" the
refresh of the text. */

#include "Paige. h"
#include "pgUtils.h"
#include "pgHLevel.h"

LOGFONT log_font;   // got this from "ChooseFont" or whatever
```

```
long style_bits, set_bits;  // used for pgSetStyleBits

// Set font (by name)
pg SetFontByName(pg_log_font.lfFaceName, NULL, FALSE);

// Set point size
pg SetPointSize(pg, pgAbsoluteValue((long)log_font.lfHeight, NULL, FALSE);

// Set style attributes:
style_bits = set_bits = 0;
if (log_font.lfWeight == FW_BOLD)
    style_bits |= X_BOLD_BIT;
if (log_font.lfItalic)
    style bits |= X ITALIC_BIT;
if (log_font.lfUnderline)
    style bits |= X UNDERLINE_BIT;
if (log_font.lfStrikeOut)
    style_bits |= X_STRIKEOUT_BIT;

// Before setting the styles, check if we actually have "plain text":
if (style_bits == X_PLAIN_TEXT)
    set_bits = X_ALL_STYLES;
else
    set_bits = style_bits;

// Note, this time we pass "TRUE" for redraw because we are done:
pgSetStyleBits(pg, style_bits, set_bits, NULL, TRUE);
```

Handling font menu (Macintosh)

```
#include "pgHLevel.h"
/* The following code assumes a "Font" menu (which lists all available fonts),
a "Style" menu (containing Plain, Bold, etc.) and a "Point" menu (with 9, 12,
18 and 24 point values). Each example assumes its respective menu has been
selected by user and "menu_item" is the item selected. */

/* For font menu: */
Str255 font;
GetItem(FontMenu, menu_item, font);
pgSetFontByName(pg, font, NULL, TRUE);
```

```
/* For style menu: */
long style_bits, set_bits;

switch (menu_item)
    {
    case PLAIN_ITEM:
        style_bits = X_PLAIN_TEXT;
        set_bits = X_ALL_STYLES;
        break;
    case BOLD_ITEM:
        style_bits = set_bits = X_BOLD_BIT;
        break;
    case ITALIC_ITEM:
        style bits = set_bits = X_ITALIC_BIT;
        break;
    case UNDERLINE ITEM:
        style_bits = set_bits = X_UNDERLINE_BIT;
        break;
    case OUTLINE_ITEM:
        style_bits = set_bits = X_OUTLINE_BIT;
        break;
    case SHADOW_ITEM:
        style_bits = set_bits = X_SHADOW_BIT;
        break;
    }

pgSetStyleBits(pg, style_bits, set_bits, NULL, TRUE);

// Setting point size

short pointsize;
switch (menu_item)
    {
    case PT9_ITEM:
        pointsize = 9;
        break;
    case PT12_ITEM:
        pointsize = 12;
        break;
    case PT18_ITEM:
        pointsize = 18;
        break;
    case PT24_ITEM:
        pointsize = 24;
        break;
```

8.8 Changing pg_ref style defaults

Changing the defaults of the `pg_ref` is done just after `pgInit`. Changing the defaults is shown in “A Different Default Font, Style, Paragraph”.

8.9 Changing Paragraph Formats

Changing the paragraph format applied to text range(s) requires a separate function call since paragraph formats are maintained separate from text styles and fonts.

To set one or more paragraphs to a different format, call the following:

```
(void) pgSetParInfo (pg_ref pg, select_pair_ptr selection, par_info_ptr info,
par_info_ptr mask, short draw_mode);
```

This function is almost identical to `pgSetStyleInfo` or `pgSetFontInfo` except a `par_info` record is used for `info` and `mask`.

The other difference is that `pgSetParInfo` will always apply to at least one paragraph: even if the selection "range" is a single insertion point, the whole paragraph that contains the insertion point is affected.

The `selection` and `draw_mode` parameters are functionally identical to the same parameters in `pgSetStyleInfo` (see "Changing Styles" and "Draw Modes"), except whole paragraphs are changed (even if you specify text offsets that do not fall on paragraph boundaries). (See also "Selection range" and "All About Selection”).

For detailed information on `par_info` records—and what fields you should set up—see "par_info".

NOTE

If you want to set or change tabs, it is more efficient (and less code) to use the functions in the chapter "Tabs & Indents".

```
(long) pgGetParInfo (pg_ref pg, select_pair_ptr selection, pg_boolean
set_any_match, par_info_ptrinfo, par_info_ptr mask);
```

This function returns paragraph information for a specific range of text.

If `selection` is a null pointer, the information that is returned applies to the current selection range in `pg` (or the current insertion point); if selection is non-null, pointing to `select_pair` record, information is returned that applies to that selection range (see "Copying and Deleting" for information about `select_pair` pointer under `pgGetStyleInfo`).

Both `info` and `mask` must both point to `par_info` records; neither can be a null pointer. When the function returns, both `info` and `mask` will be filled with information you can examine to determine what style(s), paragraph format(s), or font(s) exist throughout the selected text, and/or which do not.

If `set_any_mask` was FALSE, all the fields in `mask` that are set to nonzero indicate that the corresponding field value in `info` is the same throughout the selected text; all the fields in `mask` that are set to zero indicate that the corresponding field value in `info` is not the same throughout the selected text.

For example, suppose after calling `pgGetParInfo`, `mask.spacing` has a nonzero value. That means that whatever value has been set in `info.spacing` is the same for every paragraph in the selected text. Hence, if `info.spacing` is 12, then every character is spaced the same.

On the other hand, suppose after calling `pgGetParInfo`, `mask.spacing` is set to zero. That means that some of the characters in the selected text match the spacing in `info` and some do not. In this case, whatever value happens to be in `info.spacing` is not certain.

Essentially, any nonzero in `mask` is saying, "Whatever is in `info` for this field is applied to every character in the text," and any zero in `mask` is saying, "Whatever is in `info` for this field does not matter because it is not the same for every character in the text."

You want to pass FALSE for `set_any_mask` to find out what paragraph formats apply to the entire selection (or not).

TABLE #3: POSSIBLE RESULTS WHEN `SET_ANY_MASK` IS SET TO `FALSE`

info	mask	results
12	-1	All paragraphs have spacing of 12
12	0	Some paragraphs have spacing of 12

Setting `set_any_match` to TRUE is used to determine if only a part of the text matches a given

paragraph format. This is described in "Obtaining Current Text Format(s)". The par_info structure is described in "par_info".

9 TABS & INDENTS

9.1 Tab Support

One of the elements of a paragraph formats is a list of tab stops. Although you could set tabs (or change tabs) using pgSetParInfo, some additional functions have been provided exclusively for tabs to help save on coding:

```
void pqSetTab (pq_ref pg, select_pair_ptr selection, tab_stop_ptr tab_ptr,
short draw_mode);
```

This function sets a new tab that applies to the specified selection.

The selection parameter is used in the same way as other functions use a select_pair parameter: if it is a null pointer, the current selection in pg is used, otherwise the selection is taken from the parameter (for information about pgSetParInfo regarding select_pair records, see "Selection range").

The draw_mode is also identical to all other functions that accept a draw_mode parameter. draw_mode can be any of the values described in "Draw Modes":

```
draw_none,      // Do not draw at all
best_way,       // Use most efficient method(s)
direct_copy,    // Directly to screen, overwrite
direct_or,      // Directly to screen, "OR"
direct_xor,     // Directly to screen, "XOR"
bits_copy,      // Copy offscreen
bits_or,        // Copy offscreen, "OR" mode
bits_xor        // Copy offscreen, "XOR" mode
```

The tab_ptr parameter is a pointer to the following record (tab must not be a null pointer):

```
typedef struct
```

```
{
    long tab_type;  // Type of tab
    long position;  // Tab position
    long leader;    // Tab leader (or null)
    long ref_con;   // Can be used for anything
}
```

The tab_type field can be one of the following:

```
typedef enum
{
    no_tab,     // none (used to delete)
    left_tab,       // left tab
    center_tab, // centre tab
    right tab,  // right tab
    decimal_tab // tab on decimal point
}
```

The position field in a tab_stop defines the tab's position, in pixels. However, a tab's pixel position is relative to either the left edge of pg's page_area, or to the left edge of the window (see "Tab Base").

If leader is nonzero, the tab is drawn with that value as a "leader" character. OpenPaige assumes that the character has simply been coerced to a numeric value, which will therefore imply whether the leader character is a single ASCII byte (leader < 256), or a double byte (leader > 256).

For example, if the leader is a single ASCII byte for a "." (hexadecimal 2E), the value placed in leader should be 0x0000002E. If leader is a double-byte character, such as the Kanji with hexadecimal value 802E, then the leader value should be set to 0x0000802E, etc.

```
my_tab.leader = '-';
```

A leader is the character placed before a tab, like this

```
01234  5  6789
ABC -[TAB]DEFG
```

The ref_con field can be used for anything.

Deleting a Tab

You can delete a tab by calling `pgSetTab` with a tab record of type `no_tab` where the position field set to the exact position of the existing tab you wish to delete.

Changing a Tab

If you want to change a tab's position (location relative to the tab base), you must delete the tab and add a new one (see previous, "Deleting a Tab").

If you want to change anything else (such as the tab type or leader), simply call `pgSetTab` with a tab record whose position is identical to the one you wish to change.

NOTES:

1. The maximum number of tab settings per paragraph is 32.
2. Tab settings affect whole paragraphs. They are in fact part of the paragraph formatting.

TECH NOTE: Tabs setting different for different lines

> I am displaying information in Paige with each block of info occupying 2 lines of text. I would like to have tab stops set differently for the first and second line.

It depends on what you mean by "line."

If each line ends with a CR (carriage return), OpenPaige considers each one a "paragraph" and thus you can simply change the paragraph formatting to be different for each line.

However, if both of your lines are one continuous string of text that just word-wraps into two lines, it is virtually impossible to apply two different sets of tab stops.

This is because tabs are, by definition, a paragraph format and a paragraph is simply text that ends with a CR, no matter how many lines it might have.

I will assume you have CR-terminated lines ("paragraphs"). To apply different tab stops to the

second line, you need to simply use the tab setting function(s) as given in the manual. of course you need to know at least one of the text positions in the line you need to change (for example, you need to know that line number 2 starts at the 60th character, or the $72 \mathrm{nd}$ character, etc.); you also need to insert the text line first before you can apply the tab-stop changes (unlike text styles, paragraph styles require that you have a "paragraph" for which to apply the style change).

9.2 Changing / Getting Multiple Tabs

Get Tab List

This provides a way to look at all the tabs within a section of text:

```
(void) pgGetTabList (pg_ref pg, select_pair_ptr selection, tab_ref tabs,
memory_ref tab_mask, long PG_FAR *screen_offset);
```

The `selection` parameter operates in the same way it does for `pgGetParInfo` (see "Obtaining Current Text Format(s)" for information about `pgGetStyleInfo` and `pgGetParInfo`).

The `tabs` and `tab_mask` parameters for `pgGetTabList` are memory allocations which you must create before calling this function. When the function returns, `tabs` will be set to contain an array of `tab_stop` records that apply to the selection range and `tab_mask` will be set to contain an array of `long`s containing non-zeros for every tab that is consistent (the same) throughout the selection.

For example, supposing that the specified selection contained 3 tabs, when `pgGetTabList` returns, `tabs` would contain all three `tab_stop` records and `tab_mask` would contain 3 long words (each corresponding to the `tab` in `tabs`). If the corresponding long word in `tab_mask` is zero, that tab is inconsistent (not the same) and/or does not exist throughout the entire selection range.

The `tab_mask`, however, can be a `MEM_NULL` if you don't require a "consistency report." The `tabs` parameter, however, must be a valid `memory_ref`.

The `screen_offset` parameter should either be a pointer to a `long` or a null pointer. When the function returns, the variable pointed to by `screen_offset` will get set to the tab base value (the position, in pixels, against which tabs are measured—see "Tab Base"). If `screen_offset` is a null pointer, it is ignored.

NOTES

1. To learn how to create the allocations passed to tabs and `tab_mask`, and how to access their contents, see "The Allocation Mgr".
2. Calling this function forces the tabs memory allocation to contain `sizeof(tab_stop)` record sizes. Hence, the result of `GetMemorySize(tabs)` will return the number of `tab_stop` records. Similarly, the `tab_mask` is forced to a record size of `sizeof(long)`, so `GetMemorySize(tab_mask)` will return the same number.
3. If no tabs exist at all, `pgGetTabList` will set your tabs and `tab_mask` allocation to a size of zero.

Set Tab List

```
(void) pgSetTabList (pg_ref pg, select_pair_ptr selection, tab_ref tabs,
memory_ref tab_mask, short draw_mode);
```

The above function provides a way to apply multiple tabs all at once to a specified `selection`.

The `selection` parameter operates the same as all functions that accept a `select_pair`.

`draw_mode` can be the values as described in "Draw Modes":

```
draw_none,      // Do not draw at all
best_way,       // Use most efficient method(s)
direct_copy,    // Directly to screen, overwrite
direct_or,      // Directly to screen, "OR"
direct_xor,     // Directly to screen, "XOR"
bits_copy,      // Copy offscreen
bits_or,        // Copy offscreen, "OR" mode
bits_xor        // Copy offscreen, "XOR" mode
```

The `tabs` and `tab_mask` parameters must be memory allocations that you create. The `tabs` allocation must contain one or more tab stop records; the `tab_mask` allocation must have an identical number of long words, each `long` corresponding to the tab element in `tabs`. For every entry in `tab_mask` that is nonzero, that corresponding tab is applied to the selection range; for every `tab_mask` entry that is zero, that tab is ignored.

For example, if you set up the tabs allocation to contain 3 `tab_stop` records, and the `tabs_mask`

had three `long`s of 1, 0, 1, then the first and third tab would be applied to the selection range; the second tab would not be applied.

However, `tab_mask` can be `MEM_NULL` if you simply want to set all tabs unconditionally.

NOTES

1. To learn how to create the allocations passed to `tabs` and `tab_mask`, and how to access their contents, see "The Allocation Mgr".
2. The maximum number of `tab_stop`s applied to one paragraph is 32.
3. When setting multiple tabs, any current tab settings are maintained — they do not get "deleted". However, a `tab_stop` does get *replaced* if a new tab contains the same exact position.

9.3 Tab Base

Tab positions (the pixel positions specified in the position field of a tab_stop record) are considered relative to some other position and not absolute. OpenPaige supports three "tab base" values defining the relative position for which to place tabs. If the base value is positive or zero, OpenPaige uses that value as the tab base. If the base value is negative, the tab base implies one of the following:

```
#define TAB_BOUNDS_RELATIVE -1  // relative to page_area bounds
#define TAB_WRAP_RELATIVE -2    // relative to current line wrap edge
```

The difference between `TAB_BOUNDS_RELATIVE` and `TAB_WRAP_RELATIVE` depends on what kind of wrap shape (`page_area`) that exists in the OpenPaige object. `TAB_BOUNDS_RELATIVE` means tabs are always relative to the entire bounding area (enclosing rectangle) of the `page_area`, regardless of the shape, while `TAB_WRAP_RELATIVE` measures tabs against the leftmost edge of the specific portion of the text line for which the tab is intended.

Setting/Changing Tab Base

```
(void) pgSetTabBase (pg_ref pg, long tab_base);
```

```
(long) pgGetTabBase (pg_ref pg);
```

To set (or change) the tab base, call `pgSetTabBase` and provide the base value in `tab_base`, which can be a positive number or zero (in which case, tabs are relative to that pixel position), or a negative number (either `TAB_BASE_RELATIVE` or `TAB_BOUNDS_RELATIVE`).

To get the current tab base, call `pgGetTabBase` and the base currently used by `pg` will be the function result.

NOTE: The default tab base in a new `pg_ref` is zero (tabs are relative to pixel position 0).

The four illustrations to follow show examples of how tab positions are measured against the tab base value (the tab base value is stored in `pg_ref` and can be changed with the functions shown above).

Figure 4 ("TAB BASE = 0") shows a tab measurement with a tab base of zero, while Figure 5 ("TAB BASE = 16") shows a tab base of 16, in which case all tabs are relative to 16 pixels from the left of the window. In both cases, the window's left origin is assumed to be at coördinates (0, 0).

Figures 6 ("TAB BASE = `TAB_BOUNDS_RELATIVE`") and 7 ("TAB BASE = `TAB_WRAP_RELATIVE`") both measure tabs against the left side of `page_area`, except that, where a line of text exists, `TAB_WRAP_RELATIVE` is measured against the edge of `page_area`. If `page_area` is *a single rectangle*, both of the latter two tab base modes are *identical*.

Figures 4 – 7

The following are some illustrations of different tab base values:

FIGURE #4 **TAB BASE = 0**

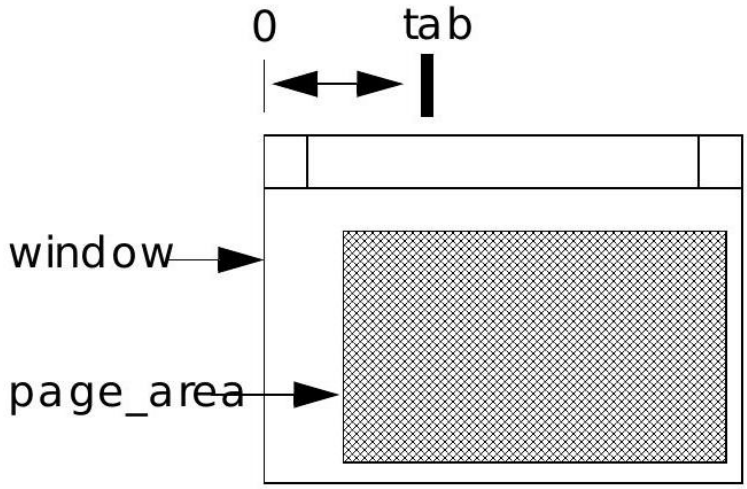


FIGURE #5 **TAB BASE = 16**

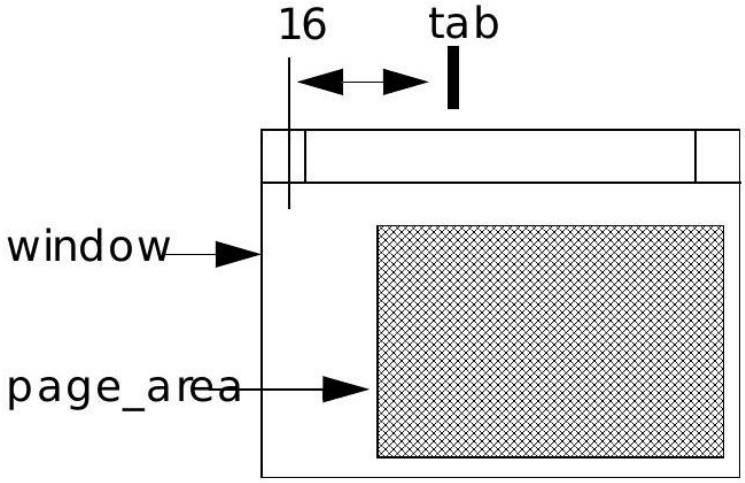


FIGURE #6 **TAB BASE = TAB_BOUNDS_RELATIVE**

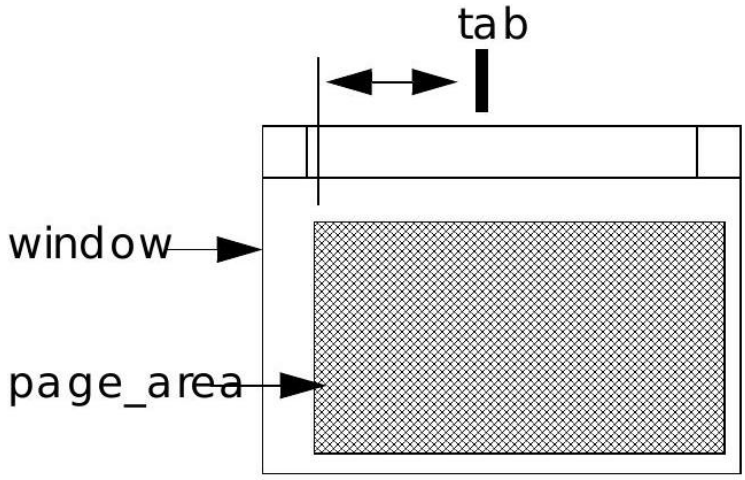
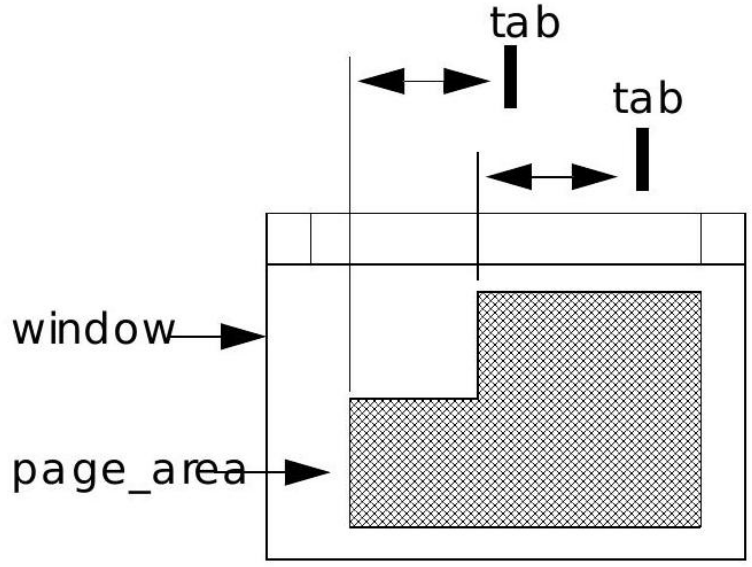


FIGURE #7 **TAB BASE = TAB_WRAP_RELATIVE**



9.4 Indentation Support

Set Indents

One of the elements of a paragraph format is a set of paragraph indentations (left, right, and first-line indents). Although you could set these using pgSetParInfo, some additional functions have been provided exclusively for indents to help save on coding:

```
void pgSetIndents (pg_ref pg, select_pair_ptr selection, pg_indents_ptr
indents, pg_indents_ptr mask, short draw_mode);
```

The function above changes the indentations for the text range specified.

The selection parameter operates in the same way it does for pgGetParInfo (see "Selection range" for information about selection ranges and "Changing Styles" about pgSetStyleInfo and pgSetParInfo).

draw_mode can be the values as described in "Draw Modes" on page 2-30:

```
typedef enum
{
    draw_none,          // Do not draw at all
    best_way,           // Use most efficient method(s)
    direct copy,        // Directly to screen, overwrite
    direct_or,          // Directly to screen, "OR"
    direct xor,         // Directly to screen, "XOR"
    bits_copy,          // Copy offscreen
    bits or,            // Copy offscreen in "OR" mode
    bits_xor,           // Copy offscreen in "XOR" mode
    bits_emulate_copy   // Copy "fake" offscreen
    bits_emulate_or // "Fake" offscreen in "OR" mode
    bits_emulate_xor    // "Fake" offscreen in "XOR" mode
};
```

The indents and mask parameter must point to the following structure (neither pointer can be null):

```
typedef struct
{
    long left_indent;   // Left margin (indent)
    long right_indent;  // Right margin (indent)
    long first_indent;  // First-line indent
```

```
}
pg_indents, PG_FAR *pg_indents_ptr;
```

The `mask` parameter should contain nonzero fields for every indent you wish to change in indents.

NOTE: "nonzero" means that you should fill the field with -1 (so all bits are set to ones).

Indentations are pixel positions relative to a text line's maximum left and maximum right, as follows: the `left_indent` is the distance from the leftmost edge of a line (which will be the `page_area`'s left edge for that line); the right indent is the distance from the rightmost edge (which will be the `page_area`'s right edge). Note that this is a positive number, not a negative inset. The `first_line_indent` is relative to the `left_indent`. Note that *only* the `first_line_indent` should ever be negative (in which case the first line of the paragraph hangs to the left of the left indent).

When indents are changed, they apply to whole paragraphs.

Get Indents

To obtain the current indent settings of a selection range, call the following:

```
(void) pgGetIndents (pg_ref pg, select_pair_ptr selection, pg_indents_ptr
indents, pg_indents_ptr_mask, long PG_FAR *left_screen_offset, long PG_FAR
*right_screen_offset);
```

The selection parameter operates in the same way it does for `pgGetParInfo` (see “Obtaining Current Text Format(s)" for information about `pgGetStyleInfo` and `pgGetParInfo`).

The indents and mask parameters should point to a `pg_indents` record (described above); neither parameter can be a null pointer.

FUNCTION RESULT: When this function returns, indents will be set to the indentation values found in the selection range, and mask will have every field that is consistent (the same) throughout the range to nonzero.

If `left_screen_offset` and `right_screen_offset` are non-null, `pgGetIndents` will set the variables to which they point to the relative left position and right position, respectively, against which the indents are measured. The usual reason you will need to have this information is to draw a "ruler" showing indents, in which case you will need to know the relative edges to draw each indentation. This is particularly important if your `page_area` is non-rectangular (because

the relative edges can change from line to line).

NOTE: The `left_screen_offset` and `right_screen_offset` values will include the scrolled position of the OpenPaige object, if any (see chapter 11, "All About Scrolling").

10 All About Selection

An OpenPaige object's text can be selected either by the user or directly by your application.

10.1 Up & Running with Selections

Selection by the user is accomplished with `pgDragSelect`; this has already been covered in detail (see "Blinking Carets & Mouse Selections" with regards to `pgDragSelect`).

Additional support functions are provided, however, to set selections directly and/or to obtain both simple selections (insertion points or a selection pair of offsets) as well as complex selections (discontinuous selections).

10.2 Simple Selections

A "simple" selection is either a single insertion point or a pair of text offsets which implies a single range. This includes vertical selections that contain only two points (topleft and bottom-right text positions). To set a simple selection, call the following:

```
(void) pgSetSelection (pg_ref pg, long begin_sel, long end_sel, short
modifiers, pg_boolean show_hilite);
```

The selection range in `pg` will be set to `begin_sel` to `end_sel`, which are byte offsets; the lowest offset is zero and the highest offset is `pgTextSize(pg)`. If `begin_sel` is the same as `end_sel`, a single insertion is implied.

The modifiers parameter is identical to the modifiers passed to `pgDragSelect` (see “Blinking Carets & Mouse Selections" for a list of bits you can pass for modifiers). This parameter controls how the text is selected, i.e., extended selection, vertical selection, word selection, etc.

If should_draw is TRUE, a new highlight region is computed and drawn. If should_draw is FALSE, nothing on the screen changes (but pg will internally change its selection).

NOTE: If you want to select all text, pass an arbitrary-but-huge number for end_sel. OpenPaige will adjust large numbers to be equal to the current text size.

To obtain the current selection (assuming it is a simple selection), call the following:

```
(void) pgGetSelection (pg_ref pg, long PG_FAR *begin_sel, long PG_FAR
*end_sel);
```

The current selection range is returned in *begin_sel and *end_sel. Either parameter can be a null pointer if you don't want the result.

If the selection range is discontinuous, you will receive the first selection pair.

NOTE: pgSetSelection will not affect the style of text. It merely highlights the text and gets the internal range within OpenPaige so that other functions can operate thereon.

10.3 Discontinuous Selections

A discontinuous selection can be accomplished with pgDragSelect and setting the appropriate bit in the modifiers parameter (in which case, every new verb of mouse_down will start a new selection pair). You can also accomplish this from your app with multiple pgSetSelection calls and the appropriate bit set in modifiers.

To set a discontinuous selection from your app all at once, however, you can use the following:

```
(void) pgSetSelectionList (pg_ref pg, memory_ref select_list, long
extra_offset, pg_boolean show_hilight);
```

The select_list parameter must be a memory allocation containing one or more xselect_pair records (see "Selection range" for information about select_pair).

The offset_extra parameter is an amount to add to each selection pair within select_list; if you want to apply the select_list as-is, pass zero for extra_offset.

If should_draw is TRUE, the new selection is drawn.

See "The Allocation Mgr" regarding memory allocations.

To obtain the current discontinuous selection, call the following:

```
(memory_ref) pgGetSelectionList (pg_ref pg, pg_boolean for_paragraph);
```

FUNCTION RESULT: This function returns a newly created memory allocation containing one or more `select_pair` records which represent the entire selection in `pg`.

If `for_paragraph` is TRUE, the selection pairs will be paragraph-aligned; otherwise, they will be character-aligned (if you want to know what paragraphs fall in the selection range(s), the distinction *must* be made).

CAUTION: If there is no selection range, e.g. only a caret, and for_paragraph is FALSE, this function will return `MEM_NULL` (zero).

You will know how many `select_pair` records are contained in the function result by calling `GetMemorySize()` on the function result—see "The Allocation Mgr".

NOTE: It is your responsibility to dispose the memory allocation returned from this function.

10.4 Additional Selection Support

Extending the selection

```
(void) pgExtendSelection (pg_ref pg, long amount_ext, short modifiers,
pg_boolean show_hilite);
```

FUNCTION RESULT: The above function extends the current selection by `amount_ext`; the new extension follows the attributes in `modifiers` if appropriate (for example, the selection could be extended by whole words or paragraphs).

Negative values in `amount_ext` extend to the left (extend the beginning selection backwards); positive numbers extend to the right (extend the ending selection forwards).

The `modifiers` can generally be a combination of:

```
#define EXTEND_MOD_BIT      0x0001  // Extend the selection
#define WORD_MOD_BIT        0x0002 // Select whole words only
#define PAR_MOD_BIT         0x0004 // Select whole paragraphs only
#define LINE_MOD_BIT        0x0008  // Select whole lines only
#define DIS_MOD_BIT         0x0020  // Enable discontiguous selection
#define STYLE_MOD_BIT       0x0040 // Select whole style range
#define WORD_CTL_MOD_BIT    0x0080 // Select "words" delimited by control chars
#define NO_HALF_CHARS_BIT   0x0100 // Click does not go left/right on half-
chars
```

These are explained in the section "Modifiers". Vertical selection cannot be extended using the modifiers. Using that modifier in combination with the others will cause unpredictable results.

If `show_hilite` is TRUE, the new highlight is drawn; if FALSE, the appearance does not change.

NOTE: If the current selection is discontinuous, only the last (ending) selection pair is affected by this function.

Handling mouse & key combinations for selection (Mac)

NOTE: This code does not handle shift-clicks and option-clicks in the same way as the demo. The point of this code is that you can change the key combinations for your own uses. Consult the demo for other ways of handling this.

```
#include "Paige.h"

#define LEFT_ARROW      0x1C
#define RIGHT_ARROW     0x1D
#define UP_ARROW        0x1E
#define DOWN_ARROW      0x1F
#define BACKSPACE_CHAR  0x08
#define RETURN_CHAR     0x0D
#define ENTER_CHAR      0x03
#define TAB_CHAR        0x09
#define LF_CHAR         0x0A
#define HOME_KEY        0x01
#define END_KEY         0x04
```

```
static int scroll to cursor(pg ref my pg);
static int key_doc_proc(EventRecord *event);
static int is an arrow(char key);
extern pg_globals paige_rsrv;
extern undo_ref last_undol

// This is the keydown proc

static int key_doc_proc(pg_ref my_pg, EventRecord *event)
{
    char the_key;
    short modifiers;
    pg_ref my_pg;

    the_key = event → message & charCodeMask;
}
```

Next we parse the event record. We have the record before going into `pgInsert` and can change the keys around or do other things before we send the key into the `pg_ref`. In this case, we intercept the `HOME_KEY` and the `END_KEY` and scroll the `pg_ref` to the top and bottom:

```
if (the_key == HOME_KEY)
{
    pgScroll(my_pg, scroll_home, scroll_home, best_way);
    UpdateScrollBarValues(my_pg);
}
else
if (the_key == END_KEY)
{
    pgScroll(my_pg, scroll_none, scroll_end, best_way);
    UpdateScrollBarValues(my_pg);
}
else
{
    ObscureCursor();
}
```

Then we check to see if they are characters that OpenPaige would normally handle and if so, we insert them into the `pg_ref`. When `pgInsert` contains the `key_insert_mode` or `key_buffer_mode` in the `insert_mode` parameter, it responds as we would expect when arrow keys are entered, i.e., by moving the insertion point, by handling backspace, by deleting previous characters, etc.

We don't need to use `pgExtendSelection`.

OpenPaige automatically handles extending the selection by holding down the shift key while using arrow keys if the `EXTEND_MOD_BIT` is set during `pgInsert`. `key_buffer_mode` will keep calling the events as long as OpenPaige is receiving keystrokes, making keyboard text insertion very fast. OpenPaige won't cycle through the event loop until the keystrokes are paused.

```
// Here are the modifiers changing the selection
modifiers = 0;
if (event → modifiers & shiftKey)
    modifiers ⊨ EXTEND_MOD_BIT;
if (event → modifiers & optionKey)
    modifiers ⊨ WORD_MOD_BIT;

if (the_key == ENTER_CHAR)
{
    event → message = LF_CHAR;
    the_key = LF_CHAR;
}

if (the key ⩾ ' ' || the key < 0 || the key == TAB CHAR || the key ==
RETURN_CHAR || the_key == LF_CHAR || the_key == BACKSPACE_CHAR ||
is_an_arrow(the_key))
{
    short verb_for_undo;
    DisposeUndo(my_pg, last_undo);
    if (the_char == paige_rsrv.bs_char)
        verb_for_undo = undo_backspace;
    else
        verb_for_undo = undo_typing;
    last_undo = pgPrepareUndo(my_pg, verb_for_undo, (void PG_FAR*) last_undo);

    pgInsert(my_pg, (pg_char_ptr &the_key, sizeof(pg_char), CURRENT_POSITION,
key_insert_mode, 0, best_way);

    if (the key == BACKSPACE CHAR)
        pgAdjustScrollMax(my_pg, best_way);

    scroll_to_cursor(my_pg);
}

return FALSE;   // to be returned
```

Number of selections

```
(pg_short_t) pgNumSelections (pg_ref pg);
```

This returns the number of selection pairs in `pg`. A result of zero implies a single insertion point; a result of one implies a simple selection, and likewise for higher numbers.

Caret & Cursor

```
(pg_boolean) pgCaretPosition (pg_ref pg, long offset, rectangle_ptr
caret_rect);
```

FUNCTION RESULT: This returns a rectangle in `caret_rect` representing the "caret" corresponding to `offset`. If `offset` equals `CURRENT_POSITION` (value of 1), the current insertion point is used. If the current selection in `pg` is in fact a single insertion, the function returns TRUE; if it is not, `caret_rect` gets set to the top-left edge of the selection and the function returns FALSE.

NOTE: If you specify some other position besides `CURRENT_POSITION`, the function will always return TRUE because you have explicitly implied a single insertion point.

```
(void) pgSetCursorState (pg_ref pg, short cursor_state); (short)
pgGetCursorState (pg_ref pg);
```

These two functions let you set the cursor (caret) to a specified state or obtain what state the caret is in.

```
typedef enum
{
    dont_draw_cursor,   // Do nothing
    toggle_cursor,      // Toggle cursor based on timer
    show_cursor,        // Show cursor
    hide_cursor,        // Hide cursor
    deactivate_cursor,  // Cursor is no longer active
    update_cursor,      // Redraw cursor per current state
    restore_cursor,     // Turn cursor back on (chiefly Windows usage)
```

```
}
```

NOTE: Except for very unusual applications, you should generally only use this function with `force_cursor_off` and `force_cursor_on`.

To obtain the current cursor state, call `pgGetCursorState`, which will return either TRUE (cursor is currently ON) or FALSE (cursor is currently OFF).

See also "Activate / Deactivate".

NOTE: *The function result of `pgGetCursorState` has differing usages in OpenPaige for Windows and for Macintosh.* For **Windows**, the result implies whether or not the System caret is actively blinking within the `pg_ref`. For **Macintosh**, TRUE/FALSE result implies whether or not the caret is visible at that instant while it is toggling during `pgIdle()`.

```
void pgSetCaretPosition (pg_ref pg, pg_short_t position_verb, pg_boolean
show_caret);
```

This function should be used to change the location of the caret (insert position); for example, `pgSetCaretPosition` is useful for handling arrow keys.

The position_verb indicates the action to be taken. The low byte of this parameter should be one of the following values:

```
enum
{
    home_caret,
    doc_bottom_caret,
    begin_line_caret,
    end_line_caret,
    next_word_caret,
    previous_word_caret
}
```

The high byte of `position_verb` can modify the meaning of the values shown above; the high byte should either be equal to zero or to `EXTEND_CARET_FLAG`.

The following is a description for each value in `position_verb`:

`home_caret` — If `EXTEND_CARET_FLAG` is set, the text is selected from the beginning of the document to the current position; if `EXTEND_CARET_FLAG` is clear, the caret moves to the beginning

of the document.

doc_bottom_caret — If EXTEND_CARET_FLAG is set, the text is selected from the current position to the end of the document; if EXTEND_CARET_FLAG is clear the caret advances to the end of the document.

begin_line_caret — If EXTEND_CARET_FLAG is set, the text is selected from the current position to the beginning of the current line; if EXTEND_CARET_FLAG is clear the caret moves to the beginning of the line.

end_line_caret — If EXTEND_CARET_FLAG is set, the text is selected from the current position to the end of the current line; if EXTEND_CARET_FLAG is clear the caret moves to the end of the line.

next_word_caret — If EXTEND_CARET_FLAG is set, the text is selected from the current position to the beginning of the next word; if EXTEND_CARET_FLAG is clear the caret moves to the beginning of the next word.

previous_word_caret — If EXTEND_CARET_FLAG is set, the text is selected from the current position to the beginning of the previous word; if EXTEND_CARET_FLAG is clear the caret moves to the beginning of the previous word.

If show_caret is TRUE then the caret is redrawn in its new location; otherwise, the caret does not visibly change.

10.5 Selection shape

It is possible to create a selection by specifying a shape. This next function returns a list of select_pairs when given a shape.

```
(void) pgShapeToSelections (pg_ref pg, shape_ref the_shape, memory_ref
selections);
```

FUNCTION RESULT: This function will place a list of selection pairs in selections that contain all the text that intersects the_shape. What gets put into selections is an array of select_pair records, similar to what is returned from pgGetSelectionList.

The memory_ref passed to selections must be a valid memory allocation (which you must create).

It is also possible to determine the selection shape.

```
(void) pgSelectToShape (pg_ref pg, memory_ref select_shape, pg_boolean
show_hilite);
```

This function sets the selection range(s) in `pg` to all characters that intersect the specified shape.

For example, if the `select_shape` was one large rectangle expanding across the entire document, then every character would be selected; if the shape were smaller than the document, then only the characters that fit within that shape—whether wholly or partially—would be selected.

If `show_hilite` is TRUE, the new selection region is drawn.

For information about shapes and individual characters and insertion point, see "Text and Selection Positions". For information about highlighting see "Activate / Deactivate".

Activate/Deactivate with shape of selection still showing

Macintosh

This function can be used to draw the selection area around text when it is deactivated.

```
void Do_Activate(Boolean Do_An_Activate)
{
    pg_ref my_pg;

    if (!(my_pg = Get_pgref_from_window(WPtr_Untitled1))) return;

    if(Do_An_Activate   // Handle the activate
    {
                        // Update the scrollbar values
                        // ---------------------------
                        // Turn on the selection hilites
        pgSetHiliteStates(my_pg, activate_verb, no_change_verb, TRUE);
    }
    else                // Handle the deactivate
    {
                        // Turn off the scroll bars here
                        // -----------------------------
```

```
                        // Turn off the selection hilites
        pgSetHiliteStates(my_pg, deactivate_verb, no_change_verb, TRUE);
        outline hilite(my pg);
        /* do this if you want to draw an outline around the selected text if
the window is deactivated, as in MPW or the OpenPaige demo */
    }                   // End IF
}

/* If you want the feature of drawing the line around the selected text when
the window is deactivated, you can use this snippet from the OpenPaige demo */

#include "pgTraps.h"    // This draws xor-hilight outline

static void outline_hilite(pg_ref the_pg)
{
    shape ref outline shape;
    outline_shape = pgRectToShape(&paige_rsrv, NULL);
    if (pgGetHiliteRgn(the_pg, NULL, NULL, outline_shape))
    {
        pg_scale_factor scale_factor;
        RgnHandle rgn;
        rectangle vis_r;
        Rect clip;
        PushPort(WPtr_Untitled1);
        PushClip();

        pgAreaBounds(the_pg, NULL, &vis_r);
        RectangleToRect(&vis_r, NULL, &clip);
        ClipRect(&clip);

        rgn = NewRgn();
        pgGetScaling(the_pg, &scale_factor);
        ShapeToRgn(outline_shape, 0, 0, &scale_factor, rgn);
        PenNormal();
        PenMode(patXor);
        SET_HILITE_MODE(50);
        FrameRgn(rgn);
        DisposeRgn(rgn);
        PopClip();
        PopPort()
    }
    pgDisposeShape(outline_shape);
}
```

11 All about scrolling

Scrolling an OpenPaige object is handled differently than previous DataPak technology, with a wider feature set.

11.1 The ways to scroll

An OpenPaige object can be scrolled in one of four ways: by *unit*, by *page*, by *absolute position*, or by a *pixel* value.

1. Scrolling by *unit* generally means to scroll one text line increment for vertical scrolling, and some predetermined distance for horizontal scrolling.
2. Scrolling by *page* means to scroll one visual area's worth of distance (clicking the "grey" areas of the scroll bar).
3. Scrolling by *absolute position* means the document scrolls to some specified location (such as the result of dragging a "thumb").
4. Scrolling by *pixel* means to move the position up or down by an absolute pixel amount; generally, this method is used if for some reason all of the above methods are unsuitable to your application.

For scrolling by a unit, page or absolute value, when an OpenPaige object is scrolled vertically, an attempt is always made to align the results to a line boundary (so a partial line does not display across the top or bottom).

11.2 How OpenPaige Actually Scrolls

In reality, neither the text nor the page rectangle within an OpenPaige object ever "moves". Whatever coördinates you have set for an OpenPaige object's `page_area` (shape in which text will flow) remains constant and do not change; the same is true for the `vis_area` and `exclude_area`.

The way an OpenPaige object changes its "scrolled" position, however, is by offsetting the display and/or the relative position of a "mouse click" when you call `pgDragSelect` or any other function that translates a coördinate point to a text location. The scrolled position is a single vertical and horizontal value maintained within the `pg_ref`; these values are added to the top-left coördinates

for text display at drawing time, and they are added to the mouse coördinate when click/dragging.

This could be important information if your application needs to implement some other method for scrolling, because all you would need to do is leave OpenPaige alone (do not call its scrolling functions) and offset the display yourself (`pgDisplay` will accept a horizontal and vertical value to temporarily offset the display). Realise that nothing every really moves; lines are always in the same vertical and horizontal position unless your app explicitly changes them.

NOTE: Class library users — when implementing an OpenPaige-based document, you are generally better off letting OpenPaige handle it own scrolling. If at all possible, do not implement `scrollView` classes that attempt to scroll by changing the window origin.

11.3 The scroll

`pgScroll`

```
void pgScroll (pg_ref pg, short h_verb, short v_verb, short draw_mode);
```

Scrolls the OpenPaige object by a single unit, or by a page unit. A unit and page unit is described at "Scroll Values". In short, `pgScroll` scrolls a specified `h_verb` and `v_verb` distance.

The values to pass in `h_verb` and `v_verb` can each be one of the following:

```
typedef enum
{
    scroll_none,    // Do not scroll
    scroll_unit,    // Scroll one unit
    scroll_page,    // Scroll one page unit
    scroll_home,    // Scroll to top of document
    scroll_end  // Scroll to end of document
}
scroll_verb
```

Because OpenPaige will scroll the text some number of pixels, a certain amount of "white space" will result on the top or bottom for vertical scrolling, or on the left or right for horizontal scrolling. Hence, the `draw_mode` indicates the drawing mode OpenPaige should use when it refreshes the "white space" areas; normally, the value given for `draw_mode` should be `best_way`.

On the other hand, while a value of draw_none will disable all drawing and visual scrolling completely, the text contents will still be "moved" by the specified amounts. In other words, were the OpenPaige document to be scrolled one page down (using pgScroll) but with draw_none given for draw_mode, nothing would change on the screen until the application redisplayed the OpenPaige text contents. In this case, the refreshed screen would appear to be scrolled one page down. The "draw nothing" feature for scrolling is therefore used only for special cases, in which an application wants to "move" the visual contents up or down without yet drawing anything.

draw_mode can be the values as described in "Draw Modes" on page 2-30:

```
draw_none,      // Do not draw at all
best way,       // Use most efficient method(s)
direct_copy,    // Directly to screen, overwrite
direct_or,      // Directly to screen, "OR"
direct_xor,     // Directly to screen, "XOR"
bits_copy,      // Copy offscreen
bits_or,        // Copy offscreen in "OR" mode
bits_xor        // Copy offscreen in "XOR" mode
```

Examples

Macintosh

```
if (the_key == HOME_KEY)
{
    pgScroll(doc -> pg, scroll_home, scroll_home, best_way);
    UpdateScrollbarValues(doc);
}
else
if (the_key == END_KEY)
{
    pgScroll(doc -> pg, scroll_none, scroll_end, best_way);
    UpdateScrollbarValues(doc);
}
```

Responding to WM_HSCROLL and WM_VSCROLL events (Windows)

```
case WM_HSCROLL:
{
    switch(wParam)
    {
        case SB_PAGEDOWN:
            pgScroll(pg, -scroll_page, scroll_none, best_way);
            break;
        case SB_LINEDOWN:
            pgScroll(pg, -scroll_unit, scroll_none, best_way);
            break;
        case SB PAGEUP:
            pgScroll(pg, scroll_page, scroll_none, best_way);
            break;
        case SB_LINEUP:
            pgScroll(pg, scroll_unit, scroll_none, best_way);
            break;
        case SB_THUMBPOSITION:
        {
            short cur_h, cur_v, max_h, max_v;
            pg getScrollValues(pg, &cur_h, &cur_v, &max_h, &max_v);
            pgSetScrollValues(pg, LOWORD(lParam), cur_v, TRUE, best_way);
            break;
        }
    }

    UpdateScrollbars(pg, hWnd);
}
case WM_VSCROLL:
    if (pg)
    {
        switch (wParam)
        {
            case SB_PAGEDOWN:
                pgScroll(pg, scroll_none, scroll_page, best_way);
                break;
            case SB_LINEDOWN:
                pgScroll(pg, scroll_none, scroll_unit, best_way);
                break;
            case SB PAGEUP:
                pgScroll(pg, scroll_none, scroll_page, best_way);
                break;
            case SB_LINEUP:
                pgScroll(pg, scroll_none, scroll_unit, best_way);
                break;
```

```
            case SB_TOP:
                pgScroll(pg, scroll_none, scroll_home, best_way);
                break;
            case SB_BOTTOM:
                pgScroll(pg, scroll_none, scroll_end, best_way);
                break;
            case SB_THUMBPOSITION:
                case SB_THUMBTRACK:
                {
                    short cur_h, cur_v, max_h, max_v;
                    pgGetScrollValues(pg. &cur_h, &cur_v, &max_h, &max_v);
                    pgSetScrollValues(pg, &cur_h, LOWORD(lParam), TRUE,
best_way);
                    break;
                }
            }
            updateScrollbars(pg, hWnd);
        }
    return 0;
```

pgScrollToView

```
(pg_boolean) pgScrollToView (pg_ref pg, long text_offset, short h_extra, short
v_extra, short align_line, short draw_mode);
```

Scrolls an OpenPaige object so a specific location in its text is visible. Canonically, this function is used to automatically scroll to the "current line," although it could also be used for a number of other purposes (such as find/replace) to show specific text location.

The location in `pg`'s text is given in `text_offset`; `pg` will scroll the required distance so the character at `text_offset` is at least `h_extra` pixels from the left or right edge of the view area and `v_extra` pixels from the top or bottom edge. Whether the distance is measured from the top or bottom, or left or right depends in the value of `h_pixels` and `v_pixels`; if `h_extra` is positive, the character must scroll at least `pg` pixels from the left, otherwise the right edge is used. For `v_extra`, a positive number uses the top edge and a negative number uses the bottom edge.

The `text_offset` parameter can be `CURRENT_POSITION` (value of -1), in which case the current insertion point is used to compute the required scrolling, if any.

FUNCTION RESULT: The function returns "TRUE" if scrolling occurred.

The `draw_mode` indicates how the text should be updated. The value given is identical to the `display_modes` described for `pgDisplay`; it should be noted that a value of zero will cause the text not to update at all, which technically could be used to simply "offset" the OpenPaige object contents without doing a physical scroll at all.

Scroll to cursor position (Windows)

```
// ScrollToCursor forces a scroll to the current insertion point (if any)

void ScrollToCursor(pg_ref, pg, HWND hWnd)
{
    short state1, state2;
    pg GetHiliteStates(pg, &state1, &state2;

    if (state1 == deactivate_verb || state2 == deactivate_verb)
        return;

    if (!pgNumSelections(pg))
    {
        pgPaginateNow(pg. CURRENT_POSITION, FALSE);

        if (pgScrollToView(pg, CURRENT_POSITION, 32, 32, TRUE, best_way))
            UpdateScrollbars(pg, hWnd);
    }
    else
        UpdateScrollbars(pg, hWnd);
}
```

Scroll to cursor position (Macintosh)

```
// ScrollToCursor is called to "autoscroll" to the insertion point

short ScrollToCursor(doc_rec *doc)
{
    short old_h_value;
    if(!pgNumSelections(doc → pg))
    {
        old_h_value = GetCtlValue(doc → h_ctl);
```

```
        if (pgScrollToView(doc → pg, CURRENT_POSITION, 32, TRUE, best_way))
        {
            UpdateScrollbarValues(doc);
            update_ruler(doc, old_h_value);
            return TRUE;
        }
    UpdateScrollbarValues(doc);
    return FALSE;
}
```

TECH NOTE: Can't scroll past end of text

> I've noticed that I cannot scroll vertically past the end of the text in the window. So if the OpenPaige document is empty, it is not possible to scroll vertically at all. I need to be able to scroll vertically until the bottom part of the 640x480 workspace is visible, even if the user has not yet typed any text. How do I do that?

You need to force your `pg_ref` to be fixed height, not "variable". When you do `pgNew`, the default document mode is "variable", meaning that the bottom of the last text line is considered the document's bottom.

A "fixed" height document is one whose page shape itself (not the text) determines the document's bottom. From your description of the app, I think this is what you want.

To do so, you need to set `BOTTOM_FIXED_BIT` and `MAX SCROLL_ON_SHAPE` in the `pg_doc_info`'s attributes field. You do this right after `pgNew`, like this:

```
pg_doc_info doc_info;
pgGetDocInfo(pg, &doc_info);
doc info.attributes ⊨ (BOTTOM FIXED BIT | MAX_SCROLL_ON_SHAPE);
pgSetDocInfo(pg, &doc_info, FALSE, draw_none);
```

This will tell OpenPaige to scroll to the bottom of your page area regardless of how much (or how little) text there is.

Of course doing this you must now make sure your page shape is exactly whatt you want, e.g. 640x480 (which you said it is).

This "bonus" on this is that you will never have to worry about scrolling; i.e. you won't need to constantly adjust the scrollbar max values once they are set up because openPaige will only look at

the page area's bottom. EXCEPTION: when you resize window you'll need to adjust (see answer below).

TECH NOTE: Smaller window/bad rectangle

> If I resize my window to be "small", scroll to the far right and far bottom edges of the workspace, then resize the window to be "large", I am left with the bottom right corner of the workspace in the upper left corner of the screen. What I need to be able to do is to have openPaige adjust the scrolled position so that the bottom right corner of the workspace is in the bottom right corner of the screen. How do I do that?

There is actually an OpenPaige function for this exact situation:

```
PG_PASCAL (pg_boolean) pgAdjustScrollMax (pg_ref pg, short draw_mode);
```

What this does is the following:

1. Checks current scrolled position, and:—
2. If you are now scrolled too far by virtue of having resized the window, OpenPaige will scroll the doc to "adjust."

Hence, you don't wind up with the situation you described. The function result is TRUE if it had to adjust (had to scroll).

However, I haven't tried this yet on a "fixed height" doc (per my suggestion above), but I can't think of why it shouldn't work.

Where this function should fit in the scheme of things is:

1. After resize, resize the `pg_ref` (`pgGrowVisArea` or whatever you do), then:—
2. Call `pgAdjustScrollMax`.

If there's nothing to "fix" in the scrolling, OpenPaige won't do anything.

TECH NOTE: Vertical scrolling behaves strangely

> In the demo & in my application as well since I extracted scrolling code from the demo, vertical scrolling behaves strangely. As the text approaches the bottom of the window

> the current position indicator moves up rather than down. When the current input position reaches the bottom of the visible portion of the window and the window automatically scrolls up to create extra visible space below the input position, the current position indicator on the scrollbar moves down. I would expect it to move up to reflect the fact that the current position is no longer at the bottom of the window.

I'm not sure how else this could ever work, at least in relation to how the demo sets up the document.

First, the reason the indicator moves "up" as you approach the bottom is that OpenPaige is adding a whole new, blank page. So let's say you start with one page and approach the bottom and the indicator shows 90% of the document has scrolled down. Suddenly OpenPaige appends a new page, so now the doc has 2 pages. In this case the scrolled position is no longer 90%, but rather 50%, so naturally the indicator has to move UP.

Following the 90-to-50% indicator change, if the document then auto-scrolls down by virtue of typing, then of course the indicator moves DOWN. This sequence is exactly as you described, which is "correct" in every respect due to the way the document has been created by the demo.

If this is too disconcerting you can work around it in a couple of ways. The first way is *not* to implement "repeater shapes" the way the demo is doing it, but instead just make one long document. You do this by not setting the `V_REPEAT_BIT` in `pg_doc_info`. The end result will be less noticeable with the scrōll indicator (might move a tiny bit but won't jump so far) because OpenPaige will just add a small amount of blank space instead of a whole page.

If you still want "repeater" shapes to get the page-by-page effect as in the demo, then the only workaround is to display something to the user that shows *why* the indicator has moved so much. For example, you could display "Page 1 of 1" and "Page 1 of 2" etc. So, when OpenPaige inserts a new blank page, it might be obvious to user why the indicator jumps if "Page 1 of 1" changes to "Page 1 of 2".

TECH NOTE: Scrolling doesn't include picture at bottom of document

> I have implement pictures anchored to the document (where text wraps around them). However, if I have a picture below the last line of text, I can't ever scroll the document down to that location. How do I fix this?

I looked over your situation with OpenPaige exclusion areas (pictures). OpenPaige actually does support what you need.

In Paige.h you will notice the following definition near the top of the file:

```
#define EX_DIMENSION_BIT    0x00000100  /* Exclude area is included as
width/height */
```

When you call pgNew, giving EX_DIMENSION_BIT as one of the attribute flags tells OpenPaige to include the exclusion area as part of the "document height"—which I believe is exactly what you want.

The reason for this attribute—and the reason OpenPaige does not automatically include an embedded objects anchored to the page—is because it cannot make that assumption, but in many cases (such as your own), setting EX_DIMENSION_BIT tells OpenPaige to go ahead and assume that.

TECH NOTE: How do I make OpenPaige scroll to the right when using word wrap

> I am building a line editor, which expands to the right, very much like a C source code editor. But my right margin is the right side of the text. How do I get it to scroll correctly?

I think the reason you're having a problem is that OpenPaige can only go by what is set in the document bounds (the "page area") to determine what the width of the document is.

Hence, the answer lies somewhere in forcing the pg_ref's page area to expand as text expands to the right. At thât time OpenPaige will adjust its maximum scroll values, its clipping area, etc.—assuming you set the page area using the high-level functions in Paige.h.

The real trick is to figure out how wide the text area is. I'll create some examples of how you determine the current width of a no-wrap document. See "Getting the Max Text Bounds".

11.4 Scroll Parameters

Set Scroll Params

```
(void) pgSetScrollParams (pg_ref pg, short unit_h, short unit_v, short
append_h, short append_v);
```

Sets the scroll parameters for `pg` as follows: `unit_h` and `unit_v` define the distance each scrolling unit shall be. This means if you ask OpenPaige to scroll `pg` by one unit, horizontal scrolling will advance `unit_h` pixels and vertical scroll will advance `unit_v` pixels.

However, `unit_v` can be set to zero, in which case "variable" units apply. What occurs in this case (i.e., with `unit_v` equal to zero) is a scrolling distance of whatever is applicable for a single line.

For example, if the line immediately below the bottom of the visual area is 18 pixels, a scrolling down of one unit will move 18 pixels; if the next line is 12 pixels, the next down scrolling would be 12 pixels, and so on.

`append_h` and `append_v` define extra "white" space to allow for horizontal maximum and vertical maximum, respectively.

For example, suppose you create an OpenPaige document whose total "height" is 400 pixels. Normally, the scrolling functions in OpenPaige would not let you scroll beyond that point. The `append_v` value, however, is the amount of extra distance you will allow for scrolling vertically: if the `append_v` were 100, then a 400-pixel document would be allowed to scroll 500 pixels.

If you create a new `pgRef` and do not call `pgSetScrollParams`, the defaults are as follows: `unit_h = 32`, `unit_v = 0`, `append_h = 0`, `append_v = 32`.

Create scroll bars (Macintosh)

```
// Create a pair of scrollbars
CreateScrollbars(WindowPtr w_ptr, doc_rec new_doc_;
{
    Rect r_v, r_h, paginate_rect;
    InitWithZeros(&new_doc, sizeof(doc_rec));

    new_doc.w_ptr = w_ptr;
    new_doc.mother = mother_window;
    new_doc.pg = create_new_paige(w_ptr);

    pgSetTabBase(new_doc.pg, TAB_WRAP_RELATIVE);
    pgSetScrollParams(new doc.pg, 0, 0, 0, VERTICAL_EXTRA);
    get_paginate_rect(w_ptr, &paginate_rect);
```

```
    r_v = w_ptr -> portRect;
    r_v.left = r_v.right - 16;
    r_v.bottom -= 13;
    r_h = w_ptr -> portRect;
    r_h.left = paginate_rect.right;
    r_h.top = r_h.bottom - 16;
    r_h.right -= 13;
    OffsetRect(&r_v, 1, -1);
    OffsetRect(&r_h, -1, 1);

    new_doc.v_ctl = NewControl(w_ptr, &r_v, "", TRUE, 0, 0, 0, scrollBarProc,
0);
    new_doc.h_ctl = NewControl(w_ptr, &r_h, "", TRUE, 0, 0, 0, scrollBarProc,
0);
}
```

Getting scroll parameters

```
(void) pgGetScrollParams (pg_ref pg, short PG_FAR *unit_h, short PG_FAR
*unit_v, short PG_FAR *append_h, short PG_FAR *append_v);
```

Returns the scroll parameters for pg. These are described above for `pgSetScrollParams`..

11.5 Scroll Values

Getting scroll indicator values

```
(short) pgGetScrollValues (pg_ref pg, short PG_FAR *h, short PG_FAR *v, short
PG_FAR *max_h, short PG_FAR *max_v);
```

This is the function you call to get the exact settings for scroll indicators.

On the Macintosh, for example, you would call `pgGetScrollValues` and set the vertical scrollbar's

value to the value given in `*v` and its maximum to the value in `*max_v`. The same settings apply to the horizontal scrollbar for `*h` and `*max_h`.

Note that the values are `short`s. OpenPaige assumes your controls can only handle ±32 K; hence, it computes the correct values even for huge documents that are way larger than a scroll indicator could handle.

FUNCTION RESULT: The function returns "TRUE" if the values have changed since the last time you called `pgGetScrollValues`. The purpose of this Boolean result is to not slow down your app by excessively setting scrollbars when they have not changed.

NOTE: The values returned from `pgGetScrollValues` are *guaranteed* to be within the ± range of an integer value. That means if the document is too large to report a scroll position within the confines of 32K, OpenPaige will adjust the ratio between the scroll value and the suggested maximum to accommodate this limitation to most controls.

CAUTION: `pgGetScrollValues` can return "wrong" values if a major text change has occurred (such as a large insertion, or deletion, or massive style and font changes) but no text has been redrawn.

The reason scroll values will be inaccurate in these cases is because OpenPaige has not yet recalculated the new positions of text lines - pwhich normally occurs dynamically as it displays text - bso it has no idea that the document's text dimensions have changed.

To avoid this situation, the following rules should be observed:

- A common scenario that creates the "wrong" scroll value is importing a large text file (without drawing yet, for speed purposes), then attempting to get the scrollbar maximum to set up the initial scrollbar parameters, all before the window is refreshed. To avoid this situation, it is generally wise to force-paginate the document following a massive insertion if you do not intend to display its text prior to getting the scroll values.
- Always call `pgGetScrollValues` *after* the screen has been updated following a major text change, and never before. Normally, this is not a problem because most of the text-altering functions accept a `draw_mode` parameter which, if ≠ 0, tells OpenPaige to update the text display. There are special cases, however, when an application has reasons to implement large text changes yet passes `draw_none` for each of these; if that be the case, the screen should be updated at least once prior to `pgGetScrollbarValues`, OR the document should be repaginated using `pgPaginateNow`.

Logical Steps

The following pseudo instructions provide an example for any OpenPaige platform when determining the values that should be set for both horizontal and vertical scrollbars:

```
if (I just made a major text change and did not draw)
    pgPaginateNow(pg, CURRENT POSITION, FALSE);
if (pgGetScrollValues(pg, &h, &v, &max_h, &max_v)) returns "TRUE" then
    I should change my scrollbar values as:
    Set horizontal scrollbar maximum to max_h
    Set horizontal scrollbar value to h
    Set vertical scrollbar maximum to max_v
    Set vertical scrollbar value to v
else
    Do nothing.
```

Update scrollbar values (Windows)

```
void UpdateScrollbars (pg_ref pg, HWND hWnd)
{
    short max_h, max_v;
    short h_value, v_value;

    if (pgGetScrollValues(pg (short far *) &h_value, short far, (short far *)
&max_h, short far *) &max_v));
    {
        if max_v < 1)
            max_v = 1; // For Windows I don't want scrollbar disappearing
        SetScrollRange (hWnd, SB_VERT, 0, max_v, FALSE);
        SetScrollRange (hWnd, SB_HORZ, 0, max_h, FALSE);
        SetScrollPos (hWnd, SB VERT, v value, TRUE);
        SetScrollPos (hWnd, SB_HORZ, h_value, TRUE);
```

Update scrollbar values (Macintosh)

```
void UpdateScrollbarValues (doc_rec *doc)
{
    short h, v, max_h, max_v;
```

```
    if (pgGetScrollValues(doc -> pg, &h, &v, &max_h, &max_v))
    {
        SetCtlMax(doc -> v_ctl, max_v);
        SetCtlValue(doc -> v_ctl, v);
        SetCtlMax(doc -> h_ctl, max_h);
        SetCtlValue(doc -> h_ctl, h);
    }
}
```

TECH NOTE: "Wrong" Scroll Values

> In my application I need to scroll to certain characters or styles in the document. I noticed, however, that the visual location of these special characters are often "wrong", so when I attempt to scroll to these places I do not wind up at the correct place.

Regarding the scrolling issues, you've touched upon a classic problem that I have been handling with support for years and years. "To Paginate or Not To Paginate, that is the question", *pace* Shakespeare.

When dealing with potentially large word-wrapping text, the editor must avoid repaginating the whole document *at all costs*; otherwise, performance is major dog-slow.

Most of our users that have graduated from TextEdit (Macintosh) or EDIT controls (Windows) are limited in their document size and never understand this problem, because TextEdit maintains an array of line positions at all times. That's because it doesn't handle a lot of text so it can get away with it. Our text engines, on the other hand, support massive documents, changing point sizes, irregular wrapping and who knows what else. Hence, to learn the exact document height at any given time, OpenPaige must calculate every single word-wrapping line to come up with a good answer.

To avoid turning into a major dog, OpenPaige (and its predecessors) elect to repaginate only at the point they *display*. There are several good reasons for this, the most important one being a typical OpenPaige-based app applies all kinds of inserts, embedding, style changing and the like before displaying; if OpenPaige decided to repaginate each time you set a selection or inserted a piece of text or made any changes whatsoever, it would become unbearably slow.

The reason I'm explaining all of this is so you understand WHY your document behaves the way it does with regards to scrolling. Your problem is simply: you have not yet drawn the part of the document that you will scroll to, hence it is unpaginated, hence the "wrong" answer from `pgGetScrollvalues`. That is also why `auto-scroll-to-cursor` works a wee bit better, because

the auto-scroll forces a redisplay, which forces a paginate, which forces new information about the doc's height which can then return the "right" answer.

Putting it simpler, pgGetScrollValues doesn't have sufficient information about the whole doc if a part of the doc is "dirty" and undisplayed. That's why forced paginate fixes the problem. That's also why the "wrong" answer from pgGetScrollValues is intermittent—your doc won't always be "dirty" every time you call the function, and also sometimes OpenPaige's best-guess in this case is correct anyway.

So yes, pgPaginateNow (see "Paginate Now") is the best approach; I would call it every time before getting the scrollbar info. The problem with your current logic—paginating *after* pgGetScrollValues—is that the document hasn't been computed yet for pgGetScrollValues, so it might return FALSE, thinking that the document is unchanged. Remember, pgPaginateNow isn't that bad since it won't do anything unless the document really needs it.

But, you should pass CURRENT_POSITION for the paginate_to parameter— that will help performance a bit.

Setting scroll values

```
(void) pgSetScrollValues (pg_ref pg, short h, short v, short align_line, short
draw_mode);
```

This function is the reverse of pgGetScrollValues. It provides a way to do absolute position scrolling, if necessary.

For example, you would use pgSetScrollValues after the "thumb" is moved to a new location. As in pgGetScrollValues, the values are shorts, but OpenPaige computes the necessary distance to scroll. (Because of possible rounding errors, however, after you have called pgSetScrollValues you should immediately change the scroll indicator settings with the values from a freah call to pgGetScrollValues.

Handling scrolling with mouse (Macintosh)

```
/* ClickScrollBars gets called in response to a mouseDown event. If mouse is
not within a control, this function returns FALSE and does nothing. Otherwise,
scrolling is handled and TRUE is returned. */
```

```
int ClickScrollBars (doc_rec *doc, EventRecord *event)
{
    Point start_pt;
    short part_code;
    ControlHandle the_control;
    start_pt = event → where;

    GlobalToLocal(&start_pt);

    if (part_code = FindControl(start_pt, doc→ w_ptr, &the_control))
    {
        scrolling_doc = doc;
        if (part_code == inThumb)
        {
            long max h, max v;
            long scrolled_h, scrolled_v;
            long scroll h, scroll v;
            short v_factor, old_h_position;

            if (TrackControl(the_control, start_pt, NULL))
            {
                old_h_position = GetCtlValue(doc → h_ctl);
                pgSetScrollValues(doc → pg, GetCtlValue(doc → h_ctl),
GetCtlValue(doc → v_ctl), TRUE, best_way);
                UpdateScrollbarValues(doc);
                update_ruler(doc, old_h_position);
            }
            else
                TrackControl(the_control, start_pt, (ProcPtr)
scroll action_proc);
        }
        return (part_code =/= 0);
    }
```

Maximum scroll value

Adjustments may be needed after large deletions; if so, call the following function.

```
(pg_boolean) pgAdjustScrollMax (pg_ref pg, short, draw_mode);
```

This tells OpenPaige that pg might need some adjustment after a large deletion or text size change.

For example, suppose you had a document in 24-point text, scrolled to the bottom. User changes the text to 12 point, resulting in a scrolled position way too far down! If you call pgAdjustScrollMax, this situation is corrected (by scrolling up the required distance).

If draw_mode ≠ 0, actual physical scrolling takes place (otherwise the scroll position is adjusted internally and no drawing occurs). draw_mode can be the values as described in "Draw Modes":

```
draw_none,      // Do not draw at all
best_way,       // Use most efficient method(s)
direct copy,    // Directly to screen, overwrite
direct_or,      // Directly to screen, "OR"
direct xor,     // Directly to screen, "XOR"
bits_copy,      // Copy offscreen
bits or,        // Copy offscreen in "OR" mode
bits_xor        // Copy offscreen in "XOR" mode
```

FUNCTION RESULT: The function returns TRUE if the scroll position changed.

11.6 Getting/Setting Absolute Pixel Scroll Positions

```
void pgScrollPixels (pg_ref pg, long h, long v, short draw_mode);
```

FUNCTION RESULT: This function scrolls pg by h and v pixels; scrolling occurs from the current position (i.e., scrolling advances plus or minus from its current position by h or v amount(s).

If draw_mode ≠ 0, actual physical scrolling takes place (otherwise the scroll position is adjusted internally and no drawing occurs).

OpenPaige will not scroll out of range — the parameters are checked and OpenPaige will only scroll to the very top or to the maximum bottom as specified by the document's height and the current scroll parameters.

NOTE: You should only use this function if you are not using the other scrolling methods listed above.

```
(void) pgScrollPosition (pg_ref pg, co_ordinate_ptr scroll_pos);
```

FUNCTION RESULT: The above function returns the current (absolute pixel) scroll position. The vertical scroll position is placed in `scroll_pos → v` and the horizontal position in `scroll_pos -> h`.

The positions, however, are always zero or positive: when OpenPaige offsets the text to its "scrolled" position, it subtracts these values.

Forcing Pixel Alignment

In some applications, it is desirable always to scroll on "even" pixel boundaries, or some multiple other than one.

For example, in a document that displays grey patterns or outlines, it can be necessary to always scroll in a multiple of two pixels, otherwise the patterns can be said to be out of "alignment."

To set such a parameter, call the following:

```
(void) pgSetScrollAlign (pg_ref pg, short align_h, short align_v);
```

The pixel alignment is defined in `align_h` and `align_v` for horizontal and vertical scrolling, respectively.

For either parameter, the effect is as follows:

- if the value is zero, the current alignment value remains unchanged.
- if the value is one, scrolling is performed to the nearest single pixel (i.e., no "alignment" is performed)
- if the value is two or more, that alignment is used.

For example, if `align_v` is two, vertical scrolling would always be in multiples of two pixels; if three, alignment would always be a multiple of three pixels, etc.

NOTES:

1. The current scrolled position in `pg` is not changed by this function. You must therefore make sure the scrolled position is correctly aligned or else all subsequent scrolling can be constantly "off" of the desired alignment. It is generally wise to set the alignment once, after `pgNew`, while the scrolled positions are zero.
2. The default alignment after `pgNew` is one.
3. You do not need to set scroll alignment after a file is opened (with `upgraded`); scroll alignment is saved with the document.

Getting Alignment

```
(void) pgGetScrollAlign (pg_ref pg, short PG_FAR *align_h, short PG_FAR
*align_v);
```

This function returns the current scroll alignment. The horizontal alignment is returned in `*align_h` and vertical alignment in `*align_v`.

Both `align_h` and `align_v` can be NULL pointers, in which case they are ignored.

11.7 Performing Your Own Scrolling

Because certain environments and frameworks support document scrolling in many different ways, a discussion here that explains what actually occurs inside an OpenPaige object that is said to be "scrolled" might prove helpful.

When OpenPaige text is "scrolled," a pair of long integers inside the `pg_ref` is increased or decreased which defines the extra distance, in pixels, that OpenPaige should draw its text relative to the top-left of the window.

This is a critical point to consider for implementing other methods of scrolling: the contents of an OpenPaige document *never actually "move" by virtue of `pgScroll`, `pgSetScrollParams` or `pgSetScrollValues`*. Instead, only two long words within the `pg_ref` (one for vertical position and one for horizontal position) are changed. When the time comes to display text, OpenPaige temporarily subtracts these values from the top-left coördinates of each line to determine the target display coördinates; but the coördinates of the text lines themselves (internally to the `pg_ref`) remain unchanged and are always relative to the top-left of the window's origin regardless of scrolled position.

Similarly, when pgDragSelect is called (to detect which character(s) contain a mouse coördinate), OpenPaige does the same thing in reverse: it temporarily adds the scroll positions to mouse point to decide which character has been clicked, again no text really changes its position.

Considering this method, the following facts might prove useful when pgScroll needs to be bypassed altogether and/or if your programming framework requires a system of scrolling:

- A pg_ref that is "scrolled" is simply a pg_ref whose vertical and horizontal "scroll position" fields are nonzero; at no time does text really "scroll." OpenPaige temporarily subtracts these scroll positions from the display coördinates of each line when it comes time to draw the text.
- The "scroll position" values can be obtained by calling pgScrollPosition.
- The "scroll position" can be set directly by doing a UseMemory(pg_ref), changing Paige_rec_ptr → scroll_position, then UnuseMemory(pg_ref).
- The "scroll positions" are always positive, i.e. as the document scrolls from top to bottom or from left to right, the scroll positions increase proportionally by that many pixels.
- The simplest way to understand a pg_ref's "scroll position" is to realise that OpenPaige only cares about the scroll position when it draws text or processes a pgDragSelect().
- When pgScroll is called, all that really happens is the screen pixels within the vis_area are scrolled, the scroll positions are changed to new values, then the text is redrawn so the "white space" fills up.
- If draw_none is given to pgScroll, all that occurs is the scroll positions are changed (no pixels are scrolled and no text is redrawn).
- A call to pgGetScrollValues merely returns the value from the scroll position members (with the values modified as necessary to achieve ≤16-bit integer result and adjusted to match what the application has defined as a "scroll unit").

11.8 Alternate Scrolling

Scrolling a pg_ref "normally", using pgScroll() and similar functions, the top-left coördinates of the document are changed internally. However, rather than changing the window origin itself, OpenPaige handles this by remembering these scroll values, and offsetting the position of text at the time it draws its text.

Using this default scrolling method, OpenPaige assumes that the window origin never changes and that the visual region is relatively constant.

This method, however, can be troublesome within frameworks that require a document to scroll in some other way, especially by changing the window origin. Additionally, certain aspects of these frameworks are difficult to disable and are therefore rendered unfriendly to the OpenPaige

environment.

Most applications that require a different method of scrolling feel they are required to bypass OpenPaige's scrolling system completely. While this may be workable, the app suddenly looses all scrolling features in OpenPaige. For instance, aligning to the top and bottom of lines can be lost; OpenPaige's built-in suggestions of where to set scrollbars is lost, etc.

Furthermore, developers that need to bypass OpenPaige's scrolling suffer a loss in performance. For example, such an application might need to have an exact "document height", and it might thus continuously need to change the OpenPaige `shapes` region and `vis_area`.

The purpose of the features and functions in this section is to provide additional support to scroll many different ways.

External Scrolling Attribute

A flag bit has been defined that can help applications that want to do their own scrolling:

```
#define EXTERNAL_SCROLL_BIT 0x00000010
```

If you include this bit in the flags parameter for `pgNew()`, OpenPaige will assume that the application's framework will be handling the document's top-left positioning in relation to scrolling.

What this means is if you create the `pg_ref` with `EXTERNAL_SCROLL_BIT`, you can continue to use all the regular OpenPaige scrolling functions without actually changing the relative position of text (i.e., you can control the position of text and the view area yourself while still letting OpenPaige compute the document's maximum scrolling, its current scroll position and the amount you should scroll to align to lines).

For example, using the default built-in scrolling methods (without `EXTERNAL_SCROLL_BIT` set), calling `pgScroll()` will move the display up or down by some specified amount; calling `pgGetScrollValues()` will return how far the text moved. However, if `EXTERNAL_SCROLL_BIT` is set, calling `pgScroll()` will change the scroll position values stored in the pg_ref yet *the text display itself remains unaffected*. But calling `pgGetScrollValues()` will correctly reflect the scroll position values (the same as it would using the default scrolling method).

Hence, with `EXTERNAL_SCROLL_BIT` set you can still use all of the OpenPaige scrolling functions—yet you can adjust the text display using some other method.

Changing Window Origin

NOTE: The term "window origin" in this section refers to the machine-specific origin of the window where the `pg_ref` is "attached;" it does not refer to the "origin" member of the `graf_device` structure.

The only problem with changing the window's origin that contains a `pg_ref` is after you have changed the origin, OpenPaige's internal `vis_area` is no longer valid.

Using the default OpenPaige scrolling system, an application would have to force new `vis_area` shapes into the `pg_ref` every time the origin changed. However, this is inefficient. The following new function has been provided to optimise this situation:

```
void pgWindowOriginChanged (pg_ref pg, co_ordinate_ptr original_origin,
co_ordinate_ptr new_origin);
```

If the window in which `pg` lives has changed its top-left origin *for the purpose of moving its view area in relation to text*, you should immediately call this function.

By "view area in relation to text" is meant that the window origin has changed to achieve a scrolling effect.

You would *not* call this function if you simply wanted the whole `pg_ref` to move, both `vis_area` and `page_area`. The intended purpose of `pgWindowOriginChanged` is to inform OpenPaige that your app has changed the (OS-specific) window origin to create a scrolled effect, hence the `vis_area` needs to be updated.

The `original_origin` should contain the normal origin of the window, i.e. what the top-left origin of the window was initially when you called `pgNew()`. The `new_origin` should contain what the origin is now.

Note that the `original_origin` must be the original window origin at the time the `pg_ref` was created, not necessarily the window origin that existed before changing it to new_origin. Typically, the original origin is (0, 0).

However, `original_orgin` can be a null pointer, in which case the position (0, 0) is assumed. Additionally, `new_origin` can also be a null pointer, in which case the current scrolled position (stored inside the `pg_ref`) will be assumed as the new origin.

OpenPaige will take the most efficient route to update its shape(s) to accommodate the new origin. Text is not drawn, nor are the scrolled position values (internal to the `pg_ref`) changed. All that

changes is the vis_area coördinates so any subsequent display will reflect the position of the text in relationship to the visual region.

Oldies but Goodies

```
pgSetScrollParams();
pgGetScrollParams();
pgGetScrollValues();
pgScroll();
```

The above functions are documented elsewhere in this manual, but they are listed again to encourage their use even when customising OpenPaige scrolling. If you create the pg_ref with EXTERNAL_SCROLL_BIT, you can begin using all the functions above without actually changing the relative position of text (i.e., you can control the position of text and the "view" area yourself while still letting OpenPaige compute the document's maximum scrolling, its current scroll position and the amount you should scroll to align to lines).

Additional Support

```
void pgScrollUnitsToPixels (pg_ref pg, short h_verb, short v_verb, pg_boolean
add_to_position, pg_boolean window_origin_changes, long PG_FAR *h_pixels, long
PG_FAR *v_pixels);
```

This function returns the amount of pixels that OpenPaige would scroll if you called pgScroll() with the same h_verb and v_verb values. In other words, if you are doing your own scrolling but want to know where OpenPaige would scroll if you asked it to, this is the function to use.

However, this function also provides the option to change the internal scroll values in the pg_ref, and/or to inform OpenPaige that you will be changing the window origin.

Note that if you created the pg_ref with EXTERNAL_SCROLL_BIT, you can change the scroll position values inside the pg_ref but the text itself does not "move." This will allow your application's framework to position the text by changing the window origin, etc., but you can still have OpenPaige maintain the relative position(s) that the document is scrolled.

Upon entry, h_verb and v_verb should be one of the several scroll verbs normally given to

pgScroll().

If add_to_position is TRUE, OpenPaige adjusts its internal scroll position (which does not affect visual text positions if EXTERNAL_SCROLL_BIT has been set in the pg_ref). If FALSE, the scroll positions are left alone.

If window_origin_changes is TRUE, OpenPaige assumes that the new scroll position, by virtue of the h_verb and v_verb values, will change the window origin by that same amount. In other words, passing TRUE for this parameter is effectively the same as calling pgWindowOriginChanged() with coördinates that reflect the new origin after the scroll positions have been updated.

When this function returns, *h_pixels and *v_pixels will be set to the number of pixels that OpenPaige would have scrolled had you passed the same h_verb and v_verb to pgScroll().

Physical Drawing/Scrolling Support

```
pg_region pgScrollViewRect (pg_ref pg, long h_pixels, long v_pixels, shape_ref
update_area);
```

This function will physically scroll the pixels within pg's vis_area by h_pixels and v_pixels; negative values cause the image to move up and left respectively.

When the function returns, if update_area is not MEM_NULL it is set to the shape of the area that needs to be updated.

```
void pgSetCaretPosition (pg_ref pg, pg_short_t position_verb, pg_boolean
show_caret);
```

This function should be used to change the location of the caret (insert position); for example, pgSetCaretPosition is useful for handling arrow keys.

The position_verb indicates the action to be taken. The low byte of this parameter should be one of the following values:

```
enum
{
    home_caret,
```

```
    doc_bottom_caret,
    begin_line_caret,
    end_line_caret,
    next_word_caret,
    previous_word_caret
};
```

The high byte of position_verb can modify the meaning of the values shown above; the high byte should be either zero or set to EXTEND_CARET_FLAG.

The following is a description for each value in position_verb:

home_caret — If EXTEND_CARET_FLAG is set, the text is selected from the beginning of the document to the current position; if EXTEND_CARET_FLAG is clear the caret moves to the beginning of the document.

doc_bottom_caret — If EXTEND_CARET_FLAG is set, the text is selected from the current position to the end of the document; if EXTEND_CARET_FLAG is clear the caret advances to the end of the document.

begin_line_caret — If EXTEND_CARET_FLAG is set, the text is selected from the current position to the beginning of the current line; if EXTEND_CARET_FLAG is clear the caret moves to the beginning of the line.

end_line_caret — If EXTEND_CARET_FLAG is set, the text is selected from the current position to the end of the current line; if EXTEND_CARET_FLAG is clear the caret moves to the end of the line.

next_word_caret — If EXTEND_CARET_FLAG is set, the text is selected from the current position to the beginning of the next word; if EXTEND_CARET_FLAG is clear the caret moves to the beginning of the next word.

previous_word_caret — If EXTEND_CARET_FLAG is set, the text is selected from the current position to the beginning of the previous word; if EXTEND_CARET_FLAG is clear the caret moves to the beginning of the previous word.

If show_caret is TRUE then the caret is redrawn in its new location, otherwise the caret does not visibly change.

NOTE: This function is simply a portable way to physically scroll the pixels within a pg_ref — no change occurs to the scroll position internal to the pg_ref , nor does the window origin or the vis_shape change in any way.

```
void pgDrawScrolledArea (pg_ref pg, long pixels_h, long pixels_v,
co_ordinate_ptr original_origin, co_ordinate_ptr new_origin, short draw_mode);
```

This function will draw the `pg_ref` inside the area that would exist (or already exists) after a pixel scroll of `pixels_h` and `pixels_v`.

For example, if you (or your framework) has already scrolled the document by, say, -60 pixels, a call to `pgDrawScrolledArea(pg, 0, -60, …)` will cause the document to update within the region that exists by virtue of such a scroll.

NOTE: This function fills the would-be update area of a scroll but does not actually scroll anything.

However, optional parameters exist to inform OpenPaige about window origin changes; if you have changed the window origin since the last display, and have not told OpenPaige about it yet, you can pass the original and new origin in `original_origin` and `new_origin` parameters, respectively. These parameters do the same exact thing as on `pgWindowOriginChanged()` — except if they are null pointers in this case, they are ignored.

```
void pgLastScrollAmount (pg_ref pg, long *h_pixels, long *v_pixels);
```

This function returns the amount of the previous scrolling action, in pixels.

The "scrolling action" would have been any OpenPaige function that has changed the `pg_ref`'s internal scroll position. That includes `pgScroll` and `pgScrollUnitsToPixels()` if applicable, *inter alia*.

By "previous scrolling" is meant the last function call that changed the scroll position. For example, there could have been 1,000 non-scrolling functions since the last scrolling change, but `pgLastScrollAmount()` would only return the values since the last scrolling.

11.9 Draw Scroll Hook & Scroll Regions

An application could repaint the area uncovered by a scroll with the `draw_scroll` hook:

```
PG_PASCAL(void) pgDrawScrollProc (paige_rec_ptr pg, shape_ref update_rgn,
co_ordinate_ptr scroll_pos, pg_boolean post_call);
```

This function gets called by OpenPaige after the contents of a pg_ref have been scrolled; the update_rgn shape contains the area of the window that has been uncovered (rendered blank) by the scrolling.

However, an unintentional anomaly exists with this method: the update_rgn contains a shape that represents the entire bounding area of the scrolled area. This presents a problem if the scrolled area is non-rectangular.

For example, an application might have a "Find..." dialogue box in front of the document. If a word is found, causing the document to scroll, the uncovered document area is non-rectangular (the region is affected by the intersection of the Find window).

The basic problem is that OpenPaige cannot convert a non-rectangular, platform-specific region into a shape_ref.

The paige_rec structure (provided as the pg parameter in the above hook) contains the member .port, which contains a member called scroll_rgn. The scroll_rgn will be a platform-specific region handle containing the actual scrolled region.

For example, if draw_scroll is called, pg → port.scroll_rgn would be a RgnHandle for Macintosh and an HRGN for Windows. In both cases, if you were to fill that region with something, it would conform to the exact scrolled area, rectangular or not.

As a rule, to avoid problems with non-rectangular scrolled area(s), use pg → port.scroll_rgn instead of the update_rgn parameter.

12 ALL ABOUT SHAPES

The quickest way to get "Up & Running" with shapes is to see "Up & Running Shapes". This shows how to get a document up within rectangles to display and/or edit.

This chapter provides more details should you wish to provide your users with more complex shapes.

12.2 Basic shape areas

As mentioned in several places in this document, an OpenPaige object maintains three basic shape

areas.

The exact description and behavior for each of these shapes is as follows:

`vis_area` — The "viewable" area of an OpenPaige object. Stated simply, anything that OpenPaige displays that is even one pixel outside the `vis_area` gets clipped (masked out). Usually, the `vis_area` in an OpenPaige object is some portion (or all) of a window's content area and remains unmoving and stationary. (See Figure 8 *infra*).

`page_area` — The area in which text will flow. For the simplest documents, the `page_area` can be considered a rectangle, or "page" which defines the top-left position of text display as well as the maximum width. For example, if you wanted to create a document representing an 8" wide page, you simply specify a `page_area` that is 8 inches wide. Hence, text will wrap within those boundaries.

The `page_area` may or may not be the same size as the `vis_area`, and may or may not align with the `vis_area`'s top-left position. In fact, a large document on a small monitor would almost always be larger than the `vis_area` (see Figure 8).

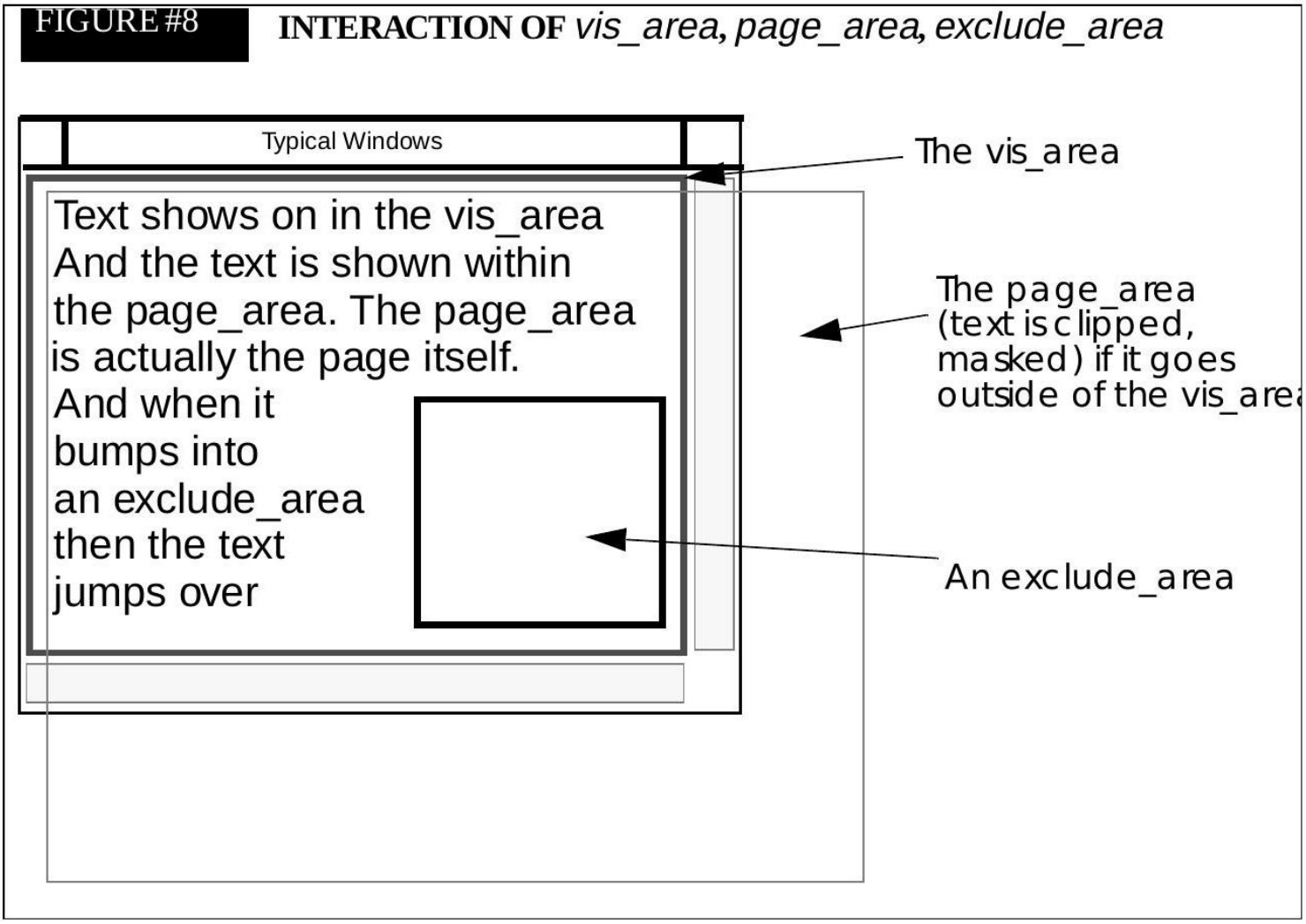
`exclude_area` - An optional area of an OpenPaige object which text flow must avoid. An good example of implementing an `exclude_area` would be placing a picture on a document which text must wrap over (or wrap around from left to right). The easiest way to do this would be to build an `exclude_area` that contains the picture's bounding frame, resulting in the forced avoidance of text for that area.

All three shapes can be changed dynamically at any time. Changing the `page_area` would force text to rewrap to match the new shape; changing the `exclude_area` would also force text to rewrap in order to avoid the new areas.

If you are specifically implementing "containers", see "Containers Support" which might provide an easier path.

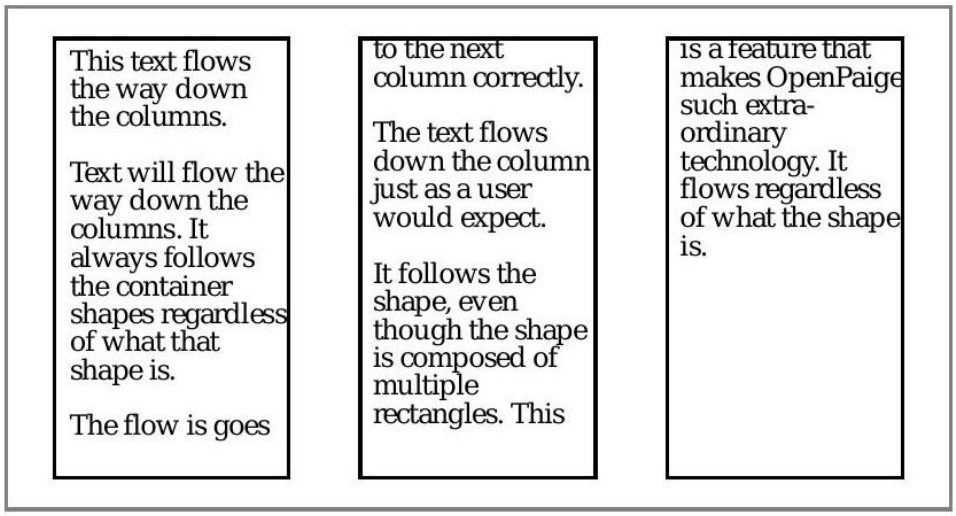
If you are implementing any kind of exclusion shapes, see "Exclusion Areas".

FIGURE #8 **INTERACTION OF** *vis_area, page_area, exclude_area*



As stated, the simplest documents are rectangles; however, the `page_area` can be non-rectangular. A good example of this would be columns in which text must flow from one column to the other. In this case, the `page_area` would look similar to what is shown in Figure 9 *infra*.

FIGURE#9 *pg_area* **NON-RECTANGULAR (text flows from "column to column")**



12.3 Coordinates & Graphic Structures

For purposes of cross-platform technology, OpenPaige defines its own set of structures to represent screen positions (coordinates) and shapes. Except for machine-specific source files, no reference is made to, say, Macintosh "QuickDraw" structures.

The main components ("building blocks") of shapes are the following record structures:

Rectangle

```
typedef struct
{
    co_ordinate top_left;   // Top-left of rect
    co_ordinate bot_right;  // Bottom-right of rect
}
rectangle, *rectangle_ptr;
```

Co_ordinate

```
typedef struct
{
    long v; // vertical position
    long h; // horizontal position
}
co_ordinate;
```

12.4 What's Inside a Shape

Shapes are simply a series of rectangles. A very complex shape could theoretically be represented by thousands of rectangles, the worst-case being one rectangle surrounding each pixel.

All shape structures consist of a bounding rectangle (first rectangle in the array) followed by one or more rectangles; the bounding rectangle (first one) is constantly updated to reflect the bounding area of the whole shape as the shape changes.

Hence, the shape structure is defined simply as:

```
typedef rectangle shape; // Also a "shape", really
typedef rectangle_ptr shape_ptr;
```

A shape is maintained by OpenPaige, however, as a memory_ref to a block of memory that contains the shape information. In the header it is defined as:

```
typedef memory_ref shape_ref; // Memory ref containing a "shape"
```

12.5 Rules for Shapes

The following rules apply to shapes with respect to the list of rectangles they contain:

1. If rectangle edges are connected exactly (i.e., if two edges have the same value), they are considered as "one" even if such a union results in a non-rectangular shape (see Figure 10).
2. If rectangle edges are not connected, they are considered separate "containers;" even if they overlap. (Overlapping would result in overlapping text if the shape definition was intended

for the area where text is drawn).

FIGURE #10 **"CONNECTING" RECTANGLES**

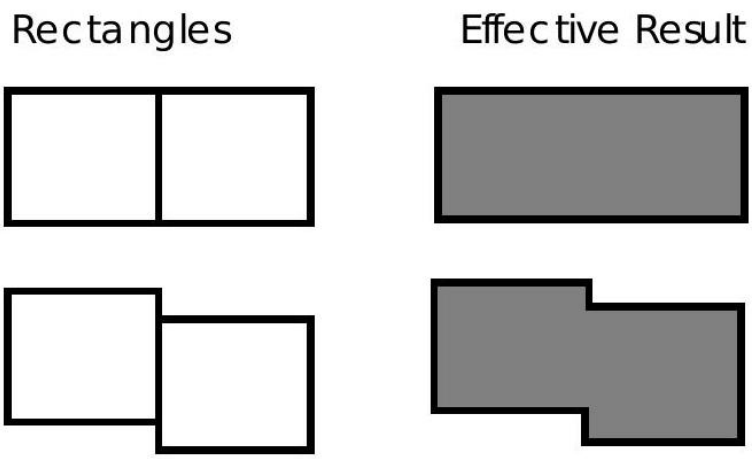
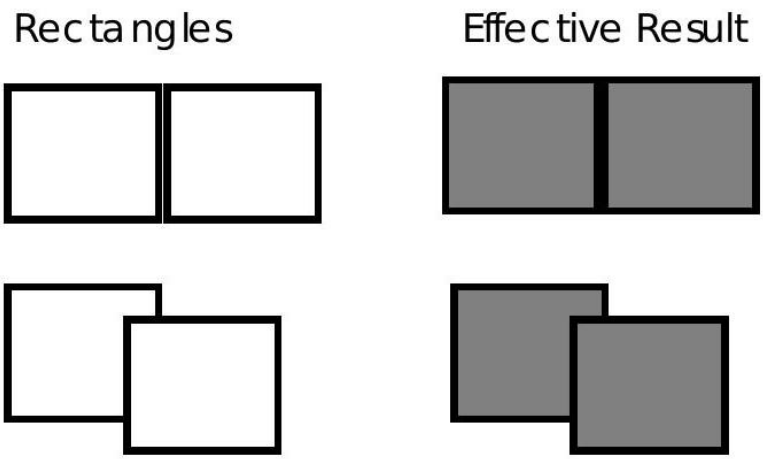


FIGURE #11 **"NON-CONNECTING" RECTANGLES**



12.6 Building Shapes

Placing data into the `shape_ref` is the subject of discussion in this section. However, you will not normally manipulate the `shape_ref` data directly.

Creating new shapes

The easiest way to create a new shape is to use the following function:

```
(shape_ref) pgRectToShape (pgm_globals_ptr globals, rectangle_ptr rect);
```

This returns a new `shape_ref` (which can be passed to one of the "area" parameters in `pgNew`). The `globals` parameter must be a pointer to the same structure given to `pgMemStartup()` and `pgInit()`.

The `rect` parameter is a pointer to a rectangle; this parameter, however, can be a null pointer in which case an empty shape is returned (shape with all sides = 0).

Setting a Shape to a Rectangle

If you have already created a `shape_ref`, you can "clear" its contents and/or set the shape to a single rectangle by calling the following:

```
(void) pgSetShapeRect (shape_ref the_shape, rectangle_ptr rect);
```

The shape `the_shape` is changed to represent the single rectangle `rect`. If `rect` is a null pointer, `the_shape` is set to an empty shape.

Adding to a New Shape

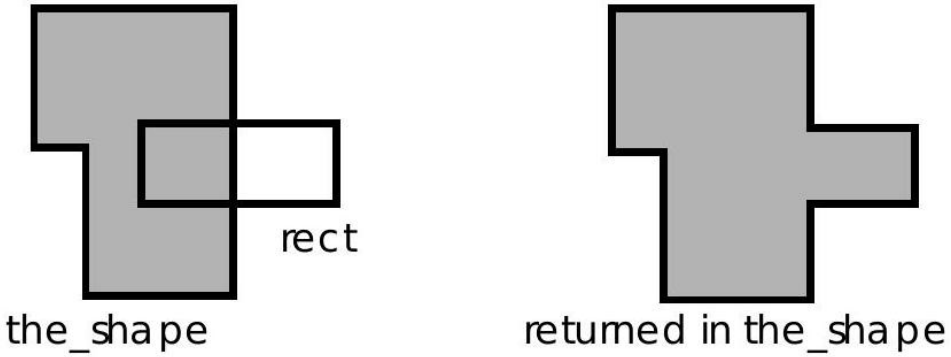
The best way to build a shape requiring more than one rectangle is to call the following:

```
(void) pgAddRectToShape (shape_ref the_shape, rectangle_ptr rect);
```

The rectangle pointed to by `rect` is added to the rectangle list in `the_shape`, combining it with other rectangles if necessary. When a rectangle is added, `pgAddRectToShape` first explores all existing rectangles in the_shape to see if any of them can "merge" with rect (see "Rules for Shapes"). If none can be combined, `rect` is appended to the end of the list.

If `the_shape` is empty, `the_shape` gets set to the dimensions of rect (as if you had called `pgSetShapeRect` *supra*).

FIGURE #12 **RESULT OF ADDING A SHAPE**



Disposing a Shape

To dispose a shape, call:

```
(void) pgDisposeShape (shape_ref the_shape);
```

`Rect` to Rectangle

Two utilities exist that make it easier to create OpenPaige rectangles:

```
#include "pgTraps.h"
(void) RectToRectangle (Rect PG_FAR *r, rectangle_ptr pg_rect);
(void) RectangleToRect (rectangle_ptr pg_rect, co_ordinate_ptr offset, Rect
PG_FAR *r);
```

`RectToRectangle` converts `Rect r` to rectangle `pg_rect`. The `pg_rect` parameter must be a pointer to a rectangle variable you have declared in your code.

`RectangleToRect` converts `pg_rect` to `r`; also, if offset is non-null the resulting `Rect` is offset by the amounts of the coordinate (for example, if `offset.h` and `offset.v` were (10, 5) the resulting

Rect would be the values in `pg_rect` with left and right amounts offset by 10 and top and bottom amounts offset by -5.

NOTE (Macintosh): Since a Mac `Rect` has a ±32 K limit for all four sides, OpenPaige rectangle sides larger than 32 K will be intentionally truncated to about 30 K.

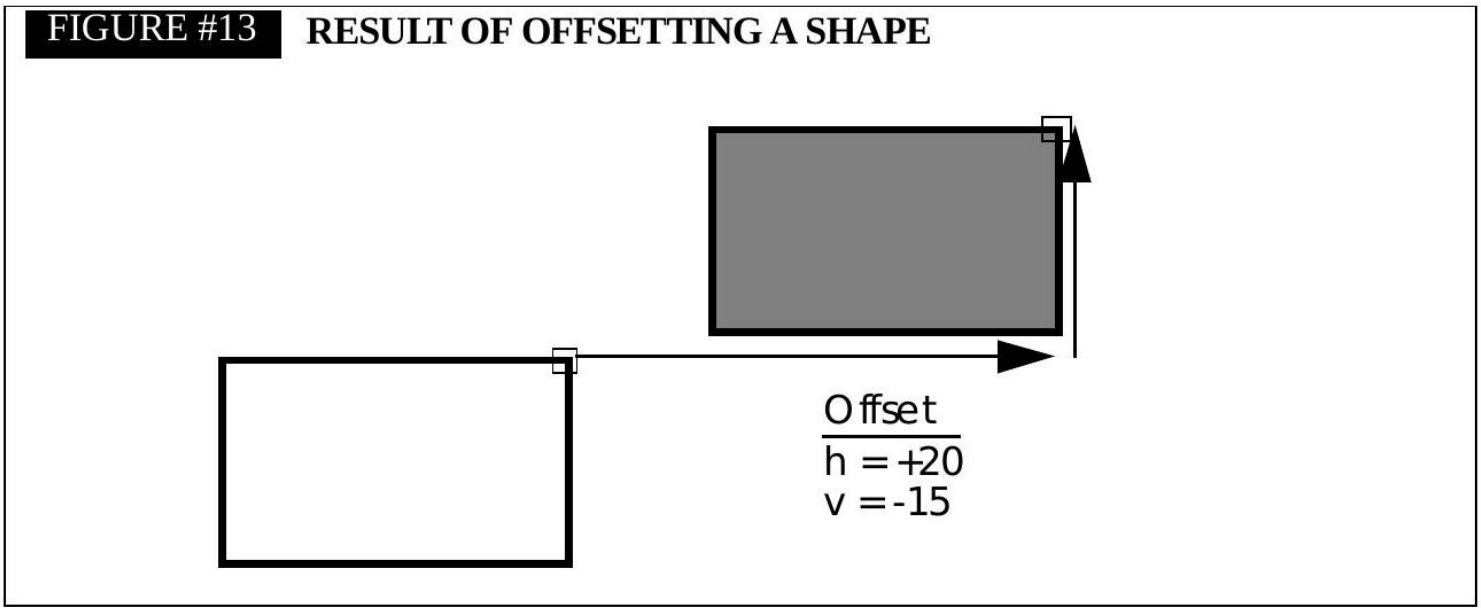
NOTE: You *must* `#include "pgTraps.h"` in any code that calls either function above.

12.7 Manipulating shapes

Moving shapes

```
(void) pgOffsetShape (shape_ref the_shape, long h, long v);
```

Offsets (moves) `*the_shape` by `h` (horizontal) and `v` (vertical) distances. These may be negative. Positive numbers move to the right horizontally and down vertically as appropriate.



Shrinking or expanding shape

```
(void) pgInsetShape (shape_ref the_shape, long h, long v);
```

Insets (shrinks or expands) *the_shape by h and v amounts. Positive numbers inset the shape inwards and negative numbers expand it.

```
(pg_short_t) pgPtInShape (shape_ref the_shape, co_ordinate_ptr point,
co_ordinate_ptr offset_extra, co_ordinate_ptr inset_extra, pg_scale_ptr
scaling);
```

pgPtInShape returns "TRUE" if point is within any part of the_shape (actually, the rectangle number is returned beginning with #1). The point is temporarily offset with offset_extra if offset_extra is non-null before checking if it is within the_shape (and the offset values are checked in this case, not the original point).

If scaling is non-NULL, the_shape is temporarily scaled by that scale factor. For no scaling, pass NULL.

Also, each rectangle is temporarily inset by the values in inset_extra if it is non-NULL. Using this parameter can provide extra "slop" for point-in-shape detection. Negative values in inset_extra enlarge each rectangle for checking and positive numbers reduce each rectangle for checking.

NOTE: For convenience, the_shape can be also be MEM_NULL, which of course returns FALSE.

```
(pg_short_t) pgSectRectInShape (shape_ref the_shape, rectangle_ptr rect,
rectangle_ptr sect_rect)
```

Checks to see if a rectangle is within the_shape. First, offset_extra, if non-null, moves rect by the amount in offset_extra.h and offset_extra.v, then checks if it intersects any part of the_shape. The result is TRUE if any part of rect is within the shape, FALSE if it is not. If the_shape is empty, the result is always FALSE.

Actually, a "TRUE" result will really be the rectangle number found to intersect, beginning with 1 as the first rectangle.

NOTE: A result of TRUE does not necessarily mean that rect doesn't intersect with any other rectangle in the_shape; rather, one rectangle was found to intersect and the function returns.

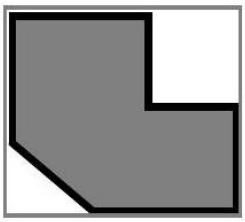
If sect_rect is not MEM_NULL, it gets set to the intersection of rect and the first rectangle in the_shape found to intersect it.

Shape Bounds

```
(void) pgShapeBounds (shape_ref the_shape, rectangle_ptr bounds);
```

Returns the rectangle bounds of the outermost edges of the_shape. The bounds is placed in the rectangle pointed to by bounds (which cannot be null).

FIGURE #14 **GRAY RECTANGLE REPRESENTS BOUNDS RETURNED BY *pgShapeBounds*.**



Comparing Shapes

```
(pg_boolean) pgEmptyShape (shape_ref the_shape);
```

FUNCTION RESULT: This function returns TRUE if `the_shape` is empty (all sides are the same or all zeros).

```
(pg_boolean) pgEqualShapes (shape_ref shape1, shape_ref shape2);
```

FUNCTION RESULT: Returns TRUE if `shape1` matches `shape2` exactly, even if both are empty.

Intersection of shapes

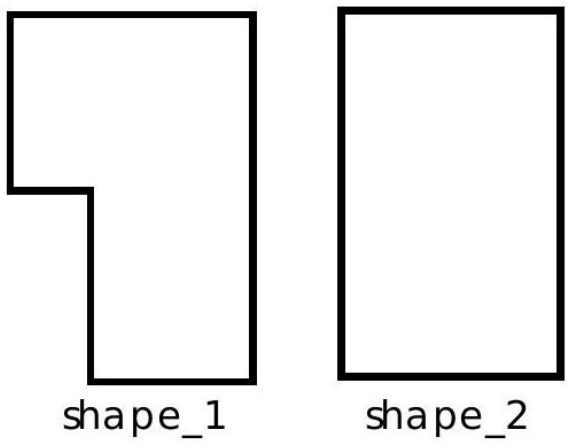
```
(pg_boolean) pgSectShape (shape_ref shape1, shape_ref shape2, shape_ref
result_shape);
```

Sets result_shape to the intersection of shape1 and shape2. All shape_ref parameters must be valid shape_ref s, except result_shape can be MEM_NULL (which you might want to pass just to check if two shapes intersect). Additionally, result_shape cannot be the shape shape_ref as shape1 or shape2 or this function will fail.

If either shape1 or shape2 is an empty shape, the result will be an empty shape. Also, if nothing between shape 1 and shape 2 intersects, the result will be an empty shape.

FUNCTION RESULT: The function result will be TRUE if any part of shape1 and shape2 intersect (and result_shape gets set to the intersection if not MEM_NULL), otherwise FALSE is returned and result_shape gets set to an empty shape (if not MEM_NULL).

FIGURE #15 **NON-INTERSECTING SHAPES RETURN *FALSE* AND *MEM_NULL* IN RETURN SHAPE**



FUNCTION RESULT: Neither shape1 nor shape2 are altered by this function.

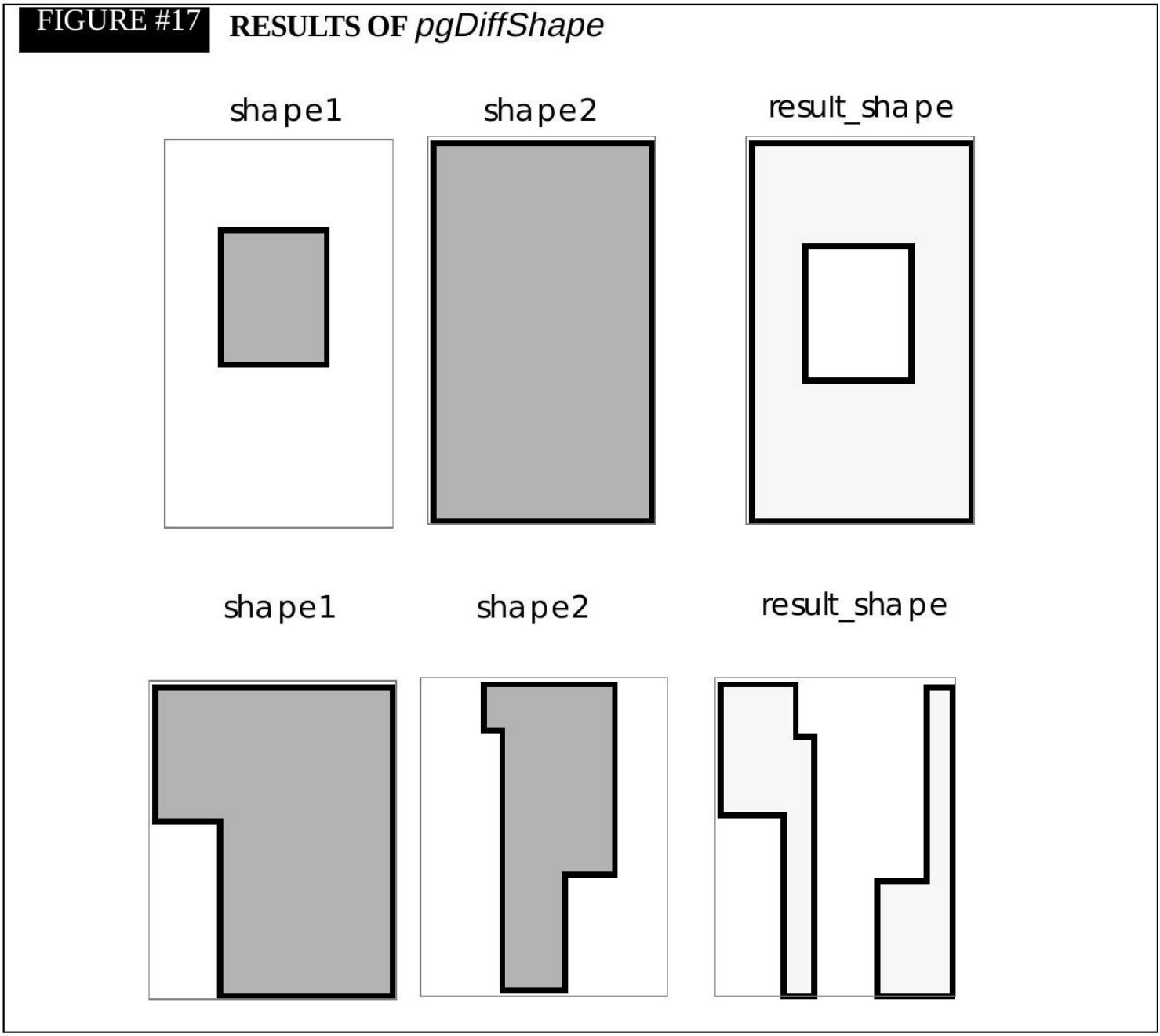
```
(void) pgDiffShape (shape_ref shape1, shape_ref shape2, shape_ref
result_shape);
```

FUNCTION RESULT: This function places the difference in result_shape between shape1 and shape2.

Unlike `pgSectShape`, `result_shape` cannot be `MEM_NULL`; however, it *can* be the same `shape_ref` as `shape1` or `shape2`.

The "difference" is computed by subtracting all portions of `shape1` from `shape2`, and the geometric difference(s) produce `result_shape`. If `shape1` is an empty shape, `result_shape` will be a mere copy of shape2; if `shape2` is empty, `result_shape` will be empty.

FIGURE #17 **RESULTS OF *pgDiffShape***



Erase a Shape

```
(void) pgEraseShape (pg_ref pg, shape_ref the_shape, pg_scale_ptr scale_factor,
co_ordinate_ptr offset_extra, rectangle_ptr vis_bounds);
```

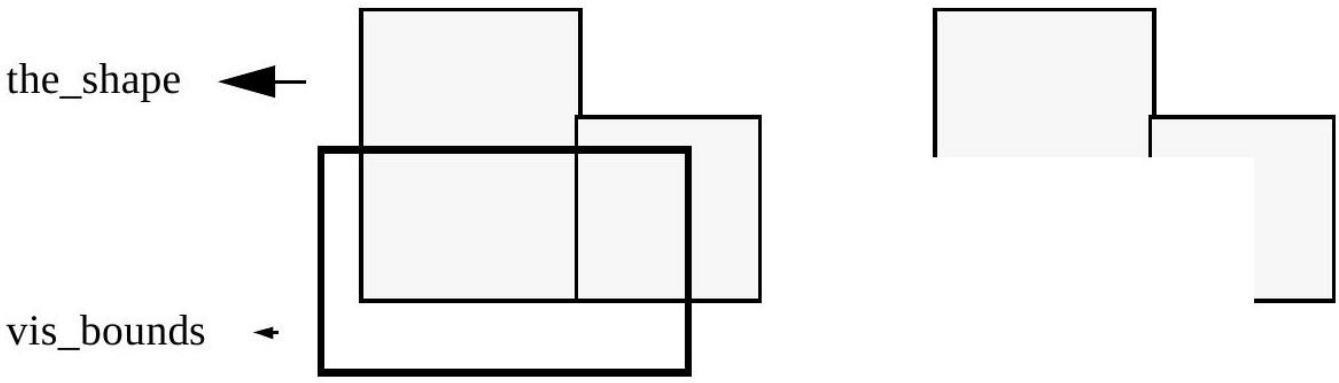
FUNCTION RESULT: This function will erase `the_shape` (by filling it with the background colour of the device in `pg`).

The `scale_factor` parameter defines scaling, if any; for no scaling, pass zero for this parameter. If you want scaling, see "Scaling an OpenPaige Object".

If `offset_extra` is non-null, `the_shape` is temporarily offset by `offset_extra → h` and `offset_extra → v` amounts before the erasure occurs.

If `vis_bounds` is non-null, then only the parts of `the_shape` that intersect with `vis_bounds` get erased; otherwise, the whole shape is erased (see illustration *infra*).

FIGURE #18 **THE EFFECT OF *pgEraseShape* IF *vis_bounds* IS NON_NULL ONLY THE PORTION(S) OF THE SHAPE THAT INTERSECT **vis_bounds* ARE ERASED. IF *vis_bounds* IS A NULL POINTER, THE WHOLE SHAPE IS ERASED.**



Moving a Shape in a `pg_ref`

```
(void) pgOffsetAreas (pg_ref pg, long h, long v, pg_boolean offset_page,
pg_boolean offset_vis, pg_boolean offset_exclude);
```

This function "moves" the page area and/or visual area and/or the exclusion area of `pg`. If `offset_page` is TRUE, the page area is moved; if `offset_vis` is TRUE the visual area is moved; if `offset_exclude` is TRUE the exclusion area is moved.

Each area is moved horizontally and vertically by `h` and `v` pixels, respectively. What occurs is `h` gets added to the left and right sides of all rectangles enclosed in the shape while `v` gets added to top and bottom. Hence the shape is moved left or right, up or down with negative and positive values, respectively.

NOTE: The contents of `pg` are not redrawn.

12.8 Region Conversion Utilities

```
void ShapeToRgn (shape_ref src_shape, long offset_h, long offset_v,
pg_scale_factor PG_FAR *scale_factor, short inset_amount, rectangle_ptr
sect_rect, RgnHandle rgn);
```

This function sets region `rgn` to `src_shape`. In addition, the region is offset by `offset_h` and `offset_v` amounts. If `scale_factor` is non-NULL, the resulting region is scaled by that scaling factor (see "Scaling").

Each rectangle added to the region is inset by `inset_amount` (`inset_amount` is added to the top and left and subtracted from right and bottom).

If `sect_rect` is non-null, every rectangle in the shape is first intersected with `sect_rect` and the intersection (only) is output to the region.

NOTE: You *must* `#include "pgTraps.h"` to use this function.

NOTE (Windows): - `RgnHandle` is `typedef`ed in `pgTraps.h` and is the same as `HRGN`.

CAUTION: Converting huge complex shapes to a region can be slow.

Picture Handle to Shape (Macintosh only)

The following function is available only for Macintosh that takes a picture and produces a shape that encloses the picture's outside edges:

```
#include "pgTraps.h"
(void) PictOutlineToShape (PicHandle pict, shape_ref target_shape, short
accuracy)
```

Given a picture in `pict` and a `shape_ref` in `target_shape`, this function sets `target_shape` to surround the outside bit image of the picture.

The `accuracy` parameter can be a value from 0 to 7 and indicates how "accurate" the shape should be: 0 is the most accurate (but consumes the most memory) and 7 is the least accurate (but consumes the least memory). The `accuracy` value actually indicates how many pixels to skip, or "group" together in forming the image. If `accuracy = 0`, the image is produced to the nearest pixel — which theoretically can mean that a rectangle is produced for every pixel surrounding the image (which is why so much memory can be consumed).

The picture does not need to be a bitmap image, and it can be in colour (the image is produced around the outside edges of all nonwhite areas for colour).

NOTE: Large, complex images can not only consume huge amounts of memory but can take several seconds to produce the image, so use this function sparingly!

NOTE: You *must* `#include "pgTraps.h"` to use this function.

12.9 Page Area Background Colours

OpenPaige will support any background colour (which your machine can support) even if the target window's background colour is different.

The page area (area text draws and wraps) will get filled with the specified colour before text is drawn; hence this features lets you overlay text on top of non-white backgrounds (or, if desirable, will also let you overlay white text on top of dark or black backgrounds).

Note that this differs from the `bk_color` value in `style_info`. When setting the `style_info` background, OpenPaige will simply turn on that background colour only for that text. Setting the general background colour (using the functions below) sets the background of the entire page area.

COLOUR TEXT AND TEXT BACKGROUND

NOTE: See "Setting/Getting Text Color" or "Changing Styles" for information about setting text

colour and text background colour.

OpenPaige will also recognize which colour is considered "transparent." Normally, this would be the same color as the window's normal background colour, typically "white."

"Transparent" is simply the background colour for which OpenPaige will not set or force. Defining which color is transparent in this fashion lets you control the background colour(s) for either the entire window and/or a different colour for the window versus the `pg_ref`'s page area.

12.10 Transparent Colour

The colour that is specified as "transparent" effectively tells OpenPaige: "Leave the background alone if the page area's background is the transparent colour."

For most situations, you can leave the transparent colour as its default — white.

Here is an example, however, where you might need to change the transparent color. Suppose that your whole window is always blue but you want OpenPaige to draw on a white background. In this case, you would set the transparent colour to something other than "white" so OpenPaige is forced to set a white background. Otherwise, OpenPaige will not change the background at all when it draws text since it assumes the window is already in that colour.

12.11 Setting/Getting the Background Colour

```
(void) pgSetPageColor (pg_ref pg, color_value_ptr color);
(void) pgGetPageColor (pg_ref pg, color_value_ptr color);
```

To change the page area background colour, call `pgSetPageColor`. The new background colour will be copied from the `color` parameter.

To obtain the current page colour, use `pgGetPageColor` and the background colour of `pg` is copied to `*color`.

After changing the background, subsequent drawing will fill the page area with that colour before

text is drawn.

NOTE: `pgSetPageColor` does not redraw anything.

12.12 Getting/Changing the Transparent Colour

The "transparent colour" is a global value, as a field in `pg_ref`. Hence, all `pg_ref`s will check for the transparent colour by looking at this field.

If you need to swap different transparent colours in and out for different situations, simply change `pg_globals → trans_color` to the desired value.

NOTE: Usually the only time you need to change the transparent colour to something other than its default (white) is the following scenario: Non-white background colour for the whole window, but white background for a `pg_ref`'s page area. In every other situation it is safe to leave the transparent colour in `pg_globals` alone.

12.13 Miscellaneous Utilities

```
(void) pgErasePageArea (pg_ref pg, shape_ref vis_area);
```

This function fills `pg`'s page area with the current page background color of `pg`.

The fill is clipped to the page area intersected with the shape given in the `vis_area` parameter. However, if `vis_area` is a null pointer, then the `vis_area` in `pg` is used to intersect instead.

NOTE: You do not normally need to call this function: OpenPaige fills the appropriate areas(s) automatically when it draws text. This function exists for special situations where you want to "erase" the page area.

12.14 OpenPaige Background Colours

The purpose of this section is to provide some additional information about OpenPaige "background" colours and their relationship to the window's background colour.

First, let's clarify the difference between three different aspects of background:

- *Page background colour* is the colour that fills the background of your page area. The "page area" is the specific area in the `pg_ref` in which text flows, or wraps. This is not necessarily the same colour as the window's background colour. For instance, if the page area were smaller than the window that contained it, the page background would fill only the page area, while the remaining window area would remain unchanged.
- *Window background colour* is the background colour of the window itself. This can be different than the window's background colour.
- *Text background colour* is the background colour of text characters, applied as a style (just as italic, bold, underline, etc. is applied to text characters). Text background colour applies only to the text character itself. This can be different from both window background and page background.

12.15 Who/What Controls Colors

When creating new OpenPaige objects, the page area background colour is purely determined by the `def_bk_color` member of OpenPaige globals. Afterwards, this colour can be changed by calling `pgSetPageColor()`.

The window background colour is purely controlled by your application and no OpenPaige functions alter that colour.

Text background is controlled by changing the `bk_color` member of `style_info`, and that color applies only to the character(s) of that particular style.

12.16 What is "trans_color" in OpenPaige globals?

The purpose of `pg_globals.trans_color` is to define the default `WINDOW` background. Since OpenPaige is a portable library, the `trans_color` member is provided as a platform-independent method for OpenPaige to know what the "normal" background colour is.

OpenPaige uses `trans_color` only as a reference. Essentially, `trans_color` defines the colour that would appear if OpenPaige left the window alone, or the colour that would be used by the operating system if the window were "erased".

The value of `trans_color` becomes the most significant when you have set the page and/or text color to something different to the window color, because OpenPaige compares the page and text colors to `trans_color` to determine whether or not to `ERASE` the background.

Its reasoning is, "... If the background color I am to draw *is not* the "normal" background color

[trans_color], then I need to force-fill the background."

Conversely, "... If the background color I am to draw *is* the same as trans_color, then I don't have to set anything special".

Herein is most of the difficulty that OpenPaige users encounter with background colors: they set the window to a non-white background, yet they usually leave pg_globals.trans_color alone. This is OK as long as trans_color and the page area colour are different.

But if you want the page background and window background to be the same, make sure pg_globals.trans_color is the same as the page background color. The general rules are:

1. Always set pg_globals.trans_color to the same value as the window's background color. Do this regardless of what the page area background color will be.
2. The only time you need to change pg_globals.trans_color is when/if you have changed the window's background color to something other than what is already in pg_globals.trans_color.
3. Setting page and/or text colour has nothing to do with the window's real background colour. These may or may not be the same, and OpenPaige only knows if they match the window by comparing them to trans_color.
4. To make the page area AND the window backgrounds match each other, you must set pg_globals.trans_color, pgSetPageColor() and the window background colour to the same colour value.